interface definition, the caller specifies the data types it would like to receive. This flexibility makes it easier for diverse kinds of callers to invoke the service.

REST/HTTP is popular for its speed and simplicity. Web Services require parameters in SOAP messages to be represented in XML, which is expensive to parse. XML is self-describing and highly interoperable, but these benefits are not always important, for example, for simple services. A very simple interface makes it easier and faster to manipulate in limited languages such as JavaScript.

Hardware Architecture

The computers that run these programs have a range of processing power. A display device could be a character-at-a-time terminal, a handheld device, a low-end PC, or a powerful workstation. Front-end programs, request controllers, transaction servers, and database systems could run on any kind of server machine, ranging from a low-end server machine, to a high-end multiprocessor mainframe, to a distributed system. A distributed system could consist of many computers, localized within a machine room or campus or geographically dispersed in a region or worldwide.

Some of these systems are quite small, such as a few display devices connected to a small machine on a PC Local Area Network (LAN). Big TP systems tend to be enterprise-wide or Internet-wide, such as airline and financial systems, Internet retailers, and auction sites. The big airline systems have on the order of 100,000 display devices (terminals, ticket printers, and boarding-pass printers) and thousands of disk drives, and execute thousands of transactions per second at their peak load. The biggest Internet systems have hundreds of millions of users, with tens of millions of them actively using the system at any one time.

Given this range of capabilities of computers that are used for TP, we need some terminology to distinguish among them. We use standard words for them, but in some cases with narrower meanings than is common in other contexts.

We define a **machine** to be a computer that is running a single operating system image. It could use a single-core or multicore processor, or it could be a shared-memory multiprocessor. Or it might be a virtual machine that is sharing the underlying hardware with other virtual machines. A **server machine** is a machine that executes programs on behalf of client programs that typically execute on other computers. A **system** is a set of one or more machines that work together to perform some function. For example, a **TP system** is a system that supports one or more TP applications. A **node** (of a network) is a system that is accessed by other machines as if it were one machine. It may consist of several machines, each with its own network address. However, the system as a whole also has a network address, which is usually how other machines access it.

A **server process** is an operating system process, *P*, that executes programs on behalf of client programs executing in other processes on the same or different machines as the one where *P* is running. We often use the word “server” instead of “server machine” or “server process” when the meaning is obvious from context.

1.3 ATOMICITY, CONSISTENCY, ISOLATION, AND DURABILITY

There are four critical properties of transactions that we need to understand at the outset:

- Atomicity: The transaction executes completely or not at all.
- Consistency: The transaction preserves the internal consistency of the database.
- Isolation: The transaction executes as if it were running alone, with no other transactions.
- Durability: The transaction’s results will not be lost in a failure.

This leads to an entertaining acronym, ACID. People often say that a TP system executes ACID transactions, in which case the TP system has "passed the ACID test." Let's look at each of these properties in turn and examine how they relate to each other.

Atomicity

First, a transaction needs to be **atomic** (or **all-or-nothing**), meaning that it executes completely or not at all. There must not be any possibility that only part of a transaction program is executed.

For example, suppose we have a transaction program that moves $100 from account A to account B. It takes $100 out of account A and adds it to account B. When this runs as a transaction, it has to be atomic—either both or neither of the updates execute. It must not be possible for it to execute one of the updates and not the other.

The TP system guarantees atomicity through database mechanisms that track the execution of the transaction. If the transaction program should fail for some reason before it completes its work, the TP system will undo the effects of any updates that the transaction program has already done. Only if it gets to the very end and performs all of its updates will the TP system allow the updates to become a permanent part of the database.

If the TP system fails, then as part of its recovery actions it undoes the effects of all updates by all transactions that were executing at the time of the failure. This ensures the database is returned to a known state following a failure, reducing the requirement for manual intervention during restart.

By using the atomicity property, we can write a transaction program that emulates an atomic business transaction, such as a bank account withdrawal, a flight reservation, or a sale of stock shares. Each of these business actions requires updating multiple data items. By implementing the business action by a transaction, we ensure that either all the updates are performed or none are.

The successful completion of a transaction is called **commit**. The failure of a transaction is called **abort**.

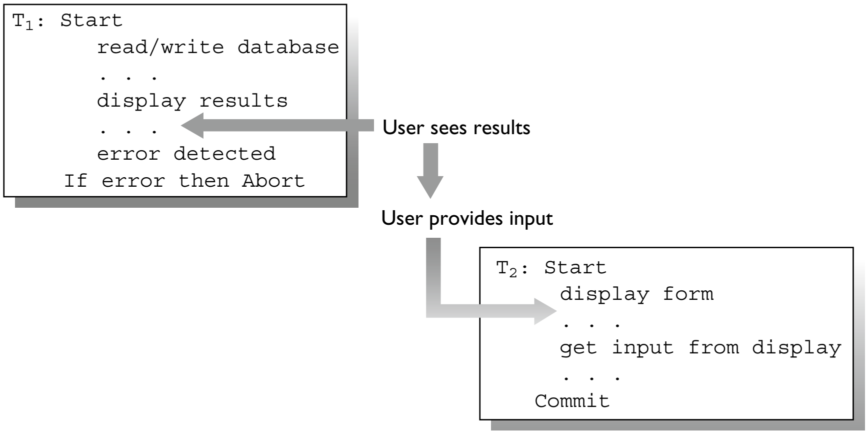
Handling Real-World Operations

During its execution, a transaction may produce output that is displayed back to the user. However, since the transaction program is all-or-nothing, until the transaction actually commits, any results that the transaction might display to the user should not be taken seriously, because it's still possible that the transaction will abort. Anything displayed on the display device could be wiped out in the database on abort.

Thus, any value that the transaction displays may be used by the end-user only if the transaction commits and not if the transaction aborts. This requires some care on the part of users (see Figure 1.4). If the system actually displays some of the results of a transaction before the transaction commits, and if the user utilizes any of these results as input to another transaction, then we have a problem. If the first transaction aborts and the second transaction commits, then the all-or-nothing property has been broken. That is, some of the results of the first transaction will be reflected in the results of the second transaction. But other results of the first transaction, such as its database updates, were not performed because the transaction aborted.

Some systems solve this problem simply by not displaying the result of a transaction until after the transaction commits, so the user can't inadvertently make use of the transaction's output and then have it subsequently abort. But this too has its problems (see Figure 1.5): If the transaction commits before displaying any of its results, and the system crashes before the transaction actually displays any of the results, then the user won't get a chance to see the output. Again, the transaction is not all-or-nothing; it executed all its database updates before it committed, but did not display its output.

We can make the problem more concrete by looking at it in the context of an automated teller machine (ATM) (see Figure 1.6). The output, for example, may be an operation that dispenses $100 from the ATM. If the system dispenses the $100 before the transaction commits, and the transaction ends up aborting, then the

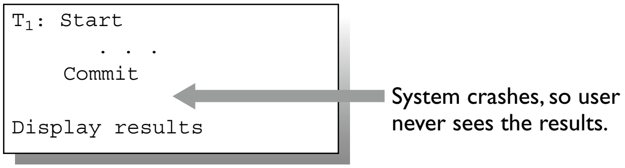


```
T1: Start
      read/write database
      . . .
      display results
      . . .
      error detected
    If error then Abort
```

```
T2: Start
      display form
      . . .
      get input from display
      . . .
    Commit
```

FIGURE 1.4

Reading Uncommitted Results. The user read the uncommitted results of transaction T_1 and fed them as input to transaction T_2. Since T_1 aborts, the input to T_2 is incorrect.



```
T1: Start
        . . .
    Commit

Display results
```

FIGURE 1.5

Displaying Results after Commits. This solves the problem of Figure 1.4, but if the transaction crashes before displaying the results, the results are lost forever.

bank gives up the money but does not record that fact in the database. If the transaction commits and the system fails before it dispenses the $100, then the database says the $100 was given to the customer, but in fact the customer never got the money. In both cases, the transaction's behavior is not all-or-nothing.

A closely-related problem is that of ensuring that each transaction executes exactly once. To do this, the transaction needs to send an acknowledgment to its caller, such as sending a message to the ATM to dispense money, if and only if it commits. However, sending this acknowledgment is not enough to guarantee exactly-once behavior because the caller cannot be sure how to interpret the absence of an acknowledgment. If the caller fails to receive an acknowledgment, it might be because the transaction aborted, in which case the caller needs to resubmit a request to run a transaction (to ensure the transaction executes once). Or it might be that the transaction committed but the acknowledgment got lost, in which case the caller must not resubmit a request to run the transaction because that would cause the transaction to execute twice. So if the caller wants exactly-once behavior, it needs to be sure that a transaction did not and will not commit before it's safe to resubmit the request to run the transaction.

Although these seem like unsolvable problems, they can actually be solved using persistent queues, which we'll describe in some detail in Chapter 4.

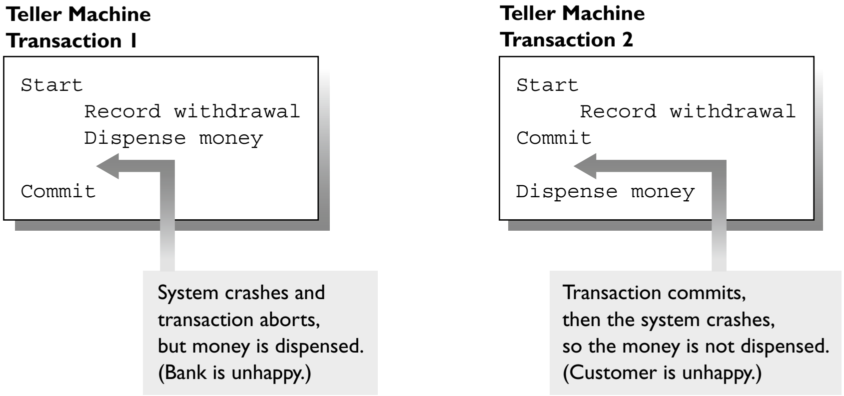


FIGURE 1.6

The Problem of Getting All-or-Nothing Behavior with Real-World Operations. Whether the program dispenses money before or after it commits, it's possible that only one of the operations executes: dispense the money or record the withdrawal.

Compensating Transactions

Commitment is an irrevocable action. Once a transaction is committed, it can no longer be aborted. People do make mistakes, of course. So it may turn out later that it was a mistake to have executed a transaction that committed. At this point, the only course of action is to run another transaction that reverses the effect of the one that committed. This is called a **compensating transaction**. For example, if a deposit transaction was in error, then one can later run a withdrawal transaction that reverses its effect.

Sometimes, a perfect compensation is impossible, because the transaction performed some irreversible act. For example, it may have caused a paint gun to spray-paint a part the wrong color, and the part is long gone from the paint gun's work area when the error is detected. In this case, the compensating transaction may be to record the error in a database and send an e-mail message to someone who can take appropriate action.

Virtually any transaction can be executed incorrectly. So a well-designed TP application should include a compensating transaction type for every type of transaction.

Multistep Business Processes

Some business activities do not execute as a single transaction. For example, the activity of recording an order typically executes in a separate transaction from the one that processes the order. Since recording an order is relatively simple, the system can give excellent response time to the person who entered the order. The processing of the order usually requires several time-consuming activities that may require multiple transactions, such as checking the customer's credit, forwarding the order to a warehouse that has the requested goods in stock, and fulfilling the order by picking, packing, and shipping it.

Even though the business process executes as multiple transactions, the user may still want atomicity. Since multiple transactions are involved, this often requires compensating transactions. For example, if an order is accepted by the system in one transaction, but later on another transaction determines that the order can't be fulfilled, then a compensating transaction is needed to reverse the effect of the transaction that accepted the order. To avoid an unhappy customer, this often involves the universal compensating transaction, namely, an apology and a free gift certificate. It might also involve offering the customer a choice of either cancelling or telling the retailer to hold the order until the requested items have been restocked.

Transactional middleware can help manage the execution of multistep business processes. For example, it can keep track of the state of a multistep process, so if the process is unable to complete then the middleware can invoke compensating transactions for the steps that have already executed. These functions and others are discussed in Chapter 5, *Business Process Management*.

Consistency

A second property of transactions is consistency—a transaction program should maintain the consistency of the database. That is, if you execute the transaction all by itself on a database that's initially consistent, then when the transaction finishes executing the database is again consistent.

By consistent, we mean "internally consistent." In database terms, this means that the database at least satisfies all its integrity constraints. There are several kinds of integrity constraints that database systems can typically maintain:

- All primary key values are unique (e.g., no two employee records have the same employee number).
- The database has referential integrity, meaning that records reference only objects that exist (e.g., the Part record and Customer record that are referenced by an Order record really exist).
- Certain data values are in a particular range (e.g., age is less than 120 and social security number is not null).

There are other kinds of integrity constraints that database systems typically cannot maintain but may nevertheless be important, such as the following:

- The sum of expenses in each department is less than or equal to the department's budget.
- The salary of an employee is bounded by the salary range of the employee's job level.
- The salary of an employee cannot decrease unless the employee is demoted to a lower job level.

Ensuring that transactions maintain the consistency of the database is good programming practice. However, unlike atomicity, isolation, and durability, consistency is a responsibility shared between transaction programs and the TP system that executes those programs. That is, a TP system ensures that transactions are atomic, isolated, and durable, whether or not they are programmed to preserve consistency. Thus, strictly speaking, the ACID test for transaction systems is a bit too strong, because the TP system does its part for the C in ACID only by guaranteeing AID. It's the application programmer's responsibility to ensure the transaction program preserves consistency.

There are consistency issues that reach out past the TP system and into the physical world that the TP application describes. An example is the constraint that the number of physical items in inventory equals the number of items on the warehouse shelf. This constraint depends on actions in the physical world, such as correctly reporting the restocking and shipment of items in the warehouse. Ultimately, this is what the enterprise regards as consistency.

Isolation

The third property of a transaction is called **isolation**. We say that a set of transactions is isolated if the effect of the system running them is the same as if the system ran them one at a time. The technical definition of isolation is serializability. An execution is **serializable** (meaning isolated) if its effect is the same as running the transactions serially, one after the next, in sequence, with no overlap in executing any two of them. This has the same effect as running the transactions one at a time.

A classic example of a non-isolated execution is a banking system, where two transactions each try to withdraw the last $100 in an account. If both transactions read the account balance before either of them updates it, then both transactions will determine there's enough money to satisfy their requests, and both will withdraw the last $100. Clearly, this is the wrong result. Moreover, it isn't a serializable result. In a serial execution, only the first transaction to execute would be able to withdraw the last $100. The second one would find an empty account.

Notice that isolation is different from atomicity. In the example, both transactions executed completely, so they were atomic. However, they were not isolated and therefore produced undesirable behavior.

If the execution is serializable, then from the point of view of an end-user who submits a request to run a transaction, the system looks like a standalone system that's running that transaction all by itself. Between the time he or she runs two transactions, other transactions from other users may run. But during the period that the system is processing that one user's transaction, the user has the illusion that the system is doing no other work. This is only an illusion. It's too inefficient for the system to actually run transactions serially, because there is a lot of internal parallelism in the system that must be exploited by running transactions concurrently.

If each transaction preserves consistency, then any serial execution (i.e., sequence) of such transactions preserves consistency. Since each serializable execution is equivalent to a serial execution, a serializable execution of the transactions will preserve database consistency too. It is the combination of transaction consistency and isolation that ensures that the execution of a set of transactions preserves database consistency.

The database typically sets locks on data accessed by each transaction. The effect of setting the locks is to make the execution appear to be serial. In fact, internally, the system is running transactions in parallel, but through this locking mechanism the system gives the illusion that the transactions are running serially, one after the next. In Chapter 6, we will describe those mechanisms in more detail and present the rather subtle argument why locking actually produces serializable executions.

A common misconception is that serializability isn't important because the database system will maintain consistency by enforcing integrity constraints. However, as we saw in the previous section on consistency, there are many consistency constraints that database systems can't enforce. Moreover, sometimes users don't tell the database system to enforce certain constraints because they degrade performance. The last line of defense is that the transaction program itself maintains consistency and that the system guarantees serializability.

Durability

The fourth property of a transaction is durability. **Durability** means that when a transaction completes executing, all its updates are stored in **stable storage**; that is, storage that will survive the failure of power or the operating system. Today, stable storage (also called **nonvolatile** or **persistent storage**) typically consists of magnetic disk drives, though solid-state disks that use flash memory are making inroads as a viable alternative. Even if the transaction program fails, or the operating system fails, once the transaction has committed, its results are durably stored on stable storage and can be found there after the system recovers from the failure.

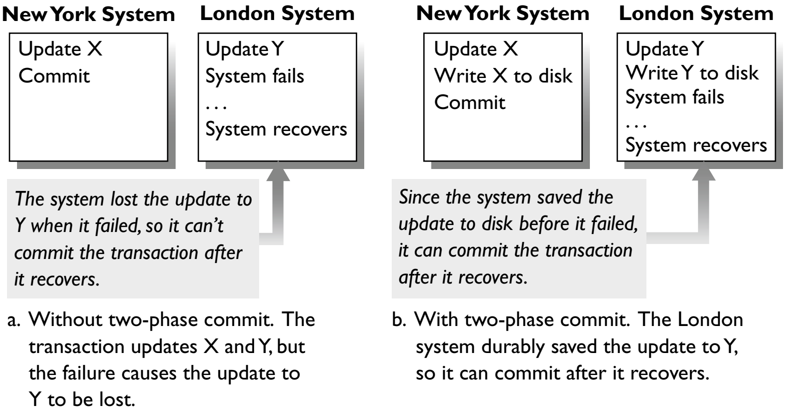
Durability is important because each transaction usually is providing a service to the user that amounts to a contract between the user and the enterprise that is providing the service. For example, if you're moving money from one account to another, once you get a reply from the transaction saying that it executed, you expect that the result is permanent. It's a legal agreement between the user and the system that the money has been moved between these two accounts. So it's essential that the transaction actually makes sure that the updates are stored on some stable storage device, to ensure that the updates cannot possibly be lost after the transaction finishes executing. Moreover, the durability of the result must be maintained for a long period, until it is explicitly overwritten or deleted by a later transaction. For example, even if a checking account is unused for several years, the owner expects to find her money there the next time she accesses it.

The durability property usually is obtained by having the TP system append a copy of all the transaction's updates to a log file while the transaction program is running. When the transaction program issues the commit operation, the system first ensures that all the records written to the log file are out on stable storage, and then returns to the transaction program, indicating that the transaction has indeed committed and that the results are durable. The updates may be written to the database right away, or they may be written a little later. However, if the system fails after the transaction commits and before the updates go to the database, then after the system recovers from the failure it must repair the database. To do this, it reads the log and checks that each update by a committed transaction actually made it to the database. If not, it reapplies the update to the database. When this recovery activity is complete, the system resumes normal operation. Thus, after the system recovers, any new transaction will read a database state that includes all the updates of transactions that committed before the failure (as well as those that committed after the recovery). We describe log-based recovery algorithms in Chapter 7.

1.4 TWO-PHASE COMMIT

When a transaction updates data on two or more database systems, we still have to ensure the atomicity property, namely, that either both database systems durably install the updates or neither does. This is challenging, because the database systems can independently fail and recover. This is certainly a problem when the database systems reside on different nodes of a distributed system. But it can even be a problem on a single machine if the database systems run as server processes with private storage since the processes can fail independently. The solution is a protocol called **two-phase commit** (**2PC**), which is executed by a module called the **transaction manager**.

The crux of the problem is that a transaction can commit its updates on one database system, but a second database system can fail before the transaction commits there too. In this case, when the failed system recovers, it must be able to commit the transaction. To commit the transaction, the recovering system must have a copy of the transaction's updates that executed there. Since a system can lose the contents of main memory when it fails, it must store a durable copy of the transaction's updates before it fails, so it will have them after it recovers. This line of reasoning leads to the essence of two-phase commit: Each database system accessed by a transaction must durably store its portion of the transaction's updates before the transaction commits anywhere. That way, if a system S fails after the transaction commits at another system S' but before the transaction commits at S, then the transaction can commit at S after S recovers (see Figure 1.7).



a. Without two-phase commit. The transaction updates X and Y, but the failure causes the update to Y to be lost.

b. With two-phase commit. The London system durably saved the update to Y, so it can commit after it recovers.

FIGURE 1.7

How Two-Phase Commit Ensures Atomicity. With two-phase commit, each system durably stores its updates before the transaction commits, so it can commit the transaction when it recovers.

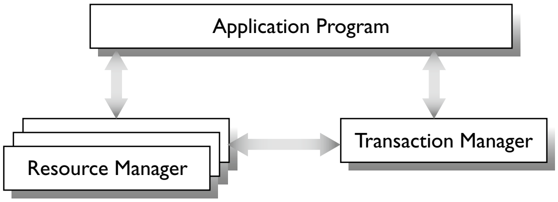


FIGURE 1.8

X/Open Transaction Model (XA). The transaction manager processes Start, Commit, and Abort. It talks to resource managers to run two-phase commit.

To understand two-phase commit, it helps to visualize the overall architecture in which the transaction manager operates. The standard model, shown in Figure 1.8, was introduced by IBM's CICS and popularized by Oracle's Tuxedo and X/Open (now part of The Open Group, see Chapter 10). In this model, the transaction manager talks to applications, resource managers, and other transaction managers. The concept of "resource" includes databases, queues, files, messages, and other shared objects that can be accessed within a transaction. Each resource manager offers operations that must execute only if the transaction that called the operations commits.

The transaction manager processes the basic transaction operations for applications: Start, Commit, and Abort. An application calls Start to begin executing a new transaction. It calls Commit to ask the transaction manager to commit the transaction. It calls Abort to tell the transaction manager to abort the transaction.

The transaction manager is primarily a bookkeeper that keeps track of transactions in order to ensure atomicity when more than one resource is involved. Typically, there's one transaction manager on each node of a distributed computer system. When an application issues a Start operation, the transaction manager dispenses a unique ID for the transaction called a **transaction identifier**. During the execution of the transaction, it keeps track of all the resource managers that the transaction accesses. This requires some cooperation with the application, resource managers, and communication system. Whenever the transaction accesses a new resource manager, somebody has to tell the transaction manager. This is important because when it comes time to commit the transaction, the transaction manager has to know all the resource managers to talk to in order to execute the two-phase commit protocol.

When a transaction program finishes execution and issues the commit operation, that commit operation goes to the transaction manager, which processes the operation by executing the two-phase commit protocol. Similarly, if the transaction manager receives an abort operation, it tells the resource managers to undo all the transaction's updates; that is, to abort the transaction at each resource manager. Thus, each resource manager must understand the concept of transaction, in the sense that it undoes or permanently installs the transaction's updates depending on whether the transaction aborts or commits.

When running two-phase commit, the transaction manager sends out two rounds of messages—one for each phase of the commitment activity. In the first round of messages it tells all the resource managers to prepare to commit by writing a copy of the results of the transaction to stable storage, but not actually to commit the transaction. At this point, the resource managers are said to be **prepared to commit**. When the transaction manager gets acknowledgments back from all the resource managers, it knows that the whole transaction has been prepared. That is, it knows that all resource managers stored a durable copy of the transaction's updates but none of them have committed the transaction. So it sends a second round of messages to tell the resource managers to actually commit. Figure 1.9 gives an example execution of two-phase commit with two resource managers involved.

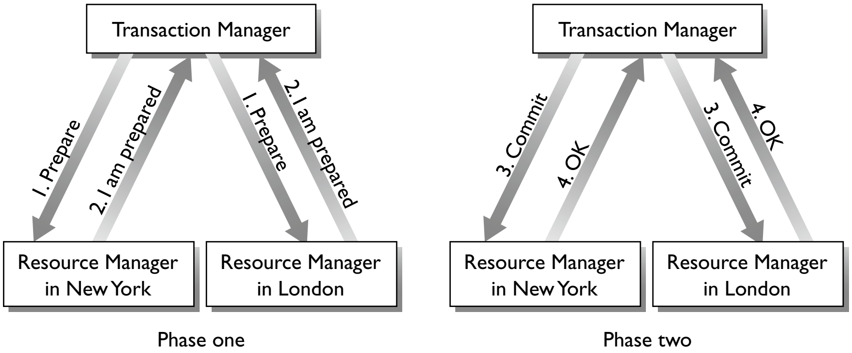


FIGURE 1.9

The Two-Phase Commit Protocol. In Phase One, every resource manager durably saves the transaction's updates before replying "I am Prepared." Thus, all resource managers have durably stored the transaction's updates before any of them commits in phase two.

Two-phase commit avoids the problem in Figure 1.7(a) because all resource managers have a durable copy of the transaction's updates before any of them commit. Therefore, even if a system fails during the commitment activity, as the London system did in the figure, it can commit the transaction after it recovers. However, to make this all work, the protocol must handle every possible failure and recovery scenario. For example, in Figure 1.7(b), it must tell the London system to commit the transaction. The details of how two-phase commit handles all these scenarios is described in Chapter 8.

Two-phase commit is required whenever a transaction accesses two or more resource managers. Thus, one key question that designers of TP applications must answer is whether or not to distribute their transaction programs among multiple resources. Using two-phase commit adds overhead (due to two-phase commit messages), but the option to distribute can provide better scalability (adding more systems to increase capacity) and availability (since one system can fail while others remain operational).

1.5 TRANSACTION PROCESSING PERFORMANCE

Performance is a critical aspect of TP systems. No one likes waiting more than a few seconds for an automated teller machine to dispense cash or for a hotel web site to accept a reservation request. So response time to end-users is one important measure of TP system performance. Companies that rely on TP systems, such as banks, airlines, and commercial web sites, also want to get the most transaction throughput for the money they invest in a TP system. They also care about system scalability; that is, how much they can grow their system as their business grows.

It's very challenging to configure a TP system to meet response time and throughput requirements at minimum cost. It requires choosing the number of systems, how much storage capacity they'll have, which processing and database functions are assigned to each system, and how the systems are connected to displays and to each other. Even if you know the performance of the component products being assembled, it's hard to predict how the overall system will perform. Therefore, users and vendors implement benchmarks to obtain guidance on how to configure systems and to compare competing products.

Vendor benchmarks are defined by an independent consortium called the Transaction Processing Performance Council (TPC; www.tpc.org). The benchmarks enable apples-to-apples comparisons of different vendors' hardware