DISTRIBUTED DATABASES

The importance of information in most organizations has led to the development of a large body of concepts and techniques for the efficient management of data. Distributing data across sites or departments in an organization allows those data to reside where they are generated or are most needed, but still to be accessible from other sites and from other departments. A *distributed database system* (DDS) is a software system that gives users transparent access to data, along with the ability to manipulate these data, in local databases that are distributed across the nodes of a communication network. Each of the local databases is managed by a local *database management system* (DBMS).

A distributed database system consists of multiple databases that are distributed across computers in a communication network. The individual computers can be personal computers (PCs), workstations, or mainframes. None of the machines share memory or disk space. The computers in a distributed database are referred to as *sites* or *nodes;* we mainly use the term *site* to emphasize the physical distribution of these systems.

Parallel database systems, which are seeing increased use, may also be designed as a set of databases connected by a high-speed local-area network. However, distributed databases can be distinguished from parallel database systems in several ways: The local databases of a distributed database are typically *geographically separated, separately administered,* and have a *slower interconnection.* The local databases also have a great deal of degree of *autonomy* in carrying out their functions such as concurrency control and recovery.

The field of distributed databases is well established, dating back to the late 1970s. In fact, several commercial implementations were built in the early 1980s, although they did not have much of commercial success. However, interest in distributed databases has greatly increased in the 1990s largely due to the explosive growth of networks, both the Internet and organization-wide intranets. Database systems developed independently are increasingly being coupled together across networks, to form organization-wide distributed databases.

Traditional online transaction processing (OLTP) applications have hitherto driven the area of distributed databases, with their need for access to remote databases and their high availability requirements. Online application continues to be an important motivator for distributed databases. However, as of the late 1990s, data warehousing applications are increasingly driving distributed database systems. Data warehouses collect data from multiple sources, integrate these data in a common format, and make them available for decision support applications. The growth of multiple database services on the World Wide Web, such as stock market information and trading systems, banking systems, and reservation systems, is also contributing to the growth of distributed database applications.

In this article we provide an introduction to the field of distributed databases. We begin with an overview of distributed databases, then provide a taxonomy of distributed databases. This follows with a description of the architecture of distributed databases and data allocation in distributed databases. The section entitled "Distributed Query Processing" covers query processing in distributed databases. The section entitled "Distributed Transaction Processing" provides an overview of transaction processing. This follows with a description of concurrency control and distributed commit processing. The section entitled "Replication of Data" describes replication issues in a distributed database system. This article concludes with an annotated list of key material for further study.

Distributed Database System Overview

To illustrate a distributed database system, let us consider a banking system that comprises four branches located in four different cities. Each branch has its own computer, with a database consisting of all the accounts maintained at that branch. Each such installation is thus a site in the distributed system. Each site maintains a relation *account* with the schema (*branch name, account number, balance*). In addition, the branch at the fourth site maintains information about all the branches of the bank. It maintains the relation *branch* with the schema (*branch name, branch city, assets*).

Figure 1 depicts this example distributed database banking system.

Our definition of a distributed database system does not mandate that all local DBMSs be the same; in fact, they may differ. There is also no requirement that the data models used at the local databases be identical. For example, one local site may use a hierarchical data model, whereas another may use a relational model and a third may use an object-oriented data model. If data models are different for different sites, the distributed database system must contain a sufficient information to translate data from one data model to another.

The sites may have no knowledge of one another, but the distributed database system must contain information about all of them. Each local site, in addition to containing its own DBMS, must contain an additional software layer to facilitate

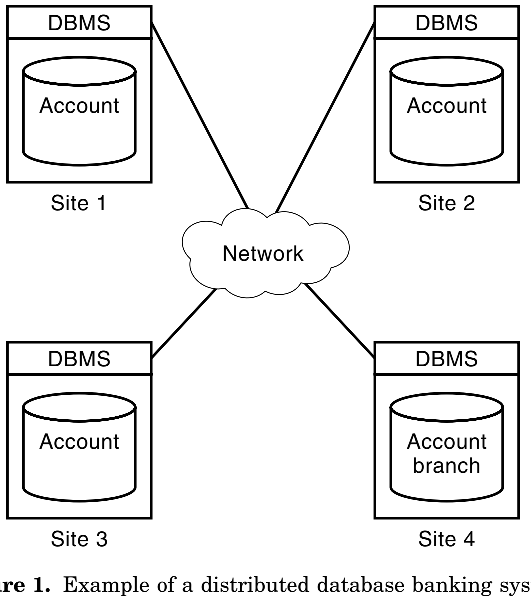

Figure 1. Example of a distributed database banking system.

coordination with other sites on behalf of a distributed database. Such coordination allows the distributed database system to enforce uniform processing of user requests regardless of whether local DBMSs are aware of one another.

Distributed Database System Advantages and Disadvantages

Distributed database system confer several important advantages over centralized systems.

- *Data Sharing.* A major advantage is that a distributed database system provides an environment where users at one site can access the data residing at other sites. For instance, in our example of a distributed banking system, a user in one branch can access data at another branch.
- *Autonomy.* The primary advantage of sharing data by means of data distribution is that each site retains a degree of control over its locally stored data. In a centralized system, the manager of the central site controls the entire database. In a distributed system, local managers have *local autonomy* in devising data policies—such as access, manipulation, and maintenance policies—at the site. Depending on the amount of the local autonomy, distributed systems fall into different categories (see the section entitled "Taxonomy of Distributed Database Systems"). The potential for local autonomy is often a major reason why an organization chooses to use a distributed database.
- *Availability.* If one site in a distributed system fails, or becomes unavailable due to communication or site failure, the remaining sites may be able to continue to operate. In particular, if data objects are replicated at several sites, a transaction that requires a particular data object may find that object at any of these sites. Thus, the failure of one site does not necessarily imply the shutdown of the system.

  The failure of one site must be detected by the system, and appropriate recovery action may need to be initiated. The system must stop relying on the services of the failed site. Finally, when the failed site recovers, mechanisms must be available to integrate it back into the system smoothly. Thus, recovery from failure is more complex in distributed systems than in centralized systems.
- *Enhanced Performance.* Data that reside in proximity to users can be accessed much faster than can data at remote sites. Furthermore, user requests for the data that are located at several sites can be processed in parallel at each site and shipped to the user's location. This parallel processing improves overall response time. For instance, if a user at site 4 in the distributed banking system requests a list of all accounts at all branches that have a balance of more than $200, then all four sites perform selection of such accounts in parallel and send the resulting list of accounts to site 4, where all these lists are coalesced into one. In contrast, if all accounts were located at a single site, the site processor could take four times as much time to select the requested accounts.
- *Expandability.* It is much easier to expand a distributed database by adding new data or new sites, in contrast to expanding of a centralized database, where the maintenance procedure results in (1) the database being unavailable to users and (2) user services being interrupted. In a distributed database system, the addition of a new site has no effect on current data processing. Data maintenance procedures are performed on a per-site basis; consequently, users always have access to the overall database system, although an individual site may be unavailable.

The primary disadvantage of a distributed database systems is the added complexity required to ensure proper coordination among the sites. In fact, several of the advantages listed (such as enhanced performance) could in certain circumstances become disadvantages. For example, if a user tries to update a data object that is replicated at several sites, the system needs to coordinate updates at all the sites to ensure that either all the sites have the new value or none of them has the new value. Among the disadvantages of distributed database systems compared to centralized systems are these:

- *Higher Software Development Cost.* Implementing a distributed database system is complicated and costly.
- *Greater Potential for Bugs.* The sites that constitute the distributed system operate in parallel, so it is hard to ensure the correctness of algorithms, especially operation during failures of part of the system and during recovery from failures. The potential exists for extremely subtle bugs.
- *Increased Processing Overhead.* The exchange of messages and the additional computation required to achieve intersite coordination add an overhead cost.
- *Decreased Security.* Distribution of data among several sites creates several entrance points for potential malicious users. It is more difficult and more expensive to design and enforce security procedures when data can be accessed from several sites, each of which may have its own security policy. In addition, data are transferred over the communication network, making it possible for people to intercept the data.

Transparency

The user of a distributed database system should not be required to know either where the data are physically located or how the data can be accessed at the specific local site. This characteristic, called *data transparency,* can take several forms:

- *Replication Transparency.* Users view each data object as logically unique. The distributed system may replicate an object to increase either system performance or data availability. Users should not be concerned with *what* data objects need to be replicated, *when* the replication should occur, or *where* new replicas should be placed.
- *Data Naming Transparency.* Each data object has a unique name in the distributed system. Since a system includes several autonomous local data sites, the same data object may have different names at different sites, and different data objects at different sites may share the same name. The distributed database system should always be able to find a unique data object that is requested by the user. For example, a distributed database

system may prefix each object name with the name of the site at which that object is located.

- *Location Transparency.* Users are not required to know the physical location of the data. The distributed database system should be able to find any datum as long as the data identifier is supplied by the user transaction.
- *Data Definition Transparency.* The same data object may have been defined differently at different local sites. For example, *date of birth* in one local database could be defined as string of six characters, whereas at the other local site it is defined as a string of eight characters. Users are not required to know the details of the object definitions used at each local site. The distributed database system provides a user with a single definition of the data object, and it translates the user data object definition to the definitions used at the local site where the data object is located.
- *Data Representation Transparency.* The user should not be concerned about how a data object is physically represented in each local site.
- *Network Topology Transparency.* A distributed database can be defined for a set of sites regardless of how those sites are interconnected. The only requirement is that any two sites be able to exchange messages. Users are not required to know the details of the network topology to access and manipulate the data.

TAXONOMY OF DISTRIBUTED DATABASE SYSTEMS

Figure 2 depicts a classification of distributed-database systems, based on the level of local sites' cooperation within the system and on differences among the local DBMS software.

In a *homogeneous distributed database system* (DDB), local sites have identical DBMS software, are aware of one another, and agree to cooperate in processing users requests. In such a system, local sites surrender a portion of their autonomy. A site can no longer process user requests without consulting other sites. For example, a site cannot unilaterally decide to commit changes that a transaction has made to its local database; it must instead coordinate its actions with other sites. Local DDB sites share their local DBMS control information with other sites. Each site has an identical local DBMS, and there is a global schema such that each local database schema is a view of the global schema. This global schema makes it relatively easy to develop a distributed database system, because the system can enforce global query and transaction processing, as well as security policies.

Next, we consider a *heterogeneous distributed database system* (HeDBS). We say that local DBMS and local database schemas are heterogeneous if they employ possibly different data models, different query and transaction-processing algorithms, and different notions of local data objects. HeDBS sites are not aware of one another; consequently, it is difficult to enforce cooperation among local sites that are processing parts of a single transaction.

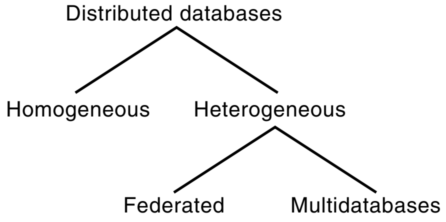

Figure 2. Taxonomy of distributed database systems.

Suppose that in our distributed-banking example, each local DBMS is relational but was developed by a different DBMS software vendor. Further suppose that the definition of the *account* relation differs across sites (say two branches require a customer date of birth on the account, whereas the two other sites require the number of the customer's dependents). If the user requests a list of all accounts and the date of birth for a given account owner and if the same user has an account in two branches, two sites may have the information to fulfill the request. Consequently, a distributed database system software must be able to determine whether information missing at one site is available from some other site; in our example, it should be able to match accounts for a named customer, even if those accounts are located at different branches.

There are several types of heterogeneous database systems. In *federated distributed database systems* (FDB), local sites have more autonomy than in a homogeneous distributed database. Each FDB site creates an import–export schema of data that it is willing to share with other sites, and on which it is willing to cooperate with other sites in processing user requests. For example, the site may indicate that the account-owner name and the account balance are available to other sites and that, for these data, it is willing to participate in implementing distributed database systems data access and transaction management policies. In an FDB, there is no global schema. Each site does have a local database schema, as well as the view of the data for off-site users (the import–export schema).

In recent years, new database applications have been developed that require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software environments. A *multidatabase system* is a distributed database system where local sites retain full autonomy; they are not aware of one another and are not prepared to share their local data access and transaction management information. A multidatabase system creates the illusion of logical database integration without requiring physical database integration. To enforce cooperation of local sites, the multidatabase must not only coordinate execution of user requests, but also reimplement its own access and transaction processing policies to be enforced outside of local DBMSs.

In this article, we are primarily concerned with homogeneous distributed databases. However, in the section entitled "Multidatabase Concurrency Control," we discuss briefly transaction management in federated databases and in multidatabases.

DISTRIBUTED DATABASE ARCHITECTURE

There are two architectural models for distributed database systems: system architecture and schema architecture. The *system architecture* describes interactions among different system components of a distributed database system and between local DBMSs and system components. The *schema ar-*

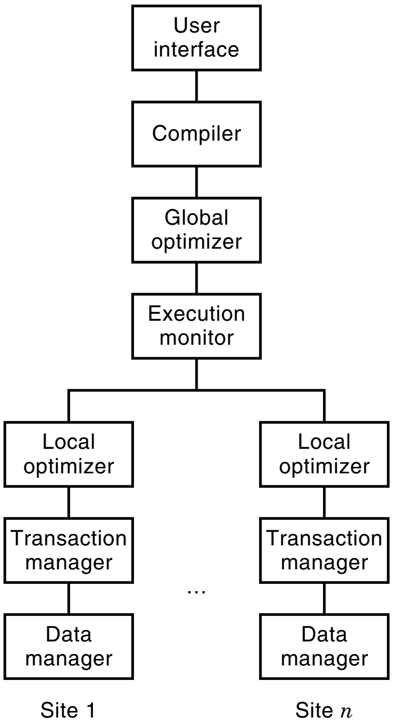

Figure 3. System architecture.

chitecture outlines an application, enterprise, and local site view of the data in the distributed database.

System Architecture

Figure 3 depicts the system architecture of a distributed database system. The *user interface* accepts user requests and translates them into the language of the distributed database, and it also represents data from the database in the form that the user expects. The *compiler* checks the syntactic correctness of the data requests, and it validates requests against security and against other system level restrictions on data. The *global query optimizer* designs an execution plan for accessing or updating data that a user has requested. The job of the optimizer is to find a plan that minimizes the request response time and data transfers between the different sites during the query processing. The *execution monitor* oversees carrying out of the requests at different sites and ensures data consistency and atomicity of any requests that require intersite communication.

After receiving the portion of a user request that the execution monitor has sent to the site, the *local query optimizer* at each site devises a local execution plan to obtain the local data in the fastest possible way. The *transaction manager* and the *data manager* at each site guarantee atomicity, consistency, isolation, durability (ACID) transaction properties at that site; global ACID transaction properties are ensured by the execution monitor.

Schema Architecture

Figure 4 depicts a schema architecture of the distributed database system. Data in a distributed database system are usually fragmented and replicated. Fragmentation and replication of data create the problem of how to represent a global data view. Each user application creates its own view of the data represented in the distributed database. An application view is called a *user view.* Various users views are combined into a global view of the data, represented by a *global conceptual schema.* At each local site, a *local conceptual schema* provides transparency of data naming and data representation. The *global directory* contains the mapping of global data objects into various users views, on one hand, and into various local conceptual schemas, on the other hand. Each local DBMS schema is represented by a *local internal schema.* A local directory maintained at each local site describes the differences between the local data representation in the local DBMS and the way the data is seen by external users. Finally, a *local storage schema* describes how the data are actually stored in the local database, and it also defines data keys and access indices.

Schema Integration

Conceptually, each relation in the global schema is defined as a view on relations from the local schemas. Schema integration is not, however, simply straightforward translation be-

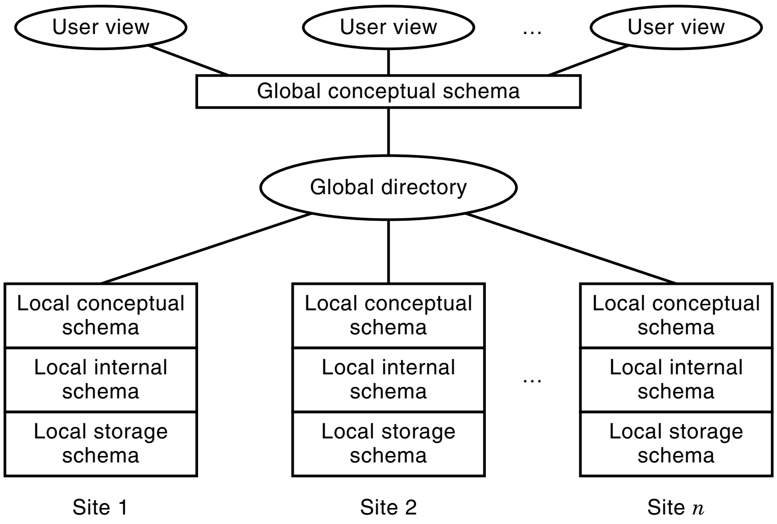

Figure 4. Schema integration architecture.

tween data definition languages; it is a complicated task due to semantic heterogeneity.

For example, the same attribute names may appear in different local databases but represent different meanings. The data types used in one system may not be supported by other systems, and translation between types may not be simple. Even for identical data types, problems may arise due to the physical representation of data. One system may use ASCII, while another may use EBCDIC. Floating-point representations may differ. Integers may be represented in *big-endian* or *little-endian* form. At the semantic level, an integer value for length may be inches in one system and millimeters in another, thus creating an awkward situation in which equality of integers is only an approximate notion (as is always the case for floating-point numbers). The same name may appear in different languages in different systems. For example, a system based in the United States may refer to the city "Cologne," whereas one in Germany refers to it as "Köln."

All these seemingly minor distinctions must be properly recorded in the common global conceptual schema. Translation functions must be provided. Indices must be annotated for system-dependent behavior (for example, the sort order of nonalphanumeric characters is not the same in ASCII as in EBCDIC). As we noted earlier, the alternative of converting each database to a common format may not be feasible because it may obsolete existing application programs.

Useful global query optimizations are possible if further information about sites is provided. For example, suppose that site 1 contains only accounts whose branch name is A. Such information is sometimes referred to as a *site description,* and it can be formally specified by defining the local data at the site as a selection on the global schema. Given the site description for the preceding example, queries that request account data for branch B do not need to access site 1 at all.

In the recent past, numerous databases have become available on the World Wide Web. In some cases the data in these databases are *structured* in the traditional database sense. In other cases the data consist of unstructured documents. Integration of data from multiple databases and optimizing queries posed on the integrated schema are topics of ongoing research.

DATA ALLOCATION

Consider a relation r that is to be stored in the database. There are several approaches to storing this relation in the distributed database:

- *Replication.* The system maintains several identical replicas (copies) of the relation. Each replica is stored at a different site. A relation is said to be *fully replicated* if a replica of the relation is stored at every site in the distributed database. If more than one, but not all, sites have a replica, the relation is said to be *partially replicated.*
- *Fragmentation.* The relation is partitioned into several fragments. Each fragment is stored at a different site.
- *Replication and Fragmentation.* The relation is partitioned into several fragments. The system maintains several replicas of each fragment.

Replication provides the following advantages:

- *Availability.* If one of the sites containing relation r fails, then r can be found in another site. Thus, the system can continue to process queries that require r, despite the failure of one site.
- *Increased Parallelism.* When the majority of accesses to the relation r result in only the reading of the relation, then several sites can process in parallel queries involving r. The more replicas of r, the greater the chance that the needed data will be found at the site where the transaction is executing. Hence, data replication minimizes movement of data among sites.

In general, replication enhances the performance of *read* operations and increases the availability of data to read-only transactions. However, *update* transactions incur greater overhead, since the update must be propagated to every replica. We can simplify the management of replicas of a relation r by choosing one of them as the *primary copy.* For example, in a banking system, an account can be associated with the site at which it was opened.

If relation r is *fragmented,* r is divided into multiple *fragments:* $r_1, r_2, \ldots, r_n$. These fragments contain sufficient information to allow reconstruction of the original relation r. There are two different schemes for fragmenting a relation:

- *Horizontal fragmentation* splits the relation by assigning each tuple of r to one or more fragments. The set of tuples in a fragment is determined by applying a selection operation on the relation r.
- *Vertical fragmentation* splits the relation by decomposing the scheme R of relation r into several subsets $R_1, R_2, \ldots, R_n$ such that $R = R_1 \cup R_2 \cup \ldots \cup R_n$. The fragmentation should be done such that we can reconstruct relation r from the fragments by taking the natural join of all vertical fragments r_i.

These two schemes can be applied successively to the same relation, resulting in many different fragments. Note that certain information may appear in several fragments.

In many distributed databases, the local relations already exist and the global schema is defined later as a view on the local schema. Thus a global relation could be a view defined, for example, as the join of several local relations or as the union of several local relations. In such a case, a join can be viewed as integrating data about the same entities from different local databases, whereas a union can be viewed as integrating data about different entities stored in different local databases. More complex expressions involving combinations of joins, unions, and other relational operations could also be used in defining the global view.

DISTRIBUTED QUERY PROCESSING

The main purpose of query optimization in a distributed database system is to reduce the costs of processing of user requests. The processing costs are determined by the usage of CPU, disk, and network resources. However, the ultimate goal is to provide users with the fastest possible response time. Evaluating joins is the most expensive part of distributed query processing, so the choice of join strategy is critical.

Simple Scheme

Consider a join of three relations: r_1, r_2, and r_3. Assume that the three relations are neither replicated nor fragmented and that r_1 is stored at site s_1, r_2 at s_2, and r_3 at s_3. Let s_I denote the site at which the query was issued. The system needs to produce the result at site s_I. Among the possible strategies for processing this query are the following:

- Ship copies of all three relations to site s_I, and apply centralized database query optimization strategies to process the entire query locally at site s_I.
- Ship a copy of the r_1 relation to site s_2; and compute $temp_1$, which is a join of r_1 and r_2. Ship $temp_1$ from s_2 to s_3, and compute $temp_2$ as a join of $temp_1$ and r_3. Ship the result $temp_2$ to s_I.
- Devise strategies similar to the previous one, but with the roles of s_1, s_2, and s_3 exchanged.

No one strategy is always the best choice. Among the factors that must be considered are the volume of data being shipped, the cost of transmitting a block of data between a pair of sites, and the relative speed of processing at each site.

Semijoins

Suppose that we wish to evaluate a join of r_1 and r_2, where r_1 and r_2 are stored at sites s_1 and s_2, respectively. Let the schemas of r_1 and r_2 be R_1 and R_2. Suppose that we wish to obtain the result at s_1. If there are many tuples of r_2 that do not join with any tuple of r_1, then shipping r_2 to s_1 entails shipping tuples that fail to contribute to the result. It is desirable to remove such tuples before shipping data to s_1, particularly if network costs are high. Consequently, we first project from r_1 all tuples on attributes that occur in both R_1 and R_2, and then we ship these tuples to s_2. At s_2, we join these tuples with relation r_2. We ship the resulting relation back to s_1. Finally, at s_1, we join the received relation with r_1. The resulting relation is exactly the same as the join of relations r_1 and r_2.

This approach is called a *semijoin* execution of a join operation. A semijoin approach is particularly advantageous when relatively few tuples of r_2 contribute to the join. For joins of several relations, this strategy can be extended to form a series of semijoin steps.

Parallel Join

Another alternative is to perform parts of the join in parallel on multiple sites, and then to combine the results to get the complete join. The parallel hash join is one way to do so. In the hash-join algorithm, a hash function h is used to partition tuples of both relations r_1 and r_2. When applied to an attribute of a tuple t, the hash function h returns a value i between 1 and $N - 1$, where N sites participate in the join. When applied to the join attribute of a natural join, the following result holds: Suppose that an r_1 tuple and an r_2 tuple satisfy the join condition; then, they will have the same value for the join attribute.

The basic idea is to partition the tuples of each of the relations amongst the sites, such that site i receives all tuples of r_1 and r_2 whose join attributes have a hash value i. Note that an r_1 tuple at site i and an r_2 tuple at a different site j cannot possibly satisfy the join condition. Each site then independently, and in parallel with other sites, computes the join of its partition of r_1 and r_2. The results at each site are shipped to the user site, and their concatenation gives the final join result.

DISTRIBUTED TRANSACTION PROCESSING

Access to the various data objects in a distributed system is usually accomplished through transactions, which must preserve the ACID properties. There are two types of transactions in a distributed database. *Local transactions* are those that access and update data at only one local site: the site where the transaction starts. *Global transactions* are those that access and update data at several local sites. Ensuring the ACID properties of local transactions is usually done by a local DBMS. In the case of global transactions, however, this task is much more complicated, because several sites may be participating in execution. The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

Transaction Management Model

Each site has its own *local transaction manager* whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions. We define a model of a transaction system as follows. At each local site there are two subsystems:

- The *transaction manager* coordinates the execution of the various transactions (both local and global) initiated at that site.
- The *data manager* manages the execution of those transactions (or subtransactions) that access data stored in a local site. Note that each such transaction may be either a local transaction or part of a global transaction that accesses data at the local site.

The overall system architecture is depicted in Fig. 5.

The transaction manager at site s_i coordinates execution of all transactions at that site. Each operation of a transaction

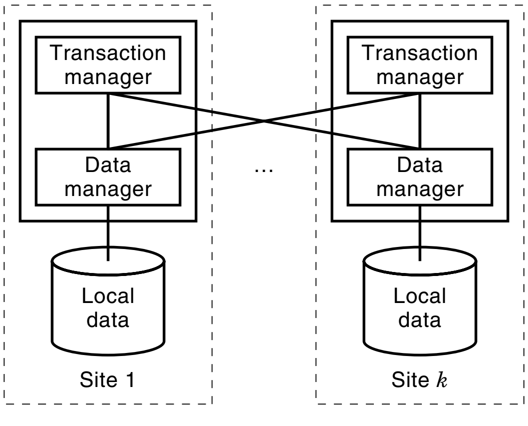

Figure 5. Transaction management model.

is submitted to the site transaction manager. The transaction manager decides at which site the operation should be executed, and it ships the operation to that site. If an operation is to be executed at the local site, the transaction manager decides whether the operation must be submitted to the data manager for execution or must wait, or whether the transaction submitting the operation must be aborted. The latter could occur if the transaction manager concluded that execution of the transaction might violate the transaction ACID properties. If a transaction T_k submits its first operation at site s_i, then the transaction manager at site s_i becomes the T_k's transaction coordinator. That is, site s_i is responsible for the coordination of T_i execution at all sites. Transaction termination should be conducted such that the transaction coordinator guarantees the transaction atomicity. That is, the database must reflect either all or no data changes made by the transaction. Transaction termination usually employs an atomic commit protocol, such as the two- or three-phase commit protocols that we discuss in the sections entitled "The Two-Phase Commit Protocol" and "Three-Phase Commit Protocol."

The data manager is responsible for logging all operations that any transaction performs at the local site. The system uses this information to restore a database to a consistent state in the event of a failure during transaction execution.

System Failure Modes

There are two basic types of failure in a distributed environment:

- *Site Failure.* Site failures occur when a site becomes nonoperational and all useful processing stops. The failure may occur at the site operating system or at the local DBMS. In most distributed database systems, each local site is considered to be in one of two modes: operational or nonoperational. Even if a site responds to some messages, if it does not respond to all messages, then it is considered to be nonoperational and thus to have failed.
- *Communication Failure.* Communication failure occurs when a message sent from site s_1 to site s_2 does not reach the destination site. Loss or corruption of individual messages is always possible in a distributed system. The system uses *transmission-control protocols,* such as TCP/IP, to handle such errors. Even if a link between two sites is down, the network may be able to find an alternative route to deliver the message, making the failure invisible to the distributed database system. If, however, due to link failure, there is no route between two sites, the sites will be unable to communicate.

If there are two sites in the network that cannot communicate at all, a *network partition* has occurred. Network partitions are the source of many different problems that degrade the performance of distributed database systems. It is generally not possible, however, to differentiate clearly between a site failure and communication failures that lead to network partitions. The system can usually detect that a failure has occurred, but it may not be able to identify the type of failure. For example, suppose that site s_1 is not able to communicate with s_2. Perhaps s_2 has failed, or perhaps the link between s_1 and s_2 has failed, resulting in a network partition.

When a network partition occurs and a transaction needs a datum located in another partition, the transaction may have to be aborted or to wait until the communication is restored. An abort of such a transaction is the preferable resolution, because otherwise the transaction may hold resources for undetermined period, potentially impeding other transactions in a partition that is operational. However, in some cases, when data objects are replicated it may be possible to proceed with reads and updates even though some replicas are inaccessible. In this case, when a failed site recovers, if it had replicas of any data object, it must obtain the current values of these data objects and must ensure that it receives all future updates. We address this issue in the section entitled "Replication of Data."

DISTRIBUTED CONCURRENCY CONTROL

The notion of transaction isolation used in distributed systems remains the same as in centralized systems, namely *serializability.* A concurrent execution of a set of transactions is said to be serializable if the effect of the execution (in terms of the values seen by the transactions, and the final state of the database) is the same as that of some serial execution of the same set of transactions.

To ensure the transaction isolation property, distributed database systems typically use a distributed version of the well-known concurrency-control protocols for centralized DBMSs.

Distributed Two-Phase-Locking Protocol

In a centralized version of the two-phase-locking protocol, the transaction manager keeps two types of locks for each data object: a *read lock* and a *write lock.* Each transaction T_i, before it can perform a read (or write) operation on data object a, must request a *read* (or *write*) lock on a. T_i receives a *read* (or *write*) lock if no other transaction keeps a *write* (or a *read* or *write*) lock on a. If a lock cannot be granted, T_i either waits or is aborted. When the transaction does not need an acquired lock, it can release the lock.

A transaction acquires locks following the two-phase-locking rule: *No lock can be granted to a transaction after that transaction has released at least one of its locks.* If each transaction follows the two-phase-locking rule, then the local DBMS ensures the isolation property. A simple way of ensuring the two-phase-locking rule is to hold all locks until the end of the transaction.

The two-phase-locking protocol is prone to deadlocks. For example, suppose that user A at site s_1 wants to transfer $200 from account acc_1 to account acc_2 that is located at site s_2. At the same time, user B wants to transfer $300 from account acc_2 at site s_2 to account acc_1 at site s_1. After A has acquired a write lock on acc_1 at s_1 and B has acquired a write lock on acc_2 at site s_2, A would have to wait for B at s_2 to get a write lock for acc_2, and B would have to wait for A at s_1 to get a write lock on acc_1. Neither A nor B can release the lock it already has due to the two-phase-locking rule. Thus, a deadlock ensues. Observe that at each site the local DBMS is not able to unilaterally determine that there is a deadlock between the A and B lock requests. Deadlock detection in distributed databases needs to be performed in a global setting. We consider the deadlock detection in the context of different lock manager implementations in the section entitled "Lock Manager Implementation."

Recall that a data object in a distributed database may have multiple replicas; all the replicas must have the same value. A simple way of ensuring that all replicas have the same value at the end of a transaction is to require the transaction to write the value to all replicas. (We consider atomic transaction commit in the section entitled "Distributed Commit Protocols.") When a transaction needs to read the data object, it can then read any replica of the data object. We shall assume for now that this simple *read-one, write-all* protocol is followed. The drawback of this protocol is that when a site that holds a replica of an item has failed, it is not possible for any transaction to write to that item. Ways of permitting writes to occur on only replicas that are located at live sites are considered in the section entitled "Replication of Data."

Lock Manager Implementation

There are several possible approaches to implement lock managers in a distributed databases. We study two approaches in this section. We consider other approaches that deal better with replicated objects in the section entitled "Replication of Data."

Centralized Lock Manager Approach. In the centralized lock manager approach, the system maintains a *single* lock manager that resides in a *single* chosen site—say, s_i. All lock and unlock requests are made at site s_i. When a transaction needs to lock a data object, it sends a lock request to s_i. The lock manager determines whether the lock can be granted immediately. If the lock can be granted, the lock manager sends a message to that effect to the site at which the lock request was initiated. Otherwise, the request is delayed until it can be granted and the message sent. The transaction can read the data object from *any* one of the sites at which a replica of that data object resides. In the case of a write, all the sites where a replica of the data object resides must be involved in the writing.

The centralized lock manager scheme has the following advantages:

- *Simple Implementation.* This scheme requires only two messages for handling lock requests, and only one message for handling unlock requests.
- *Simple Deadlock Handling.* Because all lock and unlock requests are made at one site, the deadlock-handling algorithms are identical to deadlock-handling schemes in a centralized database system.

The disadvantages of the centralized lock manager scheme include the following:

- *Bottleneck.* Site s_i becomes a bottleneck, because all requests must be processed there.
- *Vulnerability.* If the site s_i fails, the concurrency controller is lost. Either processing must stop, or a recovery scheme must be used so that a new site can take over lock management from s_i.

Distributed Lock Manager Approach. In this approach, different sites are responsible for handling locking for different data objects. In case data objects are not replicated, the site where the data object resides is responsible for handling locking of that data object. Requests for locks are sent to that site; and as in the centralized lock manager approach, the lock manager at the site responds appropriately to the request. In the case of data replication, we can choose one of the replicas as the *primary copy.* Thus, for each data object a, the primary copy of a must reside in precisely one site, which we call the *primary site of a.* For uniformity, for nonreplicated data objects we will consider the site where the object resides as the primary site of the object.

When a transaction needs to lock data object a, it requests a lock at the primary site of a. As before, the response to the request is delayed until the request can be granted. Thus, the primary copy enables concurrency control for replicated data to be handled in a manner similar to the case of nonreplicated data. This similarity allows for a simple implementation.

Deadlock detection is more complicated in this case since the information at each of the local lock managers may not be sufficient to detect a deadlock. Distributed database systems can perform global deadlock detection by collecting information from each of the local lock managers at a single site. More complicated distributed algorithms for deadlock detection have also been proposed. All these algorithms require cooperation from the different local databases. Many distributed database systems handle deadlocks by using *timeouts.* That is, after the transaction waits for a certain prespecified time (the timeout interval) if a transaction has not received a requested lock, it is aborted. Although this approach may lead to unnecessary aborts, it has the advantage of not requiring any special actions for global deadlock detection.

Multidatabase Concurrency Control

Ensuring the local autonomy of each DBMS requires making no changes to the local DBMS software. A DBMS at one site thus is not able to communicate directly with a DBMS at any other site to synchronize the execution of a global transaction that is active at several sites.

Since the multidatabase system has no control over the execution of local transactions, each local DBMS must use a concurrency-control scheme (for example, two-phase locking or timestamping) to ensure that its schedule is serializable. In addition, in case of locking, the local DBMS must be able to guard against the possibility of local deadlocks.

The guarantee of local serializability is not sufficient to ensure global serializability. As an illustration, consider two global transactions T_1 and T_2, each of which accesses and updates two data objects, A and B, located at sites s_1 and s_2, respectively. Suppose that the local schedules are serializable. It is still possible to have a situation where, at site s_1, T_2 follows T_1, whereas, at s_2, T_1 follows T_2, resulting in a nonserializable global schedule. Indeed, even if there is no concurrency among global transactions (that is, a global transaction is submitted only after the previous one commits or aborts), local serializability is not sufficient to ensure global serializability.

To guarantee global serializability, the execution monitor in a multidatabase system must take some actions. Which actions it takes depend on the degree of cooperation among local DBMSs. If local DBMSs do not cooperate at all and the execution monitor is not aware of any details of how the local

DBMS schedules local operations, then a scheme based on the idea of a *ticket* works. In the ticket approach, a special data object called *ticket* is created at each local site. Each global transaction that accesses the data at the local site must write a ticket at that site first. Consequently, any two global transactions that update data at the same local site directly conflict at that site. Since every local DBMS generates a locally serializable schedule, the global transaction manager—by controlling the order of global transactions accessing local tickets—guarantees global serializability.

If the execution monitor knows that at each local site any two transactions executed in serial order are also serialized in the order of their execution, then a scheme based on the idea of *site graphs* can be used. In the site graph approach, the execution monitor maintains an undirected bipartite graph. Global transaction T is connected to site s, if T performs any operations at s. The execution monitor can guarantee global serializability by ensuring that the site graph is always acyclic; it can ensure acyclicity by controlling the access of global transactions to sites. In general, the more information available to the execution monitor about local DBMSs, the easier it is to implement isolation of global transactions.

DISTRIBUTED COMMIT PROTOCOLS

In order to ensure atomicity, all the sites at which transaction T executed must agree on the final outcome of the execution; T must either commit at all sites or abort at all sites. To ensure this property, the execution monitor must execute a *commit protocol.* In the case of DDB, a transaction manager at the site where the transaction is initiated becomes the *transaction coordinator* that monitors the transaction-commit protocol. In the case of a multidatabase system, the execution monitor acts as the transaction coordinator. To implement the commit protocol, the sites must give up some autonomy; specifically, they cannot make the commit/abort decision for a global transaction by themselves. Instead, they need to cooperate with other sites and the transaction coordinator to make the decision.

Among the simplest and most widely used commit protocols is the *two-phase commit* (2PC) protocol. The alternative *three-phase commit* (3PC) protocol avoids certain disadvantages of the 2PC protocol but adds complexity and overhead.

The Two-Phase Commit Protocol

Let T be a transaction initiated at site s_i. That is, the transaction manager at s_i is the transaction coordinator for T. The transaction managers at all sites at which T was active are called *participants.*

After T completes execution, the coordinator records in persistent storage that it is starting a commit process, and it sends to each participant a *prepare-to-commit* message. After a participant has received a *prepare-to-commit* message, it checks whether T has performed all its operations successfully at its site, and whether it is ready to commit T at its site. The participant both records its decision in persistent storage and sends its decision to the coordinator. If the participant's decision is not to commit T, then it aborts T. If the participant has voted to commit T, it cannot unilaterally change its vote until it hears again from the coordinator. In such a case, the participant continues to keep all resources that are allocated to T, so other transactions that need any of these resources have to wait until the participant releases the T's resources. The voting process constitutes the first phase of the commit protocol.

After the coordinator receives votes from all participants, or if at least one of the participants fails to respond within an allotted time (which, from the coordinator's viewpoint, is equivalent to voting *no*), the coordinator makes the decision whether to commit T. If all the participants voted to commit T, the coordinator persistently records that T is committed and then sends a *commit* message to each of the participants. If at least one of the participants voted against committing T or did not respond, the coordinator persistently records the decision to abort T and then sends the *abort* message to all the participants. Each of the participants that has voted to commit T waits for the message from the coordinator on whether to commit or abort T.

Since unanimity is required to commit a transaction, the fate of T is sealed as soon as at least one site votes not to commit T. Since the coordinator site s_i is one of the sites at which T executed, the coordinator can decide unilaterally to abort T. The final verdict regarding T is determined at the time that the coordinator writes that verdict (commit or abort) to persistent storage.

We now examine in detail how the 2PC protocol responds to various types of failures. When the participant site s_i recovers, it first finds the state of the protocol for T from its persistent storage, and based on the state it does one of the following. If s_i had failed before it had voted, then it aborts T. If s_i had failed after it had voted to commit T but before it had received the commit/abort verdict from the coordinator, then we say that s_i failed in an *uncertain state,* since s_i does not know what has been decided about T. In this case, s_i requests a verdict about the transaction from the coordinator. If it receives the verdict, it proceeds to commit or abort T as per the verdict. If the coordinator is not available (either it has failed or is unreachable), s_i checks the status of the transaction from other participants. If any of them indicates that the transaction committed or aborted, s_i performs the same action on T. If all other live participants are in the uncertain state, s_i must wait for the coordinator to recover to find out the status of T. In the meantime, the participant must keep reserved the resources (such as locks) allocated for T. Such a situation is called *blocking.* The weakest aspect of 2PC is that the protocol is subject to blocking.

If the coordinator fails before it sends a *prepare-to-commit* message, then, after it recovers, it aborts the transaction. Observe that every participant has already aborted T while waiting for and failing to receive the *prepare-to-commit* message. If the coordinator fails before it collects all the votes or before it has sent its decision to all participants, then, after it recovers, it aborts T and sends the *abort* message to all participants. Observe that while the coordinator remains nonoperational, each participant that has voted to commit T is blocked, since it does not know the coordinator's decision.

When a network partitions, two possibilities exist:

1. The coordinator and all its participants remain in one partition. In this case, the failure has no effect on the commit protocol.
2. The coordinator and its participants belong to several partitions. From the viewpoint of the sites in one of the

partitions, it appears that the sites in other partitions have failed. Sites that are not in the partition containing the coordinator simply execute the protocol to deal with failure of the coordinator. The coordinator and the sites that are in the same partition as the coordinator follow the usual commit protocol, assuming that the sites in the other partitions have failed.

Thus, the major disadvantage of the 2PC protocol is that coordinator failure may result in blocking, where a decision either to commit or to abort T may have to be postponed until the coordinator recovers.

The two-phase commit protocol is widely used in the industry. The X/Open XA standard defines a set of functions for supporting the 2PC protocol. Any database that supports the standard can participate in a 2PC with any other databases that support the standard.

Three-Phase Commit Protocol

The *three-phase commit* (3PC) protocol is an extension of the two-phase commit protocol that avoids the blocking problem under certain assumptions. In particular, it is assumed that no network partition occurs and that not more than k sites fail, where k is some predetermined number k. Under these assumptions, the protocol avoids blocking by introducing an extra third phase where multiple sites are involved in the decision to commit. Instead of directly noting the commit decision in its persistent storage, the coordinator first ensures that at least k other sites know that it intended to commit the transaction. If the coordinator fails, the remaining sites first select a new coordinator. This new coordinator checks the status of the protocol from the remaining sites; if the coordinator had decided to commit, at least one of the other k sites that it informed will be up and will ensure that the commit decision is respected. The new coordinator restarts the third phase of the protocol if some site knew that the old coordinator intended to commit the transaction. otherwise the new coordinator aborts the transaction.

Although the 3PC protocol has the desirable property that it does not cause blocking, it has the drawback that a network partitioning will appear to be the same as more than k sites failing, violating the assumptions made earlier. Thus the 3PC protocol is after all subject to some degree of blocking, and given its significantly greater cost, it is not widely used.

Coordinator Selection

Several of the algorithms that we have presented require a process at a site to coordinate the activities of other sites. The coordinator in 2PC is an example. Other examples include, in a centralized lock manager, the site that has the lock manager, or, with a distributed lock manager, the site that performs deadlock detection. We refer to such processes as *coordinators.*

If the coordinator fails because of a failure of the site at which it resides, the system can continue execution only by starting a new coordinator on another site. One way to continue execution is to maintain a backup coordinator that is ready to assume the coordinator's responsibility. A *backup coordinator* is a site that, in addition to other tasks, maintains enough information locally to allow it to assume the role of coordinator with minimal disruption to the distributed-database system. All messages directed to the coordinator are received by both the coordinator and its backup. The backup coordinator executes the same algorithms and maintains the same internal state information (such as, for a concurrency coordinator, the lock table) as does the coordinator. The only difference in function between the coordinator and its backup is that the backup does not take any action that affects other sites.

In the event that the backup coordinator detects the failure of the coordinator, it assumes the role of coordinator. Since the backup has all the information available to it that the failed coordinator had, processing continues without interruption.

The primary advantage to the backup approach is the ability to continue processing without delay if the coordinator fails. If a backup were not ready to assume the coordinator's responsibility, a newly appointed coordinator would have to seek information from all sites in the system so that it could execute the coordination tasks. Frequently, the only source of some of the required information is the failed coordinator. In that case, it may be necessary to abort several (or all) active transactions and to restart them under the control of the new coordinator.

Thus, use of a backup coordinator avoids a substantial delay for recovery from coordinator failure. The disadvantage is the overhead of duplicate execution of the coordinator's tasks. Furthermore, a coordinator and its backup may need to communicate regularly to verify that their activities are synchronized.

In the absence of a designated backup, or in order to handle multiple failures, a new coordinator can be chosen dynamically by sites that are live. *Election algorithms* have been designed for the purpose of enabling the sites to make this decision collectively, in a decentralized manner. In the *bully algorithm,* sites have identifiers preassigned, and the site among the live ones that has the highest numbered identifier is chosen.

REPLICATION OF DATA

A major goal of replication is to create the possibility of a distributed database continuing to process transactions even when some sites are down. So far our protocols for dealing with replication have assumed that all replicas of a data object must be updated for the transaction to commit; recall the *read-one, write-all* policy for handling replicated data from the section entitled "Distributed Concurrency Control." In a distributed database system that comprises hundreds of data sites, there is a high likelihood that at least one site is not operational. If that site contains a replica of the data that needs to be written, the transaction must either abort or wait until the site recovers, neither of which is acceptable.

In this section we consider protocols that enable transactions to update just those replicas that are available. These protocols define *when* and *how* to continue operations on the available replicas, as well as how to reintegrate a site that was not available earlier, when it comes back. Reintegration of a site is more complicated than it may seem to be at first glance, because updates to the data objects may have been processed while the site is recovering. An easy solution is temporarily to halt the entire system while the failed site rejoins it. In most applications, however, such a temporary halt is unacceptably disruptive. Techniques have been developed that allow failed sites to reintegrate while allowing concurrent updates to data objects.

Enforcing global serializability is also an issue in these schemes. A centralized lock manager or a primary copy locking scheme is not acceptable since the failure of one site can prevent processing from continuing in other sites. Alternative locking schemes are therefore used. Each site maintains a local lock manager whose function is to administer the lock and unlock requests for those data objects and replicas that are stored in its site. When the local lock manager receives a lock request for a replica at that site, it determines whether, as far as it is concerned, the lock can be granted. If it can, a reply granting the lock is sent immediately; if not, the response is delayed until the request can be granted. Global deadlock detection is of course a problem; we assume that either timeouts are used or there is a coordinator that periodically performs global deadlock detection.

Ensuring atomicity of commit remains an issue as before, and it can be handled by the usual two-phase commit protocol.

Majority-Based Approach

In this approach, a version number is stored with each data object to detect when it was last written to. Whenever a transaction writes an object, it also updates the version number as we describe here.

If data object a is replicated in n different sites, then a lock-request mesage must be sent to more than one-half of the n sites in which a is stored. The transaction does not operate on a until it has successfully obtained a lock on a majority of the replicas of a.

Read operations look at all replicas on which a lock has been obtained, and they read the value from the replica that has the highest version number. (Optionally, they may also write this value back to replicas with lower version numbers.) Writes read all the replicas just like reads to find the highest version number (this step would normally have been performed earlier in the transaction by a read, and the result can be reused). The new version number is one more than the highest version number. The write operation writes all the replicas on which it has obtained locks, and it sets the version number at all the replicas to the new version number.

Failures during a transaction can be tolerated as long as the sites available at commit time contain a majority of replicas of all the objects written to; and during reads, a majority of replicas are read to find the version numbers. If these requirements are violated, the transaction must be aborted.

In this approach, reintegration is trivial; nothing needs to be done. The reason is that the writes will have updated a majority of the replicas, whereas the reads will read a majority of the replicas and find at least one replica that has the latest version.

Quorum Consensus Approach

The *quorum consensus* (QC) approach is a generalization of the majority protocol. In this scheme, each site is assigned a nonnegative weight. Read and write operations on an item x are assigned two integers, called *read quorum* Q_r and *write quorum* Q_w, that must satisfy the following condition, where S is the total weight of all sites at which x resides:

$$Q_r + Q_s > S \quad \text{and} \quad 2 * Q_s > S$$

To execute a read operation, enough replicas must be read such that their total weight is more than or equal to Q_r. To execute a write operation, enough replicas must be written to such that their total weight is more than or equal to Q_w. The arguments of correctness for the majority approach can be readily generalized for the quorum consensus approach.

The benefit of the QC approach is that it can permit the cost of either reads or writes to be selectively reduced by appropriately defining the read and write quorums. For instance, with a small read quorum, reads need to read fewer replicas, but the write quorum will be higher; hence writes can succeed only if correspondingly more replicas are available. Also, by giving higher weights to some sites (e.g., those less likely to fail), fewer sites need to be accessed by either writes or reads. However, the danger of failures preventing the system from processing transactions increases if some sites are given higher weights.

Read One, Write All Available Approach

We now consider the *read one, write all available* approach. In this approach, the read operation is done the same way as in the *read one, write all* scheme; any available replica can be read. A read lock is obtained at that replica. The write operation is shipped to all replicas; write locks are acquried at all the replicas. If a site is down, the transaction manager proceeds without waiting for the site to recover.

Although this approach appears attractive, there are several complications. In particular, temporary communications failure may cause a site to appear to be unavailable, resulting in a write not being performed; when the link is restored, however, the site is not aware that it has to perform some reintegration actions to catch up on writes that it has lost. Furthermore, if the network partitions, each partition may proceed to update the same data item, believing that sites in the other partitions are all dead.

All the read one, write all available schemes we are aware of either assume that there is never any communication failure, or are very expensive in the presence of failures, and are therefore not very practical.

ALTERNATIVE MODELS OF DISTRIBUTED TRANSACTION PROCESSING

For many applications, the blocking problem of two-phase commit is not acceptable. The problem here is the notion of a single transaction that works across multiple sites. In this section we consider alternatives that can avoid the blocking problem in many cases. We first consider *persistent messaging* and then we look at the larger issue of *workflows.*

Persistent Messaging

To understand persistent messaging, we consider how one might transfer funds between two different banks, each with their own computer. One approach is to have a transaction span the two sites and to use the two-phase commit protocol to ensure atomicity. However, the transaction may have to update the total bank balance, and blocking could have a serious effect on all other transactions at each bank, since almost all transactions at the bank would update the total bank balance.

In contrast, consider how funds transfer occurs when a banker's check is used. The bank first deducts the amount of the check from the available balance and then prints out a check. The check is then physically transferred to the other bank where it is deposited. After verifying the check, the bank increases the local balance by the amount of the check. The check constitutes a message sent between the two banks. So that funds are not lost or incorrectly increased, the check must not be lost, and it must not be duplicated and deposited more than once.

When the bank computers are connected by a network, *persistent messages* provide the same service as the check (but do so much faster, of course). Unlike regular messages, persistent messages give the guarantee that once they are generated, they will definitely be delivered and will never be multiply delivered. Database recovery techniques are used to implement persistent messaging on top of the normal network channels which do not provide delivery guarantees.

Unlike the two-phase commit implementation, with persistent messaging, there must be a code available to deal with exception conditions. For instance, if the deposit account has been closed the check must be sent back to the originating account and must be credited back there. An error handling code must therefore be provided along with the code to handle the persistent messages. In contrast, with two-phase commit the error would be detected by the transaction, which would then never deduct the amount in the first place.

In balance, there are many applications where the benefit of eliminating blocking is well worth the extra work to implement systems using persistent messages.

Workflows

A *workflow* is an activity involving the coordinated execution of multiple tasks performed by different processing entities. A *task* defines some work to be done and can be specified in a number of ways, including a textual description, a form, a message, or a computer program. A *processing entity* that performs the tasks may be a person or a software system.

Consider the processing of a loan; the relevant workflow is shown in Fig. 6. The person who wants a loan fills out a form, which is then checked by a loan officer. An employee who processes loan applications verifies the data in the form using sources such as credit-reference bureaus. When all the required information has been collected, the loan officer may decide to approve the loan; that decision may then have to be approved by one or more superior officers, after which the loan can be made. Each human here performs a task; in a bank that has not automated the task of loan processing, the coordination of the tasks is typically carried out via passing of the loan application, with attached notes and other information, from one employee to the next. Other examples of workflows include processing of expense vouchers, of purchase orders, and of credit-card transactions.

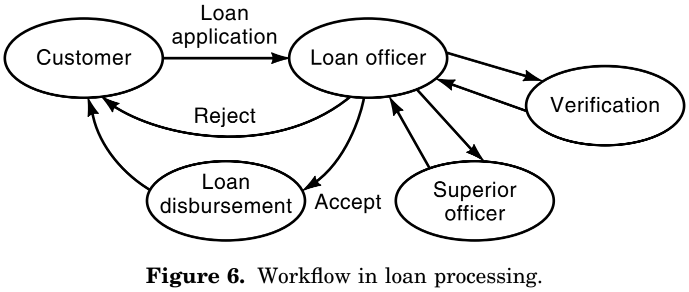

Figure 6. Workflow in loan processing.

Workflows offer an attractive way of implementing a complex long duration task that must span multiple sites in a distributed database. For instance, it may be possible to break up a distributed transaction into a workflow. Some parts of the workflow can execute even when some sites in the distributed database are not available.

Persistent messages provide a mechanism for implementing workflow systems. In a workflow, a single complex task has subtasks that must be executed at different sites. Tasks must be dispatched from one site to another in a reliable fashion. Unlike in normal transaction processing, the tasks in a workflow may take a long time to complete; and even if the database systems involved crash in-between, the workflow must be completed. Persistent messages provide a way to dispatch the tasks reliably. The message requesting a task to be performed is deleted only when the task is completed. If a crash occurs in-between, the message will still be available in a persistent message queue, and the task can be restarted on recovery.

CONCLUSIONS

Although distributed database systems have been a topic of interest since the late 1970s, there is renewed interest in the area due to he growth of corporate Intranets and the Internet, which have enabled hitherto disconnected databases to communicate easily with one another. We can expect distributed databases to form an integral part of most database applications in the future.

We have provided an overview of several aspects of distributed databases, including the architecture of distributed databases, query processing and schema integration, transaction processing including concurrency control and distributed commit protocols, and replication. We refer the interested reader to the sources listed below for further reading.

BIBLIOGRAPHY

P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Reading, MA: Addison-Wesley, 1987. A classic book on concurrency control and recovery with extensive coverage of distributed databases.

Y. Breitbart, H. Garcia-Molina, and A. Silberschatz, Overview of multidatabase transaction management, *VLDB J.,* **1**: 2, 1992. A comprehensive review of various multidatabase transaction processing schemes.

J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques,* San Mateo, CA: Morgan Kaufman, 1993. The bible on the subject of implementation of transaction processing; includes some material on recovery and concurrency in distributed databases.

T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1991. An advanced textbook on distributed databases.

A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts,* 3rd ed., New York: McGraw-Hill, 1997. A fundamental textbook on databases; includes a chapter on distributed databases and also includes material on workflows.

Y. Breitbart
H. F. Korth
A. Silberschatz
Bell Laboratories

S. Sudarshan
Indian Institute of Technology