1985

Analysis of current and future trends in synchronization, deadlock, and recovery in distributed databases

Carmen Ida Joglar

Rochester Institute of Technology
School of Computer Science and Technology

Analysis of Current and Future Trends in Synchronization, Deadlock, and Recovery in Distributed Databases

A Thesis submitted in partial fulfillment of
Master of Science in Computer Science Degree Program

By: Carmen Ida Joglar

Approved By:

Jeffrey Lasky
Professor Jeffrey Lasky, Advisor

Chris Comte
Professor Chris Comte

Rayno Niemi
Professor Rayno Niemi

Date: March 26, 1985

I Carmen Ida Joglar prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

Date: March 26, 1985

Table of Contents

Analysis of Current and Future Trends in Synchronization, Deadlock, and Recovery in Distributed Databases

1. Basic Concepts in Distributed Database Systems

1.1. Intent of the Research

In recent years, distributed database technology has become an area of significant computer science research. Implementations now being developed are bringing to light problems for which no specific solutions are yet available. A study of actual distributed database system implementations will help to provide an understanding of the characteristics and inherent problems of such systems.

The intent of this research is: (1) to discuss the core issues related to synchronization, deadlock and recovery in distributed databases, (2) to assess the known strategies for meeting the processing requirements imposed by these issues, (3) to examine systems now being developed in order to compare operating effectiveness with processing requirements, (4) to identify issues which will warrant future investigation. SDD-1, System R*, and Distributed INGRES will provide the empirical data for this

study.

The System for Distributed Databases-1 (SDD-1) is a distributed database system implemented by the Computer Corporation of America [BERN80d]. It represents an interesting subject of study because it was the first distributed database to be developed. It consists of a collection of up to several hundred geographically dispersed sites connected by the ARPANET network. The system uses a relational model and supports replicated data. A distinguishing characteristic of this system is that it does a preanalysis of transactions to determine if synchronization is needed. A disadvantage of the system is that it is not deadlock free. Recovery is handled by a facility called the Reliable Network (RELNET).

System R* is a distributed database developed by the IBM Corporation [HAAS82]. It represents a good subject of comparison because, unlike the other two systems to be discussed, System R* does not support data replication. It consists of several cooperating autonomous sites, each of which supports a relational database system. Concurrency control is provided by a 2-phase-locking mechanism. Datagram protocols are used handling deadlock and transaction recovery.

Distributed INGRES was developed at the University of California at Berkeley [NEUH77]. It is a relational distributed database system designed to operate on both ETHERNET and ARPANET networks. It supports data replication. One of its outstanding characteristics is that each object has a primary site to which all updates are first directed. Distributed INGRES uses 2-phase-locking for concurrency control. Deadlocks are handled by one machine called the SNOOP. Crash recovery is handled by two sets of algorithms: the performance algorithms and the reliable algorithms.

1.2. Motivation for Distributed Databases

A distributed database implies that data is stored at different locations of a distributed computer system and either data elements are interrelated or there may be a need to access data at one location from another location, or both.

A database is a centralized collection of data. Databases were developed to fill the need to integrate the files of an organization in order to allow better use of the data. Then, there was a need for integrating several existing databases into a single coherent database which could be made available to each of the physically isolated

units through a computer network system. Also, there was a need for decomposing very large databases into a network of geographically dispersed units. The concept of a distributed database evolved from these needs.

The objectives of distributing the data are: 1) to allocate data to the node most frequently accessing it in order to minimize response time and communication costs, 2) to achieve a high degree of data availability by protecting the data against events such as a centralized failure, 3) to increase the storage capacity to that available in the network as a whole.

1.3. Architecture of a System Supporting a Distributed Database

A distributed system can be described as a set of independent but cooperating centralized systems. It consists of a collection of sites, also called nodes, connected through communication links to form a communication network. In a distributed database system, the data is distributed among the different nodes. During normal operation, nodes are connected either through direct communication links or through one or more intermediate nodes.

In a distributed database system, each node consists of a central processing unit, a local database, and a group of online terminals. Each node has a high degree of autonomy. Since the user doesn't know how the data is distributed throughout the system, each node contains the following functional components as a mean of supporting program access to data: a communication port, a network data directory, and a network data management facility. These components are described in the following paragraphs.

To access a unit of data, the node where the request originated must determine if the data is locally available. If the data is not locally available, this node must determine which of the other nodes contains the requested data and must communicate with that node. A communications port is the component in charge of handling communications between the nodes. It contains the physical location of the nodes and the routing information.

A network data directory contains the information that relates the different units of data to the nodes on which they reside.

A network data management facility contains three modules: the remote access control, the local access control, and the redundant file maintenance control. It is

in these modules that many security and synchronization issues are resolved. It should be noticed that these modules can be called differently in different systems. However, in one way or another, all distributed systems provide these functions.

The remote access control module is responsible for detecting and processing all remote access requests originating in the node. It uses the network data directory to locate the node containing the unit of data requested and, if the node is available, transmits the request.

The local access control module is responsible for processing all remote requests received from other nodes. It transmits the data if the data is accessible.

The third module of the network data management is the redundant file maintenance control. It is responsible for coordinating all local and remote updates to be performed at the node. It manages the multiple copies of the data on the system. For example, if an update originates at this node, it looks in the network data directory to locate all nodes containing copies of the data to be updated and sends remote updates to all of them. The redundant file maintenance control module also provides translation functions, such as request translation and data reformatting, between nodes that are not compatible.

1.4. Possible Structures of Distributed Databases

There are three basic ways of distributing the data: the centralized approach, the partitioned approach, and the replicated approach. Combinations of these approaches, called hybrid approaches, can also be found.

1.4.1. The Centralized Approach

In the centralized approach, there is a single central copy of the database and at least one remote user capable of accessing the data from another node. Most database management functions reside at the central node. Often these functions include: checking for user authorization, locating the data, and retrieving it. The schema, which is the database component that holds the structural information, the format, and the access criteria for the data elements, also resides at the central node. Since the data is centralized, no network data directory is needed.

In a remote request, the remote node can check the request for validity and completeness. Then the request must be transmitted to the central node where the other database management functions are to be performed. If the request is an update, it must be preceeded by a lock.

However, locking presents no problem in the centralized approach since it is done only at the central node. In an update request only an acknowledgement is sent back to the remote user to notify him that the update has been made. When the request is a retrieval, the formatting of the data should be done at the receiving node to reduce the amount of data that has to be transmitted.

The advantages of the centralized approach are:

1) minimization or elimination of data redundancy

2) relatively straight forward recovery procedures

3) minimization of synchronization problems

4) lower data storage cost.

The disadvantages of the centralized approach are:

1) contention may exist among several processors attempting to access data simultaneously

2) the size of the database is limited by the availability of secondary storage at the central node

3) the response time can be slow if the database is large

4) all processors lose access to data during a central node failure.

This approach seems appropriate for batch jobs that do extensive processing against the database and can be transmitted to the central node for processing.

1.4.2. The Partitioned Approach

In this approach, the database is divided into physically separated units which are distributed across the different nodes based on access requirements. Sometimes this type of distribution results from the need to integrate several existing databases into a single logical entity. In any event, the partitioned approach does not allow overlapping of segments, that is, copies of data items are not allowed. This eliminates the problem of managing copies of the data.

Since the data is distributed among several nodes, this approach will normally contain a network data directory. Also, each node will normally contain a network data management facility through which all requests will be handled. All network data management facilities will use the network data directory to determine the location at which the data is stored. If the data is stored at another node, the request is transmitted for processing. If the request is a retrieval, the data requested is returned. For updates, the acknowledgement that the

update has been completed is sent back. Complex requests may require processing at several nodes; that is, different segments of the request may have to be processed at different nodes. For complex as well as for simple requests, locking can be adequately performed because there is only one copy of the data to be accessed.

In a partitioned database, there is a higher degree of reliability compared to the centralized approach. This is so because even when one or more nodes fail or when the communication system fails, the system can still be partially functional. For example, an active node can still access its local data. This approach also offers a higher degree of availability since a user's request that can be satisfied locally will not be affected by the failure of other nodes and/or the communication system.

The advantages of the partitioned approach are:

1) the cost of storage is not so high because there is no duplication of data.

2) the size of the database is no limited by the storage available at a particular node, but by the storage available on the network as a whole

3) contention will be less than on the centralized approach because the data is more evenly distributed among the different nodes

4) response time and communication costs will be less than for the centralized

approach since there can be increased parallelism and a greater percentage of the accesses are going to be local.

The primary disadvantage of the partitioned approach is that data item requests may require access to more than one node. This can result in greater communication costs than those of the centralized approach.

The partitioned approach seems to be appropriate for systems where specific parts of the database are primarily referenced from a single location, but access to the rest of the database must be available.

1.4.3. The Replicated Approach

In a replicated database all or part of the data is duplicated at several nodes. Each node will normally contain a network data directory and a network data management facility.

Many retrieval requests can be handled locally since each node contains a copy of a part or all of the database. Update requests must be performed at all nodes containing a copy of the data items to be updated, in order to maintain data consistency. For this reason updates may

not be allowed whenever some copies of the database are not available during a failure.

The advantages of the replicated approach are:

1) a high degree of reliability is achieved because several copies of the database exist and may be used to replace a copy that has been destroyed or damaged

2) a high degree of availability is achieved because, even when a node fails, other copies of the data can still be accessed.

3) improved response time since not all requests need to use the communication system (e.g. retrieval-only request) resulting in little or no contention for accessing the database.

The disadvantages of the replicated approach are:

1) high storage cost due to the duplication of the data

2) Database size is limited by the availability of secondary storage at the smallest data node

3) an update must be made on all the copies.

The replicated database seems appropriate for systems where reliability is critical and update inefficiency can be tolerated. The replicated approach provides the most difficult environment for dealing with the synchronization problem which will be explained below.

1.4.4. The Hybrid Approach

In a hybrid approach the database is divided into disjoint subsets in the same way as in the partitioned approach; but at the same time it allows copies to reside at selected nodes. The aim of such an approach is to achieve a certain degree of reliability, while at the same time making efficient use of storage. This is done by duplicating only the data that is really important. It should also be mentioned that the hybrid approach contains most of the advantages and disadvanteges of both the partitioned and the replicated approach.

1.5. Research Issues

The field of distributed databases encompasses many issues still under research. Perhaps one of the most important issues is the need for synchronization in order to maintain data validity and consistency. Synchronization is a technique used to coordinate concurrent database accesses, and to prevent incorrect modifications and lost updates. This problem is much more severe for distributed databases than for centralized databases because data is not under the control of a single computer or node, and because updates can arrive at various nodes and can yield inconsistent results even though local consistency control

is provided. No solution to this problem has yet been identified.

Another problem still under research is that of deadlock. Deadlock occurs when two or more processes cannot run to completion because they are waiting for each other to release the resource they need in order to complete their execution. Integrity considerations are the main cause of deadlock in distributed databases. Even though the same logical procedures to handle deadlock situations will work for both centralized and distributed systems, distribution may lead to performance problems.

Recovery is the process of getting both the data and the system operational after a failure has occurred. In a distributed system, recovery includes two aspects: node failures and communication failures. The technique used in a centralized environment can also be used for local site recovery in a distributed system. The complexity of recovery in a distributed system is due to the number of system components and to the requirement that all components handle the failure in a consistent manner.

2. Analysis of Current Techniques

This chapter discusses the current techniques used for synchronization, deadlock and recovery in distributed databases.

2.1. Synchronization

Centralized, partitioned and replicated databases existed even before distributed computing systems came into existence. Therefore, what is new is not the distribution of data, but the distribution of control. Distribution of control makes it harder to coordinate access to data. Access coordination is important in maintaining the correctness and validity of the data and in seeing that the values of all copies of a data item agree. The property of data correctness and validity is known as data integrity. The property of all copies of a data item being in agreement with each other is known as data consistency. The process of access coordination in order to maintain data integrity and consistency is called synchronization.

Synchronization in a distributed database is more complex than in a centralized one for several reasons. First, users at different nodes have simultaneous access

to the database. Second, there may be more than one copy of a given data element. Third, inter-node communication may be required by the synchronization mechanism.

When the database is centralized or partitioned, the synchronization method only needs to deal with controlling transactions so that no updates are lost and serial access to data objects is achieved. When the database is replicated or hybrid, the problem is extended to that of maintaining consistency of multiple copies of the data objects.

Synchronization mechanisms can be categorized by their degree of centralization and by their degree of consistency. The degree of centralization refers to the extent to which synchronization control is centralized or distributed. It can vary from one node having entire control (totally centralized), to every node having an equivalent share in controlling the data on the other nodes (totally distributed). While centralized synchronization control is simpler than distributed control, it requires more communication resources. At the same time, centralized synchronization control is less reliable since a failure in the controlling node leaves the entire system without synchronization control.

The following sections will discuss the five major synchronization techniques that have been proposed: locks, timestamps, circulating permits, conflict analysis, and reservations.

2.1.1. Locks

A lock prohibits transactions to access a data object when it is currently being used by another transaction. The purpose of a lock is to ensure that a data object is accessed by only one transaction at a time. In order to achieve this purpose, transactions must set locks before accessing unlocked data objects. If the object is already locked, it must not be locked. At end of transaction, objects must be unlocked. In the case where data objects have been modified by a transaction which could not reached completion, the data objects must be restored to their state prior to the transaction.

Locking can be implemented by having a lock bit associated with each lockable entity and having a test and set instruction which tests the bit, loops if it is already set and sets the bit otherwise.

Since a database management system processes all read and write commands, it can generate the lock requests. In

this way, locking is made transparent to the user.

The disadvantages of locking in distributed systems are:

1) the high overhead associated with transmitting locking information over the communication facility whenever a remote data item is accessed

2) possibility of a deadlock.

There are different modes of locking: shared mode and exclusive mode. An object is said to be locked in shared mode when it allows other transactions to acquire a shared lock. On the other hand, an object is said to be locked in exclusive mode when it doesn't allow any other transaction to acquire a lock of any type. Shared locks are used for read actions, while exclusive locks are used for write actions. In this way concurrent execution of read actions is allowed. At the same time, updates are restricted to execute when no other transaction can access the data object to be updated. This is done to ensure the consistency of the database.

There are also different locking levels: object locking, predicate locking and structure locking. Object locking sets locks on each data object a transaction references. It allows for maximum concurrency. However, a large

number of locks have to be set when a transaction references many objects. There is a great amount of overhead in terms of processing and storage requirements for the locks. Predicate locking sets locks at an entity formed by a group of data objects. The disadvantage is the high level of complexity predicate locking can reach if the predicate is formed by a large group of data objects. Structure locking needs the database to be organized into a hierarchy. If a node is locked, then all its children (or descendant nodes) are also locked. It prevents transactions from locking objects which contain locked components. Structure locking requires little overhead, but restricts concurrency. Therefore, the optimum locking level depends on the size of the transactions, i.e., how many data objects the transaction touches.

Numerous locking techniques have been proposed, but among the most widely known are: the lock manager, the primary copy and two-phase locking.

2.1.1.1. The Lock Manager

This locking technique has some degree of centralized control. It is aimed at reducing the number of messages that must be transmitted through the communication facility. It consists of having just one lock manager at one

site to handle locking activities throughout the entire system.

A look at the locking process shows how the lock manager works. First of all a transaction requests its local database manager to lock a data object. Then, if there are any copies of the data object, a message is sent to the lock manager requesting the locking of all the copies. The lock manager checks if the requested lock can be granted. All copies of the data object must be free from locks before a requested lock is granted. A grant or reject message is sent back to the lock request originator. Only after the request originator receives the lock grant message can the action be performed. Once a lock on a data object is granted, the transaction requesting it can access any copy of the data object at whatever site it resides. In case of updates, all copies must be updated before the lock is released. Updates are handled by simultaneously sending the result of the update action to all sites having a copy of the data object and waiting for an acknowledgement. At end of transaction all locked data objects are unlocked.

This technique has two disadvantages:

1) it is unreliable because if the locking site fails, the entire synchronization system fails

2) the locking site tends to be a bottleneck, because its capacity to process locks bounds the capacity of the entire system.

2.1.1.2. The Primary Copy

In primary copy locking, one of the copies of the data item is designated as the primary copy. If a data item has no copies, then it is considered to be the primary copy itself. All updates are directed to the site containing the primary copy of the data object. This site is then responsible for the synchronization of updates to all other copies, if any exist. It is aimed at eliminating the bottleneck problem of the lock manager technique, because primary copies of different data objects can be stored at different sites.

When a transaction requests a lock of a data object, a message is sent to the site containing the primary copy of that data object. If it is already locked, the transaction must wait. Otherwise, the primary copy is locked; the update is performed on the primary copy and then broadcasted to all sites containing a copy of the data object.

The disadvantages of this technique are:

1) if the site containing the primary copy of the data object requested for locking has failed, the primary copy is unavailable and the lock cannot be granted

2) it is difficult to coordinate deadlock detection since all sites containing a copy of the data items being accessed must participate in this process.

2.1.1.3. Two-Phase Locking

Two-phase locking consists of two basic phases that are sometimes called the growing phase and the shrinking phase. During the growing phase a transaction acquires all necessary locks without releasing any lock. If a requested lock cannot be granted, the transaction must wait. This can result in a deadlock situation if the waiting transaction holds a lock on a data item that is needed by another transaction and at the same time that other transaction is holding a lock on a data item needed by the waiting transaction.

During the shrinking phase a transaction releases locks and is prohibited from obtaining additional locks. By releasing locks before the end of transaction, parallelism and performance can increase, because transactions

which have been waiting for a particular lock may proceed without having to wait until the preceeding transaction completes. However, if for any reason the transaction releasing the locks cannot reach completion, all necessary information to undo the actions performed by the transaction must be available. Therefore, although reducing parallelism, it is prefered to keep all locks acquired until the transaction terminates. It is only at commit time (i.e. at the time when changes are permanently recorded in the database, that locks should be released.

For replicated databases, a transaction may read any copy of the data item and only needs to obtain a lock on the copy it will actually read. For write operations, a transaction must update all copies of the data item and must therefore obtain locks on all copies before updating.

A disadvantage of this technique is that it requires the locking of all copies of a data object before processing updates on that data object. If a failure at a site containing one of the copies occurs after the transaction has granted some of the locks, the transaction must wait. This can result in tying data objects which may be needed by other transactions.

2.1.2. Timestamps

A timestamp is a unique number assigned to a transaction or to a data object. In a distributed system, a timestamp usually consists of a concatenation of the local time and local site identifier of the site at which the transaction originated. The site identifier is assigned the least significant position to avoid the possibility that all timestamps generated by one site are greater than all timestamps generated by another site. Clocks or counters can be approximately synchronized. If a site receives a message with a timestamp greater than its current counter, it increments its counter to be the value of the received timestamp plus one. Timestamps specify a total ordering of all the transactions in the system. They are used to synchronize the interleaved execution of a set of transactions.

Conflicts are resolved by restarting transactions. If a transaction is restarted, it is assigned a new timestamp. If a transaction's timestamp has a smaller value than that of the last completed transaction, the transaction is aborted and restarted with a new timestamp until this timestamp's value is greater than the timestamp value of the last completed transaction. Restarting a transaction means aborting it, returning all modified data to its original value, and assigning a new and larger timestamp

value to the restarted transaction.

Since transactions are restarted and no locks are used, no deadlock should occur. Therefore, there are no communication overheads from locking and deadlock detection. On the other hand, there is a high storage cost associated with the timestamping technique since timestamps must be permanently stored with each data object.

2.1.2.1. Basic Timestamping

In basic timestamping, each transaction receives a timestamp when it is initiated. Also, each of the read or write requests that is part of a transaction has the same timestamp as the transaction. Transactions do not issue write requests; they issue prewrites instead. Prewrites are buffered and are not applied directly to the database. At end of transaction, the write requests corresponding to the buffered prewrites are then applied to the database.

Each data object contains the largest timestamp received from a read request and the largest timestamp received from a write request. If a read request has a timestamp value smaller than the data object's largest timestamp of a write request, the transaction is restarted. If there is a pending prewrite operation having a

smaller timestamp value than the read request, the read request is buffered until the transaction which issued the prewrite reaches end of transaction. Otherwise, the read request is executed and its timestamp is recorded as the largest timestamp received from a read request for that data object.

If a write request has a timestamp value smaller than the data object's largest timestamp of a read or a write request, the transaction is restarted. If there is a pending prewrite operation having a smaller timestamp value than the write request, the write request is buffered until the transaction which issued the prewrite reaches end of transaction. That means that the write request is executed after all prewrites with smaller timestamps have applied their corresponding writes to the database.

When using basic timestamping no deadlocks are expected to occur, but this is obtained at the cost of restarting transactions [CERI84].

2.1.2.2. Conservative Timestamping

Conservative timestamping eliminates the need for restarts. When a transaction is received, it is buffered until there is no other transaction having a smaller

timestamp.

A transaction is executed only at its site of origin. This site is then responsible for directing the transaction's read and write requests to other remote sites and seeing that they are received in timestamp order. This can be achieved by having a queue of read requests and a queue of write requests for each site in the system. Queues are to be maintained in timestamp order. A read (or a write) request will be processed when the first entry in each update queue has a timestamp greater than that of the read (or write) request. Timestamped null requests are sent periodically because this technique requires each queue to contain at least one request. Sites can ask for null requests. Asking for null requests provides a mean of detecting failed sites and eliminates unnecessary waiting for them.

The advantages of conservative timestamping over basic timestamping are:

1) conservative timestamping reduces the amount of processing time needed by eliminating restarts

2) conservative timestamping reduces the amount of storage space needed by eliminating timestamps on data objects.

The disadvantage of conservative timestamping is that it restricts concurrency because processing must follow a specific order (the timestamp order).

2.1.2.3. The Majority Consensus Algorithm

The majority consensus algorithm is a synchronization technique based on timestamps and voting. It was proposed by R. H. Thomas as a possible solution to the synchronization of updates to multiple copies [THOM79]. It assumes that data objects are duplicated at every site. It is based on the idea that the correct value of a data item is that value held by the majority of its copies.

Read requests are executed at the site where they originate and without using any synchronization mechanism.

An update (or write) request does not modify the database until a majority of the sites vote on accepting the transaction. Each data object contains its value and the timestamp corresponding to the time at which the current value was assigned. For an update request, the data object and its new value are recorded in an update list that is sent to every site. Since the data object must be read before it is updated, each site checks that the data object has not been modified since it was read.

If it has been modified, the site must vote to reject the request. If there is a conflict with another request that has not yet been accepted or rejected, the site must also vote to reject the request. If a majority of the sites vote to reject the transaction, it is restarted.

The advantage of this algorithm is that it works as long as the majority of the sites are working, that is, it works even if there are some site or communication failures.

The disadvantages are:

1) storage costs can be high because data objects need to store the timestamp value of their last update

2) concurrency is limited by the number of restarted transactions.

2.1.3. Circulating Permits

This synchronization technique visualizes the system as a set of sites forming a virtual communication ring. A control token circulates the ring. In this way, update requests are guaranteed to be performed in serial order, by passing the token from one site to its successor. Only the site having the token can initiate an update request.

When using this technique, an algorithm must be included to provide that there is always only one token on the ring and to regenerate it in case it is lost. The algorithm must also exclude faulty sites from the ring and must insert them again when they are operational.

The disadvantage of this technique is that it allows little or no concurrency. In an effort to increase the amount of concurrency allowed, another version of this technique was created. This other version partitions the database. Each of the partitions at a site can be updated concurrently whenever the site owns the control token. However, concurrency is limited by the number of partitions.

2.1.4. Conflict Analysis

Conflict analysis is a synchronization technique based on an analysis of transactions that may conflict with each other. Its purpose is to eliminate unnecessary delays and to increase concurrency by identifying transactions that need no synchronization. For this analysis, the concept of transaction classes is introduced. A transaction class is a set of transactions defined by the set of data objects to be read or written. The sets of data objects to be read or written are called read-sets and

write-sets, respectively. Concurrent transactions are not in conflict with each other as long as only their read-sets intersect.

Since transactions on the same class are executed serially, only transactions on different classes need to be considered in the conflict graph analysis. Transaction classes should be defined with as small a read-set and write-set as possible in order to minimize conflicts. Transaction classes and conflict analysis are defined when the database is designed. A conflict graph is constructed by representing read-sets and write-sets as nodes and representing conflicts as edges. Transactions that lie on cycles require synchronization.

At run time, each time a transaction is submitted, the transaction class to which it belong must be identified and the synchronization level required for that class must be applied. The strongest synchronization level is that required for transactions that are not known members of any of the predefined transaction classes.

2.1.5. Reservations

This synchronization technique is based on the use of two protocols and a reservation list.

The pessimistic protocol requires transactions to preclaim and reserve data objects before execution. It is aimed at avoiding conflicts and avoiding the need to restart transactions. It requires the transaction's site of origin to assign a timestamp to the transaction as a measure against deadlocks. The data objects in the transaction's read-set and write-set must be reserved. However, if any of the data objects in these sets are already reserved, the reservation request is entered in a reservation list. The reservation list is kept in timestamp order. Then the transaction's site of origin broadcasts the reservation request. If the database is replicated, each site makes reservations of the data objects' copies it holds and sends back an acknowledgement. These reservations are made in the same way they are made at the transaction's site of origin.

If the reservation request was entered in a reservation list, the transaction cannot execute until this request becomes the oldest in the list and the actual reservation is made. Reservation requests are removed from the reservation list at the time when they become the oldest in the list. It is at this time that the actual reservations are made. The pessimistic protocol orders conflicting requests through the use of the reservation list. It does not affect concurrency of nonconflicting requests.

It should be used for transactions that result in a costly restart.

The optimistic protocol does not require any reservations prior to execution. It locks data objects and executes the transaction at its site of origin. Then, if the database is replicated, it broadcasts reservation requests for all other copies of the data objects. Updates at the transaction's site of origin are not permanent until all other sites' acknowledgments have been received. In case of conflict, the transaction is restarted. It may even be restarted indefinitely because restarted transactions are assigned a new timestamp each time. This protocol should be used for transactions that are known to have a low probability of conflicts and for transactions that update large portions of the database. In these two cases, making reservations would be inefficient.

2.2. Deadlock

Deadlock is a waiting situation that forms a cycle in the system's resources allocation graph (see Figure 2.1). This is caused by a set of transactions where each transaction waits for another one. When shown on a graph, each node represents a transaction and a directed edge goes from the transaction that is waiting to the transaction

that is being waited for. This graph is often called a wait-for graph.

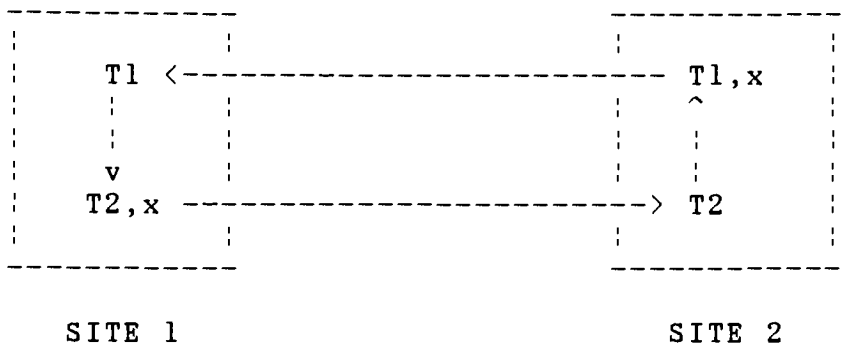

Figure 2.1

Wait-for graph showing a deadlock situation. At site 1, transaction T1 must wait until transaction T2 releases data item X. However, transaction T2 is waiting to get the copy of X that transaction T1 is holding at site 2.

There is another form of wait-for graph. It is constructed in the following way. When a transaction requests a data item, if the request can be granted, the pointer associated with that data item is set to point to the transaction (e.g., in Figure 2: C--->T2). However, if the request cannot be granted, the pointer associated with the transaction is set to point to the requested data item (e.g., in Figure 2: T1--->C).

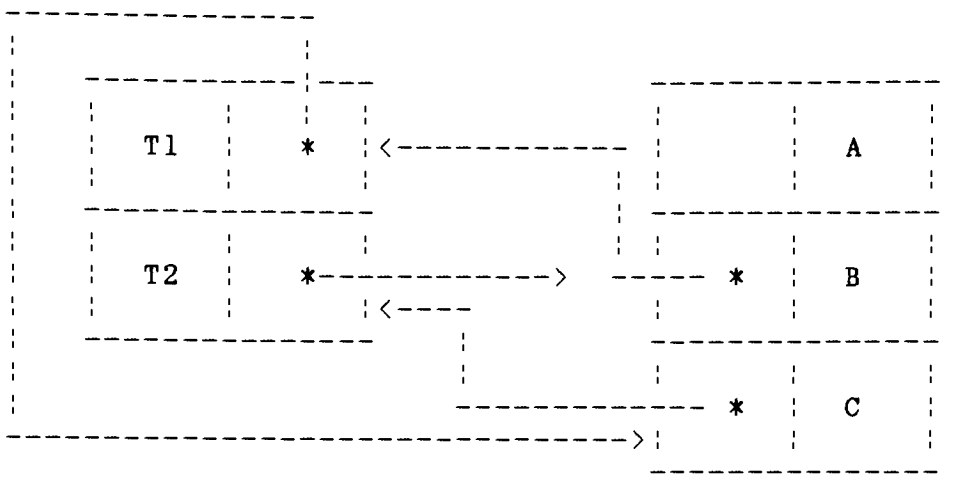

Figure 2.2

Another form of wait-for graph.
A deadlock is shown by the loop T1-C-T2-B-T1.
Transaction T1 is waiting for data item C, which is being used by transaction T2. Transaction T2 is waiting for data item B, which is being used by transaction T1.

In a deadlock in a distributed system, a transaction can be waiting for another transaction to release a resource it needs, or it can be waiting for another transaction to perform a required function. This other transaction can be executing at the same site or at a different site from that where the first transaction is executing. Dealing with deadlock in a distributed system is more difficult than in a centralized one because a deadlock can involve more than one site and therefore, can require transactions carrying information between the different sites. A deadlock involving two or more sites is

called a global deadlock.

There are three ways in dealing with deadlocks: detection, avoidance and prevention.

2.2.1. Deadlock Detection

Under the deadlock detection approach, transactions wait for each other and are only aborted when a deadlock actually occurs. A deadlock is detected by constructing a wait-for graph and looking for cycles in it. For deadlock detection, the pointers in the wait-for graph (see figures above) are traversed until either a null pointer is found or a pointer pointing to the transaction from which the search began is found. The second case indicates a cycle has been found; which in turn indicates a deadlock situation exists. The deadlock is then resolved by aborting one of the transactions in the cycle.

Deadlock detection does not only involve finding out that a deadlock situation exists. It requires the abortion of a transaction, so that its resources can be released and other waiting transactions can be allowed to continue their execution. It also requires to undo all the updates that the transaction has already done, which is called roll back. A roll back is done with the purpose of leav-

ing the database in a consistent state. Then, it requires the transaction to be restarted. The goal is to minimize the cost of aborting a transaction. Therefore, the criteria for selecting which transaction should be aborted when a deadlock situation is detected, is very important. Some of the alternatives are: 1) to abort the transaction that requires the less number of updates to be undone; 2) to abort the youngest transaction, i.e., the one that began executing last; 3) to abort the transaction with the longest expected time to complete; 4) to abort the transaction that owns the less number of resources.

Deadlock detection in a distributed system involves the transmission of information between different sites. Delays in the transmission of this information can cause false deadlocks to be detected. As an example of a false deadlock, suppose there are two transactions executing at different sites. Transaction 1 is waiting for transaction 2 to release the lock on data item X. Transaction 2 releases the lock and requests a data item that is locked by transaction 1. If the deadlock detector receives the information that transaction 2 requested a data item being held by transaction 1, before receiving the information that transaction 1 is no longer waiting for transaction 2, a false deadlock will be detected. As a solution to dealing with false deadlocks, the information can be gathered

a second time and only if the deadlock is real it will still be detected. However, on systems where the detection of false deadlocks has a very low percentage of occurrence, they can be treated as real deadlocks without degrading the performance of the system.

The disadvantages of deadlock detection are:

1) possible response time degradation

2) unnecessary abortion of transactions.

These disadvantages are due to the necessary periodic transmission of information between sites. Site failures, communication system failures and transmission delays may cause a deadlock to go undetected for some time, or may even cause a false deadlock to be detected.

There are several methods for detecting deadlocks in a distributed system: the timeout method, the centralized deadlock detector, the hierarchical deadlock detector, and the distributed method.

2.2.1.1. The Timeout Method

In the timeout method of deadlock detection, when a transaction has been on a wait state for a specified time interval, it is aborted, rolled back and restarted. However, it is difficult to select a workable timeout interval. When the timeout interval is too long, time is wasted by deadlocked transactions before they are timed out, aborted and restarted. On the other hand, when the timeout interval is too short, many transactions that are not deadlocked are restarted unnecessarily. In congested systems, a too short timeout interval can make the situation even worse. Transactions may take longer to complete due to the system's overload. If by this reason they are timed out and restarted, they will be adding to the system's overload. Therefore, the timeout method is only acceptable for lightly loaded systems.

2.2.1.2. The Centralized Method

Under centralized deadlock detection, all sites contain a local deadlock detector that is responsible for discovering local deadlocks, but one site is chosen as the system-wide deadlock detector. All other sites periodically send their local wait-for graphs to the centralized deadlock detector. The deadlock detector: 1) combines all

the local graphs into a system-wide wait-for graph; 2) searches for any cycles in it; 3) selects the transaction to be aborted.

Centralized deadlock detection is simple. However, its disadvantages are:

1) involves a loss of autonomy for all other sites

2) is vulnerable to failure of the centralized deadlock detector's site

3) requires high communication costs

4) is difficult to determine the optimal length of the period over which sites must send their local wait-for graphs to the central deadlock detector.

•

2.2.1.3. The Hierarchical Method

Under hierarchical deadlock detection, the database sites are organized into a hierarchy or tree. The leaves of the tree contain local dealock detectors, while the nonleaf nodes contain nonlocal deadlock detectors. Local deadlock detectors determine local deadlocks and transmit their local wait-for graphs to the nonlocal deadlock detector at the immediately higher level in the hierarchy. Nonlocal deadlock detectors determine deadlocks involving

only the nodes below them. They also transmit wait-for graphs to higher levels in the hierarchy.

Hierarchical deadlock detection is aimed at detecting deadlocks by a site located as close as possible to the sites involved in the cycle. The choice of the hierarchy influences deadlock detection. A subtree should be formed by a group of sites having a high percentage of the database accesses within itself. Communication costs can be optimized if the appropriate topology of the hierarchy is chosen.

2.2.1.4. The Distributed Method

In the distributed deadlock detection method, each site has the same responsibility for detecting deadlocks. Any blocked transaction can start a deadlock detection computation in order to determine whether it is involved in some deadlock. The intention is to have a more reliable mechanism in environments subject to failures [OBER82].

References to or from remote sites are represented in the graph at the local site by a special node labeled as external. A directed edge pointing from a transaction to the external node means the transaction is waiting for

some data item located at another site. A directed edge pointing from the external node to a local transaction means a remote transaction is waiting for the local transaction. A potential deadlock cycle is represented at a local wait-for graph as a path beginning and ending at the external node:

e.g., external-->Ti-->Tj-->...-->Tk-->external.

When no potential deadlock cycles exist in the local wait-for graphs, then no global deadlock exists. However, when a potential deadlock cycle is detected, the wait-for graph is transmitted to the site represented by the external node in the local graph. In the example of the path in the paragraph above, this means the site for which Tk is waiting. The receiving site can then add the information to its local graph, check for cycles, and either transmit to another site (if only a potential deadlock is discovered) or abort a transaction to resolve a deadlock (if a real deadlock is discovered).

In reality, the potential deadlock information can be transmitted in any direction along the cycle; but this will only lead to unnecessary transmission of information and the same deadlock being discovered at more than one site. To avoid this, a potential deadlock cycle is transmitted only if the identifier of the transaction for

which the external node waits is greater than the identifier of the transaction waiting for the external node.

The potential deadlock information transmitted consists of a string containing the identifiers of the transactions, in the order in which they appear in the cycle.

2.2.2. Deadlock Avoidance

In deadlock avoidance techniques, the possibility of a deadlock is determined before data items are assigned to a transaction. Every time a transaction requests a data item, it is determined whether there is a way for all already initiated but not yet completed transactions to finish. Deadlock avoidance requires transactions to preclaim the data items that will be accessed. However, preclaiming of data items reduces concurrency.

Deadlock avoidance techniques are rarely found to be used in conjunction with distributed database systems. They are not practical in such systems because data items are distributed enough to make these techniques inefficient. Also, deadlock avoidance techniques are not feasible in systems where the data items to be accessed are computed dynamically; that is, they are not fixed.

2.2.3. Deadlock Prevention

Deadlock prevention preconditions the system to remove any possibility of deadlocks occurring. Transactions are aborted and restarted whenever there is a possibility that they may lead to a deadlock.

Whenever a transaction requests a data item that is being held by another transaction, the requesting transaction is allowed to wait only if it is ensured (by performing a wait-for graph analysis) that no deadlock will occur. Otherwise, one of the conflicting transactions is aborted. As with deadlock detection, the goal is to minimize the cost of aborting transactions. However, in deadlock prevention, the selection of the transaction to be aborted depends on the specific method being used. Three basic methods have been proposed: the nonpreemptive method, the preemptive method, and the preordering of resources.

There are some disadvantages associated with deadlock prevention techniques. The first two methods may lead to unnecessarily restarting transactions that, although in conflict, are not involved in a deadlock. This may happen because no wait-for graph is used by these methods. The method of preordering of resources can result in poor database utilization, as will be seen below.

2.2.3.1. The Nonpreemptive Method

The nonpreemptive method for deadlock prevention is also called Wait-Die, because a requesting transaction found to be in conflict with another transaction, either waits or dies (is aborted).

Each transaction must have a unique identifier. By assigning transactions a timestamp at the time at which they begin executing, they are granted a unique local identifier. However, since it is a distributed system, the site identifier must be appended in order for them to have a system-wide unique identifier.

Whenever a transaction requests a data item being held by another transaction, the requesting transaction is allowed to wait only if it is older than the other transaction, that is, if it has an earlier timestamp. Otherwise, it is aborted and restarted with the same timestamp. In this way, only younger transactions are restarted.

An advantage of this approach is that transactions can only be aborted and restarted when they request data items for the first time. This means that if a transaction accesses all its required data items before interacting with external devices, it is guaranteed that it will not be aborted by the deadlock prevention mechanism in the middle of output.

This method has two disadvantages:

1) a transaction may be aborted and restarted several times

2) the older the transaction gets, the more and more younger transactions it may have to wait for.

A transaction may be aborted and restarted several times because after being aborted and restarted, it will request the same data item as before. If the transaction holding that data item has not completed its execution, the requesting transaction will be aborted and restarted again and again, depending on how long the other transaction holds the requested data item. The older a transaction gets, the more and more younger transactions it may have to wait for, because older transactions wait for younger ones.

2.2.3.2. The Preemptive Method

The preemptive method for deadlock prevention is also called Wound-Wait because in a conflict, the requesting transaction either waits or wounds the other transaction. To wound a transaction means that a message is sent to every site the wounded transaction has visited and if the

message arrives before this transaction has initiated termination, it is aborted and restarted.

As in the nonpreemptive method, each transaction must have a unique identifier. However, in the preemptive method, whenever a transaction requests a data item being held by another transaction, the requesting transaction is allowed to wait only if it is younger than the other transaction. In the case where the requesting transaction is the older, the other transaction is aborted and restarted, and the requested data item is granted to the requesting transaction. Notice that only younger transactions are aborted and restarted.

The advantages of the preemptive method for deadlock prevention are:

1) an older transaction never waits for a younger one, meaning that as a transaction gets older it gets increased priority

2) the preemptive method induces fewer restarts than the nonpreemptive method.

The preemptive method induces fewer restarts than the nonpreemptive method because in the preemptive method a transaction holding a data item may be aborted if an older transaction requests that data item. When the aborted

transaction is restarted, it will still be younger than the other transaction and will not be aborted but will be forced to wait instead.

The disadvantage of the preemptive method is that it holds no guarantee that a transaction will not be restarted even if it has acquired all the data items it needs. A restart will be required if an older transaction requests any of the data items it holds.

2.2.3.3. The Preordering of Resources Method

The preordering of resources is a deadlock prevention method that avoids restarts. It requires all data items to be numbered. Transactions request data items one at a time in numeric order. The requests are passed from node to node in the preassigned order. Each node examines the requests for data items located at that node. When all the requests have been granted, the node at which the requesting transaction was originated is informed and the transaction is allowed to execute. Othewise, the transaction is delayed until all transactions holding requested data items complete.

An advantage of this method of deadlock prevention is that an interactive user can terminate a delayed transac-

tion without any consequences, since it means the transaction has not been yet initiated.

The disadvantages of this method are:

1) it requires preclaiming of data items, which reduces concurrency

2) it requires data items to be obtained sequentially, which increases response time

3) it requires all data items to be numbered, which is difficult to achieve in a database because data items are constantly being created and deleted.

2.3. Recovery

Recovery means to resume correct operation after a failure or error. Many types of failure can exist in a distributed system. Widely recognized failures are: node failures and communication failures. A node failure occurs when a node ceases correct operation. If the node detects the error, gets itself into local recovery and loses no data, the failure is known as a soft failure. If the node loses data, the failure is known as a hard failure. A communication failure occurs when messages are not properly transmitted. Recovery facilities must be able to cope with

situations where more than one failure occurs simultaneously. This is known as multiple failures.

An appropriate recovery technique should be planned for any failure that can be identified in advance. Recovery facilities must be able to reapply the effects of committed (i.e., permanently recorded) transactions and to remove the effects of partially completed transactions from the database. Queries (i.e., read-only transactions) need only to be restarted, whenever a failure is detected. Since they do not alter the database, there is no problem in leaving them uncompleted. On the other hand, updates need some additional steps to be performed during normal operation in order to have the necessary information for a recovery.

In a distributed database, the recovery facilities at the different sites must be coordinated to uniformly retain or reject the effects of multi-site transactions. All sites must have local and distributed recovery facilities.

2.3.1. Local Recovery Facilities

Local recovery facilities are used in local site recovery. They allow each node to restore their portion of

the transaction in the event of a failure. They include: the backup copy, the transaction log, before and after images, and checkpoints [AFRA82].

In a partitioned database, since there is no redundancy, recovery for site (i.e., node) failures can be handled by any of these facilities. In a replicated database, recovery of redundant data can be accomplished by transmitting a copy from another node.

2.3.1.1. The Backup Copy

The backup copy is a complete dump of the local database that is usually taken by the end of each work day. This is one of the most time consuming techniques. It consists of loading this copy to disk to replace the damaged database. It is used when the local database is left unreadable and must be totally restored. It can be used in conjunction with other recovery techniques.

2.3.1.2. The Transaction Log

The transaction log is a file that contains a record of all the changes to the database. It contains the information necessary to roll back the database to a previous

consistent state. It can be used together with the backup copy to incorporate, to a previous consistent version of the database, all the changes that occurred before the failure.

2.3.1.3. Before and After Images

A before image is a copy of that portion of the database that is going to be updated by a transaction. The copy is taken before the database is updated. It is used to return the database to the state it was before the initiation of an aborted transaction.

An after image is a copy of that portion of the database that was updated by a transaction. The copy is taken after the database is updated. It is used to return the database to a consistent state by incorporating to an earlier version of the database all changes made by those transactions which were committed before a given failure.

Both, before and after images are taken for each update transaction executed.

2.3.1.4. Checkpoints

Checkpoints are marks in the log file that indicate the time when the database was at a consistent state. For a checkpoint to be written, all incoming transactions must be suspended and all active transactions must be allowed to complete. Checkpoints are used to denote a point in time to which the state of the database can be restored. Restoration can then be done by loading a backup copy and incorporating all after images until a specified checkpoint is reached.

2.3.2. Distributed Recovery Facilities

In a distributed database, although each site may have its local recovery facilities to restore its portion of a transaction in the event of a failure, distributed recovery facilities are still needed to ensure that all sites handle the failure in a consistent way and to deal with communication failures such as lost messages and network partitions. A network is partitioned when it is divided into two or more parts with no communication path available between them.

There are many possible communication failures. However, most networks' communication systems are able to

take charge of most of these failures. As an example, sequence numbers are used to check for out-of-order messages; rerouting is used for physical line failures; and redundancy checks are used in detecting damaged messages.

Several algorithms have been designed to deal with site failures, lost messages and partitions from a distributed point of view: the short-time and long-time failure recovery, the two-phase commit, and the intention lists.

2.3.2.1. The Short-Time and Long-Time Failure Recovery

A short-time failure is one where only the central memory has been lost. It is sufficient to abort, roll back and restart those transactions which have not reached the ready to commit phase. A long-time failure occurs when the database has been damaged. It requires: 1) the local database to be restored with a backup copy; 2) modifications recorded in the log up to the last checkpoint to be applied to the restored database; 3) recovery to be synchronized with the other sites.

There are two strategies to deal with these kinds of failure: the wait strategy and the back-out strategy. These strategies are discussed in the following paragraphs.

The wait strategy consists of making transactions wait until recovery is finished. In a distributed database, transactions may need processing at several sites. Using the wait strategy for a site failure in which one of these multi-site transactions is involved, causes its processing at remote sites also to be stopped. It also prevents other transactions from using the resources that are being held by the waiting transaction. The cost of waiting, then, consists of the price for holding certain resources during recovery time. This is why the longer the recovery period, the higher the cost for waiting. The wait strategy seems to be adequate only for short-time failures.

The wait strategy begins by re-installing the operating system of the site that failed. The operating system then restarts the database system. The transaction monitor, a database system process in charge of transaction management, coordinates the recovery with the remote sites. It examines the transaction log and rolls back and restarts those transactions for which the end of a phase is not indicated. A failure during the recovery procedure will only mean that another recovery must be initiated.

On the other hand, by using the back-out strategy, all blocked transactions are rolled back and other transactions are able to use the released resources. However,

the back-out strategy can be expensive in terms of undoing the effects of noncommitted transactions and restarting them. This cost includes: the time to back-out, the time to restart and reach again the execution point where the failure occurred, and the time to send messages to other sites to coordinate recovery. It should be noticed that this cost does not depend on the media recovery time. Therefore, the back-out strategy seems adequate only for long-time failures, where other transactions can make use of the released resources during recovery.

The back-out strategy consists of rolling back all uncommitted transactions. It begins when the site which detected the failure sends a back out request to all other sites. If the database is replicated, a modified primary copy approach is used. The primary copy is updated and then updates are sent to all other sites. If a failure occurs at the primary copy, one of the other sites is selected as the new primary copy. One way of doing the selection can be by having a predefined ordering of all sites and selecting the next site in turn.

The back-out strategy is also useful for network partitions. It allows continued operation after a network partition has occurred. If a replicated database exists, only that subnetwork holding the primary copy is allowed to process updates during a network partition. Another way

of handling such situations is by allowing only that subnetwork which contains more than half of the original network, to select a new primary copy.

The basic idea is to use the wait strategy or the back-out strategy depending on the expected duration of the recovery. The aim is to select for each failure situation the strategy with minimal costs. Based on the observation that the wait strategy seems to be cheaper for short-time failures while the back-out strategy seems cheaper for long-time failures, an algorithm for selecting the least costly strategy for each transaction has been proposed [WALT80b].

The algorithm identifies all affected transactions. It then computes the time for sending messages around the network to coordinate the back-out. If this time is greater or equal to the media-recovery time, the wait strategy is selected. Otherwise, the time for sending messages around the network to coordinate the back-out is added to the back-out time and to the time to restart and reach the state previous to the failure. If the sum is greater or equal to the media-recovery time, the wait strategy is selected; otherwise the back-out strategy is selected.

2.3.2.2. The Two-Phase Commit

The basic idea of the two-phase commit is to ensure that all sites either commit a transaction or abort it. All sites have the autonomy to decide whether to commit or abort a transaction until the moment they agree to enforce a common decision. At this point, the site designated as the coordinator is responsible for taking the final decision, based on what it heard from the other sites. Only if all the sites are ready to commit, the coordinator will reach the commit decision. Otherwise, it will decide to abort the transaction.

Phase one begins when the coordinator records in a log in non-volatile storage the identifiers of all subtransactions which are part of the transaction being considered. It sends a message to all participating sites asking them if they are ready to commit the transaction. It also activates a timeout which will force the transaction to be aborted if any of the sites does not respond. A site will answer as ready to commit only if it has been able to write all its subtransaction's records in its log in non-volatile storage. This will ensure that it has available all the information necessary to commit the transaction even in the event of a failure. Otherwise, the site will write an abort message into its log and will answer with an abort message to the coordinator. If all

sites answered that they were ready to commit, the coordinator will decide to commit the transaction. However, if one or more sites answered with an abort message or if the timeout expired, the coordinator will decide to abort the transaction. As can be seen, at the end of phase one a common decision has been reached.

The second phase is where the decision is carried out. It starts when the coordinator writes its decision into its log. This means that the transaction will eventually be committed or aborted, whatever was decided, even in the event of a failure. Then the coordinator sends its decision to all sites. The sites write this decision into their logs, execute it, and send an acknowledgment to the coordinator. This acknowledgment is different from the regular acknowledgment used by the communication system to report that a message has been received. It means, not only that the decision message was received, but that the decision was recorded in the log in non-volatile storage. When the coordinator receives the acknowledgments from all the sites involved, it records the transaction as completed in its log.

A failure can occur at any point in this procedure. However, the valuable aspect of the two-phase commit protocol is that it is capable of recovering from all failures as long as they do not involve the loss of the

information recorded in the log. If the coordinator site fails, the recovery mechanism will be able to know at which phase it was before failing by looking at the records in the log. If the log holds the identifiers of all the subtransactions involved, but there is no commit or abort record, then the coordinator was at phase one. This means that after recovery the coordinator must resume execution by asking all sites again if they are ready to commit. On the other hand, if the coordinator was at phase two when the failure occurred, the log will show either a commit or an abort record, but no record that the transaction was completed. In this case, the coordinator must send again its decision to all sites.

Another possibility is for a failure to occur at a site different from the coordinator. If the site has not recorded in its log that it is ready to commit, the coordinator will timeout and the decision will be to abort the transaction. However, if the ready to commit record appears in the site's log, when the failed site recovers, it must ask the coordinator to retransmit its decision. In case of a failure of the coordinator or in case of a network partition, the site can ask other sites for the decision. As long as one of the sites in the group has received the decision, the transaction can be completed. If this is not so, the site must wait until the decision

can be received from the coordinator.

When a message from the coordinator is lost, it will not receive an answer from the site; the coordinator's timeout will expire; and the transaction will be aborted. However, if the sites are expecting the decision, they should have a timeout to request a repetition of the message. This prevents a transaction from being aborted, after the coordinator has decided to commit, due to the loss of a message.

2.3.2.3. The Intention Lists

The intention list is a list of all actions necessary to complete the write commands of a transaction. When a transaction is executed, it creates the intention list first. Then the actions in the intention list are carried out. It is at this time that the real update of the database occurs. If the update is successful, the intention list is deleted.

If the system fails before the intention list is completed, it means that the transaction has not finished and that nothing has been written into the database. Therefore, at recovery time the only thing that has to be done is to abort the transaction. On the other hand, if the

intention list was completed and carried out, it must have been erased. This means that the transaction already completed and nothing has to be done by the recovery procedure. However, if the intention list was completed but not erased, it means the transaction has not completed yet and the recovery procedure must carry out the actions of the intention list.

3. Presentation of Models

Three distributed databases will be presented in the following sections: SDD-1, System R*, and Distributed INGRES. Emphasis will be given to the synchronization, deadlock and recovery mechanisms they use.

3.1. SDD-1

The System for Distributed Databases - 1 (SDD-1) is a distributed database system implemented by the Computer Corporation of America. It consists of a collection of up to several hundred geographically dispersed sites connected by the ARPANET network. The system uses a relational model and supports replicated data. It has three basic subsystems: transaction modules, data modules and a reliable network. Each transaction module (TM) is in charge of transaction execution. Each data module (DM) is in charge of the manipulation of local data. The reliable network (RELNET) is a network communication facility which provides a global network clock and guarantees an ordered delivery of messages, even in the event of site failures.

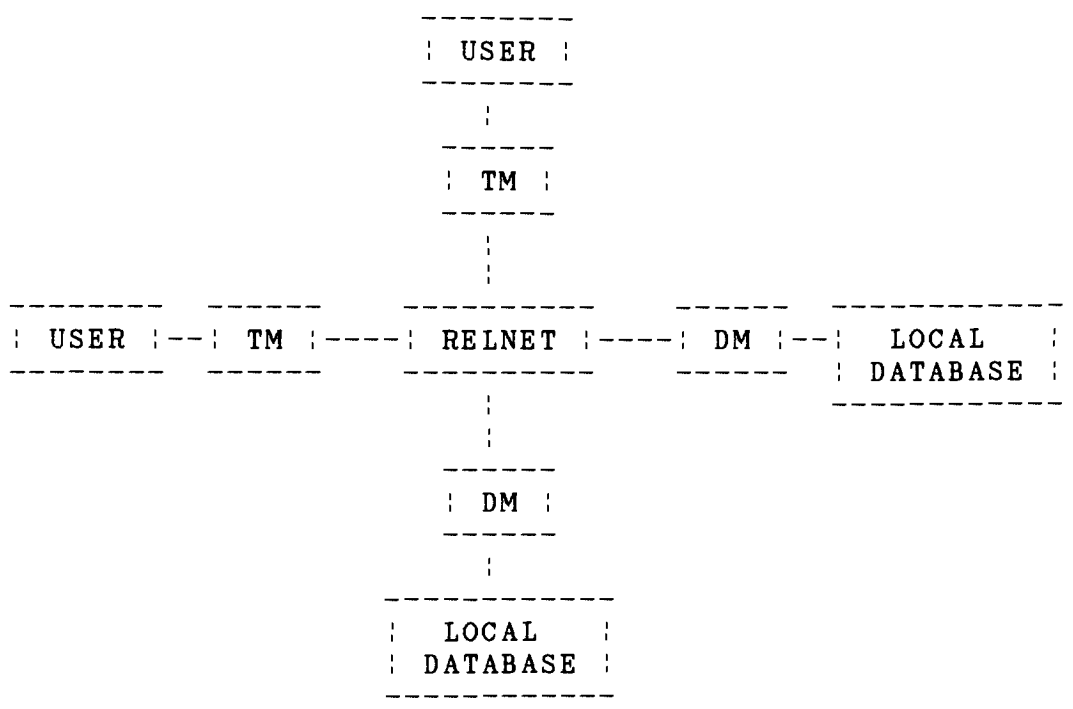

Figure 3.1

SDD-1 Architecture

3.1.1. SDD-1 Synchronization Strategy

The aim of the synchronization mechanisms used in SDD-1 is to use the minimal internode synchronization required and to allow the greatest possible degree of concurrency while preserving database consistency. Database consistency is violated if update messages sent by two or more transactions that update the same data item are processed in different order at different sites. SDD-1

synchronization mechanisms consist of: a number of predefined transaction classes, a conflict graph analysis done at the system's design time, a set of four protocols each of which provides a different level of synchronization, and the use of timestamps.

3.1.1.1. Transaction Classes

A transaction class is defined by a read-set and a write-set. A read-set is a set of data items used as input by a transaction. A write-set is a set of data items that are updated by a transaction.

At database design time, the database administrator must assign each expected transaction to a transaction class. This assignment is based on the read-set and write-set of the transactions. If the transaction's read-set and write-set are contained in the read-set and write-set of the transaction class, the transaction can be assigned to that transaction class.

A transaction module is associated with each transaction class. Since transactions within the same transaction class are in conflict (i.e., they have the same read-set and write-set), the transaction module processes them in serial order, according to their timestamps.

Transaction classes overlap if their read-sets and/or write-sets intersect. However, two transaction classes are said to be in conflict only if the read-set or write-set of one intersects the write-set of the other. Transactions in different classes can conflict only if their corresponding classes conflict. Therefore, conflicts between transactions can be determined by conflicts between transaction classes. Conflicts between transaction classes are detected by a conflict graph analysis. Only those transactions that lie on cycles on the conflict graph need to be synchronized.

3.1.1.2. Conflict Graph Analysis

The purpose of the conflict graph analysis is to determine which transactions need to be synchronized and what level of synchronization they require. A conflict graph is constructed by creating one node for the read-set and one node for the write-set of each transaction class.

Since transactions pertaining to different transaction classes conflict only if their classes conflict, conflict graph analysis is performed on transaction classes rather than on transactions. An undirected edge must link the read and the write node belonging to the same transaction class. Then, undirected edges are drawn between

pairs of nodes representing the intersection of the read-set or write-set of one transaction class with the write-set of another.

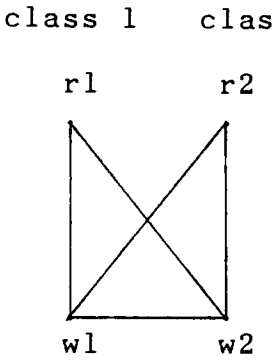

Figure 3.2

Conflict graph showing two transaction classes at conflict. Class 1 has nodes r1 (read-set) and w1 (write-set). Class 2 has r2 (read-set) and w2 (write-set). Both classes may read and write the same data items.
Example: Class 1 has read-set {x,y}
write-set {y}
Class 2 has read-set {y}
write-set {x,y}

Once the graph is constructed, it is inspected for cycles. Edges contained on a cycle represent conflicts requiring synchronization. Through the analysis of the conflict graph an adequate protocol (i.e., synchronization algorithm) can be chosen for each transaction class. For example, protocol 1 is used for transaction classes that

are not involved in a cycle in the conflict graph; while protocol 3 is used for transaction classes involved in a cycle containing an edge between two write-sets. A protocol table is created from this analysis. A protocol table maps transaction classes to the protocols they require for synchronization. At run time, the protocol table is used to find the protocol needed to run a given transaction.

Conflict graph analysis is performed once, at database design time, because to perform it at execution time will be time consuming and will require a great amount of internode communication.

3.1.1.3. Protocols

The protocols are algorithms used for transaction synchronization. The purpose of the protocols is to force the ordering of transactions' read and write actions so that data integrity and consistency is maintained.

All protocols make the following assumptions: 1) sites transmit messages in timestamp order; 2) the communication facility (RELNET) guarantees an ordered delivery of the messages; 3) transactions are atomic, i.e., no intermediate effects can be observed; 4) a transaction is executed at its site of origin and a list of the updates

is immediately sent to all other sites. It should be noticed that the list of updates contains the final values of the data items, not the computations to be performed on them.

SDD-1 has four protocols, each of which provides a different level of synchronization. All protocols use timestamps to resolve conflicts.

3.1.1.3.1. Protocol 1

This protocol offers the least synchronization control. It provides no synchronization beyond a local locking mechanism and a guarantee that messages are delivered in timestamp order. It has the effect of executing the read messages from one transaction and the write messages from another transaction in the same order at all sites. The only intersite communication consists of broadcasting the update messages, therefore, execution of transactions under this protocol is fast.

Step 1: At the site where the transaction originates, set share locks on the transaction's read-set. Set exclusive locks on the transaction's write-set.

Step 2: Execute the transaction locally (i.e., only at the site where it originates).

Step 3: Broadcast the update messages to all other sites.

Step 4: Release all the transaction's local locks and inform the user that the transaction has been executed.

3.1.1.3.2. Protocol 2

Protocol 2 is used for read-only transactions that are involved in a cycle (see Figure 3.3). It uses timestamps and preclaims all nonpreemptive data items in an effort to avoid deadlocks.

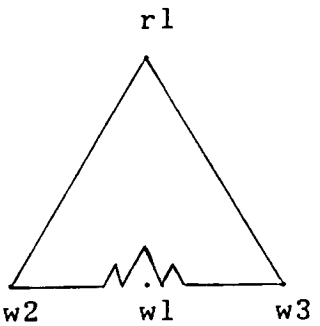

Figure 3.3

Graph showing a cycle formed by read-only transactions. There is no vertical edge.

Step 1: Assign to the transaction the most recent timestamp of any data item which belongs to the transaction's read-set.

Step 2: Send to all sites a request message; the transaction's timestamp must be included.

Step 3: Wait for either an accept message or an update message from every other site. Update messages received during this wait period are executed as long as their timestamp is less than that of the transaction being waited for, or the data items to be updated are not part of the read-set of the transaction being waited for. Assuming that messages are received in timestamp order, this wait period ensures that the site receives and processes all messages with timestamp value less than that of the transaction being waited for.

Step 4: One of three possibilities may happen: 1) the other site is processing a transaction with a timestamp value greater than that of the request message and will send back an accept message with timestamp value equal to that of the transaction it is processing; 2) the other site is processing a transaction with a timestamp value less than that of the request message and will send back an accept message with timestamp value equal to the maximum of the current time at that site or the timestamp

received; 3) the other site is idle and will send back an accept message with timestamp value equal to the maximum of the current time at that site or the timestamp received.

Step 5: When all necessary messages have been received, set (at the local site) share locks on the transaction's read-set and exclusive locks on the transaction's write-set.

Step 6: Execute the transaction locally.

Step 7: Broadcast update messages to all other sites.

Step 8: Release all locks.

Step 9: Resume execution of update messages; including those which were forced to wait during the wait period (see step 3).

3.1.1.3.3. Protocol 3

Protocol 3 achieves serializability (i.e., execution of transactions in serial order) by requiring that conflicting read and write messages be executed in timestamp order. In this way it also guarantees that the data read by a transaction is up-to-date. It can guarantee the serializability of all transactions in a distributed

database system, because timestamps are system-wide unique. Like protocol 2, it also uses timestamps and pre-claims all nonpreemptive data items in an effort to avoid deadlocks.

Step 1: Select as the transaction's timestamp any time value as long as there has been no other transaction processed by the site with a greater timestamp than the one selected.

Step 2: Send to all sites a request message that includes the transaction's timestamp.

Step 3: Wait for either an accept message or an update message from every other site. Update messages received during this wait period are executed as long as their timestamp is less than that of the transaction being waited for, or the data items to be updated are not part of the read-set of the transaction being waited for. Assuming that messages are received in timestamp order, this wait period ensures that the site receives and processes all messages with timestamp value less than that of the transaction being waited for.

Step 4: One of three possibilities may happen: 1) the other site is processing a transaction with a timestamp value greater than that of the request message and will send back an accept message with timestamp value

equal to that of the transaction it is processing; 2) the other site is processing a transaction with a timestamp value less than that of the request message and will send back an accept message with timestamp value equal to the maximum of the current time at that site or the timestamp received; 3) the other site is idle and will send back an accept message with timestamp value equal to the maximum of the current time at that site or the timestamp received.

Step 5: When all necessary messages have been received, set, at the local site, share locks on the transaction's read-set and exclusive locks on the transaction's write-set.

Step 6: Execute the transaction locally.

Step 7: Broadcast update messages to all other sites.

Step 8: Release all locks.

Step 9: Resume execution of update messages; including those which were forced to wait during the wait period (see step 3).

3.1.1.3.4. Protocol 4

Protocol 4 offers the higher degree of synchronization control. It is the most expensive. It is used for unanticipated transactions.

Step 1: Select as the transaction's timestamp any time value as long as no site has processed a transaction with a greater timestamp.

Step 2: Send to all sites a request message that includes the transaction's timestamp.

Step 3: Wait for an accept message from every other site. If a reject message is received, the transaction is restarted with a greater timestamp. Update messages received during this wait period are executed as long as their timestamp is less than that of the transaction being waited for, or the data items to be updated are not part of the read-set of the transaction being waited for.

Step 4: If the other site is processing or has processed a transaction with a timestamp value greater than that of the request message, it will send back a reject message. Otherwise, it will send back an accept message, will set its clock to the timestamp value received in the request message, and will agree to execute its next transaction using only either protocol 3 or protocol 4.

Step 5: When all necessary messages have been received, set, at the local site, share locks on the transaction's read-set and exclusive locks on the transaction's write-set.

Step 6: Execute the transaction locally.

Step 7: Broadcast update messages to all other sites.

Step 8: Release all locks.

Step 9: Resume execution of update messages; including those which were forced to wait during the wait period (see step 3).

3.1.1.4. Timestamps

A timestamp indicates the time at which a transaction was initiated. Each transaction has a globally unique timestamp. This is achieved by reading the clock value and appending the site identifier to the low order digits. Clocks can be kept approximately synchronized by setting their values, whenever a message with a greater timestamp is received, to the value of the received timestamp plus one.

In SDD-1, a transaction is first executed at the site where it originates. Then, a list of its updates is

broadcasted to all other sites. Update messages contain the transaction's timestamp and the final values data items must have after being updated (not the computations to be performed on them).

Each data item have an associated timestamp that indicates the time when the item was last updated. Each copy of a data item has its own timestamp. For an update to be performed, the transaction's timestamp is compared to the timestamp of the data item. If the transaction's timestamp is greater than that of the data item, the update's value is written into the data item and stamped with the timestamp of the updating transaction. Otherwise, the update message is considered obsolete and is ignored.

Since transactions in the same transaction class are conflicting, they are executed in timestamp order.

The disadvantage of having timestamps associated with data items is the high storage cost incurred.

3.1.2. SDD-1 Deadlock Strategy

The SDD-1 database management system was thought to be deadlock free. It was designed to use timestamps as a deadlock prevention technique. Since timestamps in SDD-1

are globally unique, they give a system-wide ordering which is used in conflict resolution. It was believed that transactions only waited for the completion of transactions with smaller timestamps. However, Mc Lean has shown that the system is not deadlock free [MCLE81]. This is described below.

Suppose there are two transaction classes, i and j. If the read-set of class i is equal to the write-set of class j, and the read-set of class j is equal to the write-set of class i, the conflict graph will look as that of figure 3.4. A transaction running on class i will wait for either a write message or a nullwrite message from class j with a timestamp greater than its own. This is the only way to be sure that the data item to be read by the class i transaction is up-to-date. At the same time, a transaction from class j can be waiting for a write or nullwrite message from class i. The system is deadlocked.

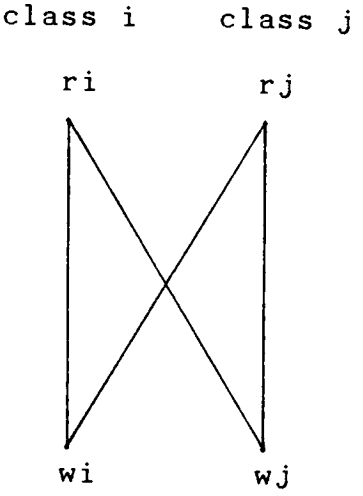

Figure 3.4

Possible deadlock situation in SDD-1.

A suggested solution to this problem is to use timeouts. Then, when a timeout expires, a request for a nullwrite message can be sent.

3.1.3. SDD-1 Recovery Strategy

SDD-1 uses the ARPANET network facility. ARPANET is a network facility which has the ability to correct errors such as lost, duplicate and damaged messages. ARPANET also has the ability to detect a failure of a communication line and to reroute any message sent through that line [TANE81]. However, other things must be guaranteed for the system to be reliable. The system must continue

to operate correctly even if site and/or communication failures occur. Transactions at nonfailed sites must be able to continue their execution without causing updates to be performed in different order at different sites. To provide these things, SDD-1 uses an extended communication facility known as the Reliable Network (RELNET).

RELNET runs on top of the ARPANET. It is composed of three software layers: the Global Time Layer, the Guaranteed Delivery Layer, and the Transaction Control Layer. Each layer provides some facilities for achieving reliable communication and coordination among the sites comprising the database.

The Global Time Layer is composed of four sublayers: the local clock sublayer, the global clock sublayer, the local status sublayer, and the global status sublayer.

The local clock sublayer supports a real time clock and a logical local clock at each site. The real time clock is used by timeout mechanisms. The logical local clock is used by timestamp mechanisms and consists of a counter which is increased by one unit each time it is read. In the event where a message received has a timestamp greater than the value of the logical local clock, this clock is set to the value of the timestamp received plus one.

The global clock sublayer not only provides the global clock, but also provides a facility for detecting site failures. Site failures are detected by requesting periodically the acknowledgment of a message sent. If a timeout expires before the acknowledgment is received, a message is sent to the site to start a recovery procedure.

The local status sublayer maintains a status table which tells whether each site is up or down. It also provides a facility for informing a waiting process when the recovery of a site is completed.

The global status sublayer coordinates the facilities provided by the other sublayers.

RELNET's second layer is the Guaranteed Delivery Layer. It deals with failures of receiving sites. When a message is sent to a failed site, it is stored at a spooler. A spooler serves as a first-in first-out message queue. An acknowledgment is sent to the sending site. When the failed site recovers, it receives the queued messages from the spooler and brings its local database up to date. Notice that no transaction is forced to wait for the recovery of a site for committing, as long as its update messages are stored in the spoolers. A higher degree of reliability is achieved by increasing the number of spoolers per site.

RELNET's third layer is the Transaction Control Layer. It deals with failures of sending sites. It guarantees transaction atomicity, i.e., a transaction is committed at all sites or at none. It uses a four-phase commitment protocol. It also uses backup processes that can substitute the coordinator in the event of a coordinator's failure. The degree of reliability depends on the number of backup processes.

The four-phase commit protocol works as follows. On phase one the coordinator establishes a set of ordered backups and waits until an acknowledgment of their existence is received. On phase two the coordinator sends the update messages and wait for acknowledgments. On phase three the coordinator sends its decision to commit or abort a transaction to the backup processes and waits for acknowledgments. On phase four the coordinator sends its decision to commit or abort a transaction to each site; waits for acknowledgments and then destroys the backups.

RELNET is resilient to the failure of some of its parts, but it is not resilient to some failures called catastrophe situations. Some examples of RELNET's catastrophe situations are: 1) unavailability of all copies of a data item, 2) failure of a receiving site and all its associated spoolers, 3) failure of the coordinator site

for the four-phase commitment and all its backup processes, 4) network partitions. In the event of any catastrophe situation, manual procedures may be required for recovery. A recovery of this type may require reinitializing the system.

Table 3.1 The RELNET

Layer	Facilities
Global Time Layer	
Local clock sublayer	- real time clock - logical local clock
Global clock sublayer	- global clock - detection of site failures - initiation of site recovery
Local status sublayer	- status table - recovery completion inf.
Global status sublayer	- coordinates facilities of the other sublayers
Guaranteed Delivery Layer	- spoolers to save messages until the receiving site recovers
Transaction Control Layer	- transaction atomicity using four-phase commit

3.2. System R*

System R* is the distributed version of an IBM Corporation's relational database called System R. It is composed of several cooperating autonomous databases connected by the IBM CICS Inter System Communication facility. It has four basic components: a storage system, a data communication system, a transaction manager and a database language processor. The storage system is the local database management facility. It is in charge of the actual storage and retrieval of data. The data communication system is in charge of message transmission. The transaction manager coordinates multi-site transactions. The database language processor translates programs written in the SQL data manipulation language into operations to be executed by the storage system.

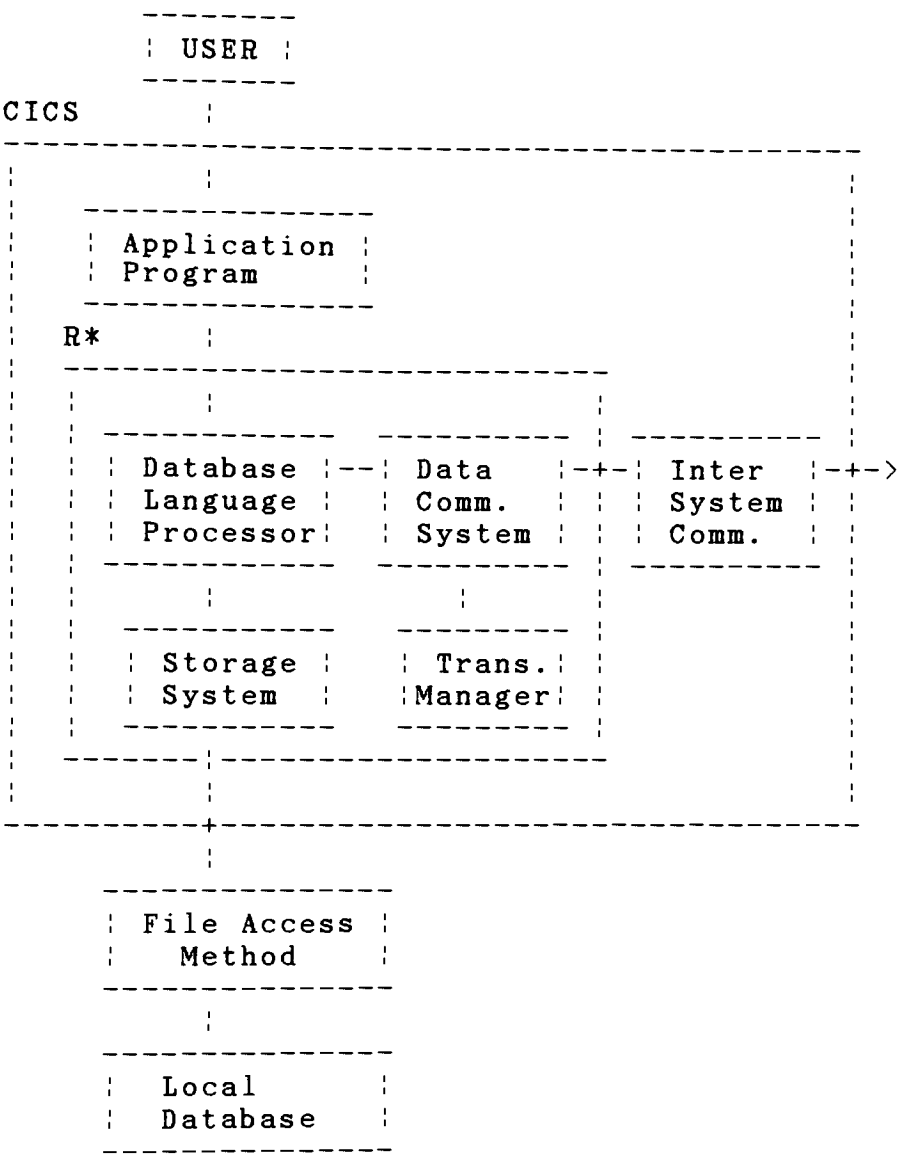

Figure 3.5

R* Architecture

3.2.1. System R* Synchronization Strategy

System R* uses two-phase locking to provide synchronization. During the first phase, locks for all data items to be updated are acquired. No locks are released during this phase. If a requested lock cannot be granted, the transaction must wait.

After all required locks are acquired, the transaction proceeds. Locked data items cannot be accessed by any other transaction, that is, locks provide synchronization by allowing data items to be accessed by only one transaction at a time.

All locks are held until a commit or abort decision is reached. On phase two, locks are released. No locks can be acquired on this phase.

3.2.2. System R* Deadlock Strategy

In System R* each site has a deadlock detector. Deadlocks local to a single site are detected by a cycle in the local wait-for graph.

Global deadlocks involve transactions at several sites. They must be detected in a distributed way. A site involved in the global deadlock will detect a poten-

tial deadlock cycle in its wait-for graph by a path beginning and ending in an external node. This site then transmits the wait-for graph to the site represented by the external node. The receiving site adds this information to its local graph and checks for cycles. If a deadlock is detected, it is resolved by this site. If only a potential deadlock cycle is detected, the information must be transmitted to the next site.

Both local and global deadlocks are resolved by aborting, rolling back and restarting one of the transactions involved in the deadlock cycle.

It should be noticed that this deadlock detection strategy has a low probability of being affected by a site failure. It also reduces the amount of internode traffic since only graphs containing potential deadlock cycles need to be transmitted.

3.2.3. System R* Recovery Strategy

System R* uses two variations of the two-phase commitment protocol: the presumed abort protocol and the presumed commit protocol.

In the presumed abort protocol, each site has to send a message to the coordinator indicating whether it is

aborting or committing a transaction, just the same as in the standard two-phase commitment protocol. The difference is that for an abort decision the coordinator only needs to broadcast an abort message to the other sites and write a global abort record in the log. It no longer needs to write this record on non-volatile storage, nor wait for an acknowledgment from the other sites. Only commit records have to be written into the log at non-volatile storage. This is so because at recovery time, if no information is found about a transaction, it is presumed it has been aborted.

In the presumed commit protocol, if at recovery time no information is found about a transaction, it is presumed that the transaction has been committed. It requires the coordinator to write into non-volatile storage the identifiers of all subtransactions which are part of the transaction. It the coordinator fails, then upon its recovery it must notify all other sites involved that the transaction must be aborted. This is to prevent the other sites from committing a transaction due to the lack of information at the coordinator as a result of its failure. It should be noticed that besides the identifiers of all subtransactions, only the abort messages need to be written into non-volatile storage.

System R* has the ability to select the protocol to be used for each transaction. The presumed commit protocol should be used for update transactions since it doesn't require acknowledgment records nor the recording of the global commit record into non-volatile storage. The presumed abort protocol should be used for read-only transactions since it doesn't require that the identifiers of all subtransactions be written into the coordinator's log at non-volatile storage.

3.3. Distributed INGRES

Distributed INGRES is the distributed version of the INGRES relational database system. It was developed at the University of California at Berkeley.

Distributed INGRES runs under the Unix operating system. For a distributed transaction, the INGRES database system at the site where the transaction originates must invoke the systems on all other sites involved in the transaction's execution. The collection of INGRES processes running at the site where the transaction originates is called Master INGRES. The cooperating processes running at the other sites involved in the execution of the distributed transaction are called Slave INGRES.

Distributed INGRES can run on a broadcast type of network such as Ethernet, or on a point-to-point type of network such as Arpanet.

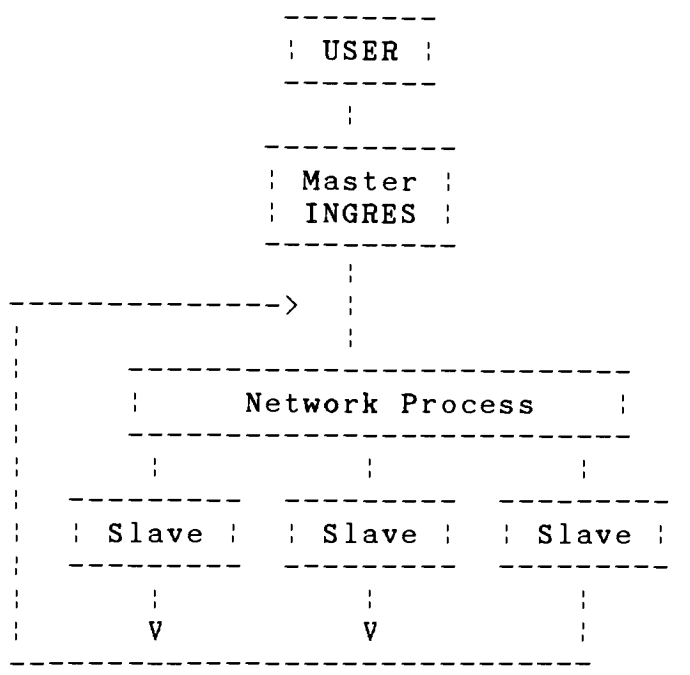

Figure 3.6

Distributed INGRES Architecture

3.3.1. Distributed INGRES Synchronization Strategy

The synchronization strategy used by Distributed INGRES is called two-phase locking. During the first phase a transaction acquires all its locks. No locks can be released during this phase. The transaction can only be executed when all necessary locks are obtained.

After a transaction obtains all necessary locks, it follows a primary copy update technique. Each site has a

local concurrency controller that receives from the Master INGRES the updates for those data items for which the site holds the primary copy. After the primary copy has been updated, an update list containing all changes to be made is sent to each site holding a copy of the data items. Lock tables are local to each site'.

However, if a transaction cannot obtain all necessary locks, it must release all locks already acquired and try again some time later. When a transaction releases a lock, it enters the second phase of the two-phase locking algorithm. During this second phase no further locks can be acquired. Locks should be released only by one of two reasons: a requested lock cannot be obtained (as explained above) or the transaction reached its commit point.

On the standard two-phase locking algorithm, transactions are forced to wait whenever a requested lock cannot be granted. The two-phase locking algorithm used by Distributed INGRES differs from the standard two-phase locking in that transactions release all the locks they have obtained, if a requested lock cannot be granted. This is done to avoid the possibility of deadlocks which can be caused by forcing transactions to wait.

3.3.2. Distributed INGRES Deadlock Strategy

Distributed INGRES uses a deadlock detection strategy. If a local deadlock is detected, the local concurrency controller aborts, rolls back and restarts one of the transactions involved. Then the Slave INGRES informs the Master INGRES of what happened and the Master INGRES informs all other slaves involved.

A distributed deadlock is detected and handled in the following way. When a concurrency controller detects a transaction waiting for another transaction to release a lock, it sends this information to the SNOOP. The SNOOP is the machine that acts as a centralized deadlock detector. The SNOOP will detect the deadlock by analyzing the global wait-for graph. It will decide how to resolve the deadlock and will send the necessary information to the sites involved.

3.3.3. Distributed INGRES Recovery Strategy

Distributed INGRES uses two types of algorithms which are similar to a two-phase commit: performance algorithms and reliability algorithms. Performance algorithms process updates with the minimum possible delay; but can generate database inconsistencies in the event of failures.

On the other hand, reliability algorithms guarantee no consistency problems, at the cost of a longer response time.

The set of algorithms is basically the same for both performance and reliability algorithms. However, reliability algorithms include some changes to guarantee no consistency problems, as will be shown later. There are three basic algorithms: 1) the master algorithm is run at the site where the transaction originates; 2) the slave algorithm is run under supervision of the master algorithm at the site that holds the primary copy of the data item to be updated; 3) the copy algorithm is run at every site holding a copy of the data item. In addition, there are three algorithms that deal directly with site crashes: 1) local recovery, 2) reconfigure, 3) slave promote. Local recovery is run to restore a site after a crash. Reconfigure investigates which sites are up and creates a new list of operational sites based on the information collected. Slave promote sees that slaves can commit or roll back the transaction being executed in the event of a master site failure.

3.3.3.1. Master Algorithm

The master algorithm will arrange for a received update to be processed at the site containing the primary copy of the data item to be updated.

Step 1: Reject the transaction if the network is not under normal operation.

Step 2: Find, in the list of operational sites, the primary site of the data item to be updated by the transaction. Reject the transaction if the item's primary site cannot be found in the list.

Step 3: Coordinate getting an update list to each site that holds a copy of the data item.

Step 4: Wait for a ready message from all the sites involved in the transaction's execution. If not all the sites involved respond, send an abort message to all of them. Queue the message for later delivery to the nonresponding sites. Ask for the reconfigure algorithm to be run at all operational sites. Nonoperational sites will run the reconfigure algorithm automatically during their recovery. End execution.

Step 5: Set the commit flag. This means the transaction will eventually be committed, even in the event of a failure. Send a commit message to all sites involved in

the execution of the transaction.

Step 6: Wait for a message from all sites involved, informing the update has been executed. If not all of them respond, queue the commit message and a message telling there is a potential trouble situation. These messages will be sent to the nonresponding sites when they recover. Ask for the reconfigure algorithm to be run at all operational sites. End execution.

Step 7: Send a message to the user process and to the SNOOP informing that processing is complete. The SNOOP will update its wait-for graph by using this information. End execution.

3.3.3.2. Slave Algorithm

The slave algorithm is in charge of updating the primary copy of the data item.

Step 1: Write into non-volatile storage all information necessary to commit the transaction. Prepare the update list.

Step 2: Send a ready message to the master.

Step 3: Wait for a commit or an abort message from the master.

Step 4: If an abort message is received, run the local recovery algorithm and end execution.

Step 5: If a commit message is received, set the local commit flag, commit the transaction, and send a message to the master informing the transaction has been executed to completion.

Step 6: Send an update list to each site holding a copy of the updated data item.

Step 7: Wait for a message, from all sites holding copies of the data item, informing the updates have been executed. If not all sites holding copies respond, queue the update list for later delivery and ask for the reconfigure algorithm to be run.

Step 8: End execution.

3.3.3.3. Copy Algorithm

The copy algorithm is in charge of updating all copies of the data item other than the primary copy.

Step 1: Wait for the update list.

Step 2: Perform the update. Send a message to the slave process informing the update has been executed.

Step 3: End execution.

3.3.3.4. Local Recovery Algorithm

The local recovery algorithm is run after a site crash or when the slave algorithm requests it to be run due to a transaction's abortion.

Step 1: Read all queued messages and perform their requested actions.

Step 2: If the local commit flag is set, commit the transaction. Otherwise, abort the transaction.

Step 3: If the algorithm was requested to run by the slave algorithm, send it a message informing it is done. Otherwise, ask for the reconfigure algorithm to be run at all operational sites.

Step 4: End execution.

3.3.3.5. Reconfigure Algorithm

The reconfigure algorithm creates the list of operational sites.

Step 1: Set a flag indicating the system is going through a reconfiguration. Ask all sites to complete processing any current transactions.

Step 2: Send to all other sites a message indicating this site is up.

Step 3: Wait for replies and create from them the list of operational sites.

Step 4: If this site is the lowest in the established ordering of all the sites, send the new list to all other sites and wait for them to agree or disagree. If disagreements or nonrespondents exist, send a reconfigure message and start all over. If all sites agree, set a flag indicating the system is under normal operation. Send a message indicating this to all sites and go to step 8.

Step 5: Wait for a list of operational sites from some other site.

Step 6: Compare the local list with the received list. Answer with either an agree or a disagree message.

Step 7: Wait for a message informing the system is resuming normal operation.

Step 8: If the SNOOP is not in the list of operational sites, a new SNOOP must be selected before resuming normal operation.

3.3.3.6. Slave Promote Algorithm

The slave promote algorithm is executed whenever a slave process cannot communicate with the master process. Its purpose is to allow the slave process to finish the execution of the transaction being processed.

Step 1: Ask for the reconfigure algorithm to be run.

Step 2: Wait for a message telling the system is under normal operation.

Step 3: Check the list of operational sites. If this site is not the lowest in the established ordering of sites, wait for a message asking if the commit flag is set; reply yes or no; and continue normal operation of the slave algorithm from Step 3.

Step 4: Send a message asking all other sites if they have their commit flags set. Notice that this step is executed only if this site is the lowest in the ordering; which means it now becomes the coordinator or Master.

Step 5: Wait for replies.

Step 6: If one or more sites are found to have their commit flags set, send a commit message to all sites involved. Finding some site with its commit flag set will mean the master has send a commit decision before failing. Wait for a message informing they are done and send such a message to the user process and to the SNOOP. End execution. If not all sites answer to the commit message, queue the commit message along with a potential trouble message; ask for the reconfigure algorithm to be run; and end execution.

Step 7: If no site is found to have its commit flag set, send an abort message to all sites involved. Queue this message for later delivery to nonoperational sites. Inform the user process of the decision to abort the transaction. End execution.

3.3.3.7. Reliable Algorithms

The set of reliability algorithms include the master, slave, local recovery, reconfigure, and slave promote algorithms but with the following modifications. Instead of first updating the primary copy of a data item and then having a different algorithm update the copies, the update

will be sent to all copies simultaneously. There will be no primary copy. All copies will use the slave algorithm. The slave algorithm will not include steps 6 and 7; that is, the steps that used to deal with the copies. Last, if the algorithms are to be K resilient, there should be at least K copies of each data item. Also, whenever there is the need to queue a message for later delivery, it should be queued at K sites.

4. Conclusion

4.1. Systems' Overview

This dissertation has presented the techniques that have been proposed so far for solving the problems of synchronization, deadlock and recovery in distributed databases. Three prototype distributed databases, SDD-1, System R* and Distributed INGRES, have been discussed in terms of the strategies they use to deal with these problems.

The final issue to be presented here is a discussion on how these systems face the following objectives: ease of use, reduced processing time, minimal internode communication, high degree of concurrency, high level of reliability, low storage cost, and site autonomy.

4.1.1. Ease of Use

The three systems provide ease of use by supporting location transparency. That is, they present a single site image to the user, which makes him unaware of the data distribution. In addition, System R* provides a language (SQL) in which identical requests can be used for

accessing both remote and local data.

4.1.2. Reduced Processing Time

SDD-1 has several techniques oriented toward achieving reduced processing time. First, it uses conservative timestamping as a synchronization technique because it avoids the need for restarting transactions. This is achieved by buffering a transaction until there is no other transaction having a smaller timestamp. Second, it uses the conflict graph analysis technique to preanalyze transactions. This analysis is part of the synchronization strategy used by SDD-1. It is performed at the system's design time. This reduces the processing time because it doesn't have to be done more than once. Furthermore, transactions are grouped into transaction classes defined by the transactions' read-set and write-set. The conflict graph analysis is performed using the transaction classes instead of having to perform the analysis for every known transaction. Conflict graph analysis has the purpose of selecting the least expensive synchronization protocol required, in terms of processing time and amount of internode communication. Third, SDD-1 uses a deadlock prevention strategy aimed at avoiding the processing time required for detecting deadlocks. On the other hand, the

recovery strategy selected by SDD-1 involves more processing time than the strategies used by System R* and Distributed INGRES. This is so because SDD-1's recovery strategy was oriented toward achieving a high level of reliability. This will be further discussed below.

The synchronization and deadlock strategies used by System R* are not particularly oriented toward achieving reduced processing time. However, System R* uses for its recovery strategy a variation of the two-phase commit protocol that reduces processing time through the use of defaults. It uses a presumed abort protocol that assumes a transaction has been aborted whenever there is no information available at recovery time. Using this protocol, only commit records have to be written into the log in non-volatile storage. In the presumed commit protocol, whenever the recovery process doesn't have any information whether a transaction was committed or aborted, it is assumed the transaction was committed. When using the presumed commit protocol, only abort records need to be written into the log in non-volatile storage.

The synchronization and deadlock strategies used by Distributed INGRES are not oriented toward achieving reduced processing time. However, the two-phase commit protocol variation used by Distributed INGRES provides the capability of selecting between a protocol that offers

reduced processing time and a protocol that requires more processing time, but guarantees no consistency problems will occur.

4.1.3. Minimal Internode Communication

In SDD-1, the use of timestamps remove the communication overhead caused by locking techniques such as those used by System R* and Distributed INGRES. SDD-1 has a series of synchronization protocols which vary on the level of synchronization provided. For transactions requiring little or no synchronization, the amount of internode communication is minimal. Also, the deadlock prevention strategy used by SDD-1 is aimed at eliminating the message overhead caused by deadlock detection techniques. On the other hand, the four-phase commit protocol used for recovery requires a large amount of internode communication.

The two-phase locking synchronization technique used in System R* requires a large amount of internode communication. Although System R* does not completedly eliminate the message communication overhead, it uses a distributed deadlock detection technique that requires less internode communication than that required by the centralized deadlock detection technique used by Distributed INGRES.

Only information on potential deadlocks need to be transmitted. The two-phase commit protocol used for recovery requires a large amount of internode communication. However, through the use of the presumed abort and presumed commit protocols, fewer messages need to be exchanged than on the standard two-phase commit protocol.

The two-phase locking synchronization technique used by Distributed INGRES requires a large amount of internode communication. Although the centralized deadlock detection strategy used by this system also requires a large amount of internode communication, deadlock information needs only to be transmitted in one direction, to the central deadlock detector. The two-phase commit protocol used for recovery also requires a large amount of internode communication.

4.1.4. High Degree of Concurrency

The preanalysis of transaction conflicts performed in SDD-1 allows a high degree of concurrency by eliminating run-time synchronization for those transactions which do not really need to be synchronized. On the other hand, the timestamping technique used limits the amount of concurrency by imposing a specific ordering of transaction execution (the timestamp order).

The two-phase locking synchronization technique used by System R* and Distributed INGRES allows an even higher degree of concurrency than that allowed by the synchronization techniques used by SDD-1 because it doesn't force a specific ordering of transaction execution.

4.1.5. High Level of Reliability

In the synchronization strategy used by SDD-1 there is no centralized function. Failure of one site only affects transactions using that site. SDD-1 also supports data replication. Copies of data items at other sites may be used when the site holding a data item is unavailable due to a failure. SDD-1 uses an extended communication facility known as the Reliable Network (RELNET) which provides reliable communication facilities. This facilities include: queues for buffering messages sent to failed sites, and a four-phase commit protocol. The four-phase commit protocol uses an ordered set of backup processes that can substitute the coordinator in the event of a coordinator's failure. However, RELNET is not resilient to network partitions.

In System R*, all synchronization, deadlock and recovery strategies are decentralized. In this way they provide a high level of reliability because they are not

subject to failure of a central component.

Distributed INGRES allows data replication for reliability purposes. If a site fails, transactions are able to continue their execution as long as they do not involve a data item whose only copy resides at the failed site. The system's synchronization and recovery strategies are decentralized. However, Distributed INGRES uses a centralized deadlock detection strategy that has a very low level of reliability because failure of the central deadlock detector leaves the entire system without any deadlock detection strategy.

4.1.6. Low Storage Cost

SDD-1 supports data replication. Although replication provides reliability, it also results in a high storage cost. The timestamp synchronization technique used by SDD-1 also results in a high storage cost because each copy of a data item stores an associated timestamp which indicates the time when the item was last updated.

Both System R* and Distributed INGRES use a two-phase locking synchronization strategy which provides low storage cost, since no timestamps need to be stored.

Distributed INGRES, like SDD-1, supports data replication. It therefore, has a high storage cost associated with that.

4.1.7. Site Autonomy

All three systems provide some degree of site autonomy because transactions can be controlled from a single site. However, some loss of site autonomy occurs when using techniques such as the four-phase commit used by SDD-1 and the two-phase commit used by Distributed INGRES and System R*. They relinquish their site autonomy when they agree that the coordinator is going to make the final decision whether to commit or abort a transaction.

One of the major objectives in the design of System R* was site autonomy. Therefore, System R* was designed as a set of cooperating autonomous databases. All techniques used are decentralized. Each site has control over its own data. In this way, sites are able to perform operations on local data even when they cannot communicate with other sites in the network.

The techniques used by SDD-1, System R* and Distributed INGRES to deal with the problems of synchronization, deadlock and recovery are summarized in Table 4.1. The

objectives met by these three systems are summarized in Table 4.2.

Table 4.1

	Synchronization	Deadlock	Recovery
SDD-1	Trans. classes Conflict graph analysis Protocols Conservative timestamping	Prevention thru timestamps and timeouts	4-phase commit RELNET
R*	2-phase locking	Distributed deadlock detection	2-phase commit with presumed abort and presumed commit
INGRES	2-phase locking	Centralized deadlock detection	2-phase commit with performance and reliability algorithms

Table 4.2

Objectives		SDD-1	System R*	Dist. INGRES
Ease of Use		yes	yes	yes
Reduced Processing Time	S	yes		
	D	yes		
	R		yes	yes
Minimal Internode Communication	S	yes		
	D	yes	yes	
	R			
High Degree of Concurrency	S	yes	yes	yes
	D			
	R			
High Level of Reliability	S	yes	yes	yes
	D		yes	
	R	yes	yes	yes
Low Storage Cost			yes	
Site Autonomy			yes	

NOTE: S stands for synchronization
D stands for deadlock
R stands for recovery

4.2. Future Trends

Distributed databases seem to fill the needs of distributed organizations. These include improved data availability and easier adjustment to growth. For this reason, the demand for distributed databases is likely to increase. This motivates research and the desire to improve the available systems.

SDD-1 is an attractive distributed database because it has carefully planned synchronization and recovery techniques that provide a high degree of reliability. However, it only works with a local database manager called Datacomputer [BRAD83]. This could represent a major drawback in SDD-1 becoming a widely used system, because organizations already having a local database manager will not be motivated to incur the high cost of changing to another local database manager.

System R* has a higher probability of becoming commercially accepted than SDD-1, because it is an extension of the already commercially available System R (SQL/DS and DB2). Another aspect that gives System R* the opportunity of becoming a widely accepted system is that it has been designed to achieve site autonomy. This means that each site has a full-function database system capable of communicating with the other sites through messages exchanged

using an intersystem communication facility.

The probability of Distributed INGRES becoming commercially accepted is difficult to predict. It uses basically the same attractive synchronization and recovery techniques as System R*, but has the advantage of supporting replicated data. On the other hand, it doesn't implement the distributed database as a set of cooperating autonomous sites, but as a system whose data is spread over several sites that might be geographically distant from one another.

Some trends can be identified in the discussion of SDD-1, System R* and Distributed INGRES. It is likely that most distributed databases will continue to support a relational type of database because relations are easier to distribute than hierarchical and network structures. Two-phase locking and timestamping seem to be the most accepted synchronization strategies. When dealing with the deadlock problem, detection seems to be the most practical approach over prevention and avoidance. It appears that a distributed transaction recovery facility such as two-phase commit will be incorporated in most systems to be developed. However, there is a high probability that new techniques will be developed.

Another trend seems to be for distributed databases to allow having different database managers at different sites. This can be achieved by having an interface to a global database manager. Research is being done in this area.

The study presented here of the existing synchronization, deadlock and recovery strategies is aimed at providing a foundation for further investigations and for the development of new approaches that can take advantages of their good features and minimize the impact of the undesirable ones.

5. Appendix I: Glossary

atomic action - computation, that although composed of primitive computational steps, cannot be decomposed and whose results are not revealed until the computation is completed, that is, failures do not allow intermediate states of objects to result from the partial execution of an atomic action.

CICS - an IBM's system whose main function is to control the execution of online applications requested from terminals by the users.

commit - permanently record in the database all changes up to that point.

concurrency control - coordinated control of access to data for the purpose of maintaining database integrity and consistency.

conflict graph - undirected graph that summarizes potential conflicts between transactions; its nodes represent transaction class read-sets and write-sets, and its edges represent conflicts.

consistency - property where the value of the copies in the database are in agreement with each other.

data module - process that is part of the database management system and is in charge of the data manipulation.

database - centralized collection of data.

database management system - subsystem used to access the database.

distributed database - collection of data stored at different locations of a distributed computer system.

deadlock - condition on which a set of transactions can never be granted all the resources (e.g. data items) they need in order to reach completion.

exclusive lock - kind of lock by which no other transaction can acquire a lock of any type on the data item holding it.

integrity - condition that ensures the data in the database has three qualities: accuracy, correctness, and validity.

lock - mechanism that prevents a data item from being accessed by certain transactions.

network partition - situation where a network is divided into two or more parts having no communication path available between them.

non-volatile storage - kind of storage that guarantees no information will be lost in the event of a failure.

preemptive - that can be obtained, e.g. a data item that can be obtained by a transaction even if it is being used by another transaction.

querie - read-only transaction.

read-set - set of items a transaction uses as input to compute the write-set.

recovery - to resume correct operation after a failure or error.

reliable network - network communication facility that provides a global network clock and guarantees an ordered delivery of messages, even in the event of site failures.

resiliency - property of having the capability to survive failures.

rollback - operation that has the effect of undoing all the updates made by an unsuccessful transaction.

serial execution - each transaction is executed to completion before the next one begins.

shared lock - kind of lock that allows a distinct transaction to acquire a shared lock, but not an exclusive lock, on the data item already holding a shared lock.

synchronization - refers to the solution of two related problems: 1) control of the joint activity of cooperating sequential processes, 2) serialization of concurrent access to objects shared by multiple processes.

timestamp - value used to serialize transaction execution; it is formed by reading the local clock time and appending the site identifier to the low order bit.

transaction - a sequence of operation on one or more database items that together perform a logical unit of work. It transforms a current consistent state of the system into a new consistent state. E.g. A deposit to a bank account is a transaction that involves the reading of the actual account balance, and the writing of the new balance (actual balance plus deposit).

transaction module - process that is part of the database management system and is concerned with generating transaction identifiers, maintaining the local transaction log, requesting the needed data items, sending sub-transactions to remote sites, coordinating multi-site transactions, and coordinating recovery.

wait-for graph - graph used in the detection of deadlocks; each node represents a transaction, directed edges represent waiting situations, and cycles represent deadlock situations.

write-set - set of items whose values are modified by a transaction.

6. Appendix II: Personnal Qualifications

In 1979 I received my Bachelor's Degree in Business Administration with a major in Computer Information Systems and a major in Statistics from the University of Puerto Rico. I worked for two years as a Programmer Analyst and the next two years as Assistant Database Administrator on an IMS Database. During the last year I have completed the courses required by the Rochester Institute of Technology Master Degree Program in Computer Science. I selected the following courses for my concentration: Database Systems I, Data Communication and Networks I, Data Communication and Networks II. My main research interests are in the field of Database Systems, in particular Distributed Databases.

B I B L I O G R A P H Y

[AFRA82] A Framework for Distributed Database Systems: Distribution Alternatives and Generic Architectures. New York: Association for Computing Machinery, 1982.

[BADA80] Badal, D. "The Analysis of the Effects of Concurrency Control on Distributed Database System Performance," Sixth International Conference on Very Large Data Bases, (1980), 376-383.

[BERN81] Bernstein, Philip A. and Goodman, Nathan. "Concurrency Control in Distributed Database Systems," ACM Computer Surveys, XIII (June, 1981), 185-222.

[BERN78] Bernstein, Philip A., Rothnie, J.B., Goodman, Nathan and Papadimitriou, C.A. "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)," IEEE Transactions on Software Engineering, IV (No. 3), (1978), 154-168.

[BERN80a] Bernstein, Philip A. and Goodman, Nathan. "Approaches to Concurrency Control in Distributed Data Base Systems," in Auerbach, Isaac L. (ed.). The Auerbach Annual 1980 Best Computer Papers. New York: Elsevier North Holland, Inc., 1980.

[BERN80b] Bernstein, Philip A. and Goodman, Nathan. "Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems," Sixth International Conference on Very Large Data Bases, (1980), 285-300.

[BERN80c] Bernstein, Philip A. and Shipman, D. W. "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)," ACM Transactions on Database Systems, V (March, 1980), 52-68.

[BERN80d] Bernstein, Philip A., Shipman, D.W. and Rothnie, J.B. "Concurrency Control in a System for Distributed Databases

(SDD-1)," ACM Transactions on Database Systems, V (March, 1980), 18-51.

[BRAD83] Bradley, James. Introduction to Data Base Management in Business. New York: CBS College Publishing, 1983.

[CERI84] Ceri, Stefano and Pelagatti, Giuseppe. Distributed Databases Principles and Systems. McGraw-Hill, Inc., 1984.

[CHAM80] Champine, George A. Distributed Computer Systems Impact on Management, Design, and Analysis. Amsterdam: North Holland Publishing Co., 1980.

[CHAN83] Chandy, K.M., Haas, L.M. and Misra, J. "Distributed Deadlock Detection," ACM Transactions on Computer Systems, I (1983), 144-156.

[CHUW81] Chu, Wesley W. and Ohlmacher, G. "Avoiding Deadlock in Distributed Data Bases," in Liebowitz, Burt and Carson, John (Eds.) Tutorial: Distributed Processing, IEEE Computer Society Press, 1981.

[DANI82] Daniels, Dean et al. "An Introduction to Distributed Query Compilation in R*," Research Report RJ3497 (41354), IBM Reseach Lab., San Jose, CA, (June, 1982), 1-21.

[DATE83] Date, C.J. An Introduction to Database Systems. Reading: Addison-Wesley Publishing Co., 1983.

[DEIT84] Deitel, Harvey M. An Introduction to Operating Systems. Reading: Addison-Wesley Publishing Co., 1984.

[GARC81] Garcia-Molina, Hector. Performance of Update Algorithms for Replicated Data. Ann Arbor: UMI Research Press, 1981.

[GARD80] Gardarin, George. "Integrity, Consistency, Concurrency, Reliability in Distributed Database Management Systems," in Delobel, C. and Litwin, W. (Eds.). Distributed Data Bases. Amsterdam: North Holland Publishing Co., 1980.

[GLIG80] Gligor, V.D. and Shattuck, S.H. "On Deadlock Detection in Distributed Systems," IEEE Transactions on Software Engineering, IV (Sept., 1980), 435-439.

[HAAS82] Haas, L.M. et al. "R*: A Research Project on Distributed Relational DBMS," Research Report, RJ3653 (42509), IBM Reseach Lab., San Jose, CA, (Oct., 1982), 1-5.

[HAMM80] Hammer, M. and Shipman, David. "Reliability Mechanisms for SDD-1: A System for Distributed Databases," ACM Transactions on Database Systems, V (December, 1980), 431-466.

[KATZ78] Katzan, Harry Jr. An Introduction to Distributed Data Processing. New York: Petrocelli Books, Inc., 1978.

[KOHL81] Kohler, Walter H. "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," ACM Computing Surveys, XIII (June, 1981), 149-183.

[LELA80] Le Lann, G. "Consistency, Synchronization and Concurrency Control," in Draffan, I. W. and Poole, F. Distributed Data Bases. Cambridge: Cambridge University Press, 1980.

[LIND79] Lindsay, Bruce G. et al. "Notes on Distributed Databases," Research Report, RJ2571 (33471), IBM Research Lab., San Jose, CA, (July, 1979), 1-57.

[LIND80a] Lindsay, Bruce G. "Single and Multi-site Recovery Facilities," in Draffan, I. W. and Poole, F. Distributed Data Bases. Cambridge: Cambridge University Press, 1980.

[LIND80b] Lindsay, Bruce G. "Object Naming and Catalog Management for a Distributed Database Manager," Research Report, RJ2914 (36689), IBM Research Lab., San Jose, CA, (August, 1980), 1-16.

[LIND84] Lindsay, Bruce G. et al. "Computation and Communication in R*: A Distributed

Database Manager," ACM Transactions on Computer Systems, II (Feb. 1984), 24-32.

[MCLE81] McLean, G. Jr. "Comments on SDD-1 Concurrency Control Mechanisms," ACM Transactions on Database Systems, IV (June, 1981), 347-350.

[MENA79] Menasce, D.A. and Muntz, R.R. "Locking and Deadlock Detection in Distributed Databases," IEEE Transactions on Software Engineering, V (1979), 195-202.

[MUNZ80] Munz, Rudolf. "Realization, Synchronization and Restart of Update Transactions in a Distributed Database System," in Delobel, C. and Litwin, W. (Eds.). Distributed Data Bases. Amsterdam: North Holland Publishing Co., 1980.

[NEUH77] Neuhold, E. and Stonebraker, Michael. "A Distributed Version of INGRES," Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, (1977), 1-28.

[OBER82] Obermarck, R. "Distributed Deadlock Detection Algorithm," ACM Transactions on Database Systems, VII (June, 1982), 187-208.

[ROSE78] Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M. "System Level Concurrency Control for Distributed Database Systems," ACM Transactions on Database Systems, III (1978), 178-198.

[ROTH80] Rothnie, J.B. "Introduction to a System for Distributed Databases (SDD-1)," ACM Transactions on Database Systems, V (March, 1980), 1-17.

[SCHL80] Schlageter, G. and Dadam, P. "Reconstruction of Consistent Global States in Distributed Databases," in Delobel, C. and Litwin, W. (Eds.). Distributed Data Bases. Amsterdam: North Holland Publishing Co., 1980.

[SHAP79] Shapiro, Robert and Millstein, Robert. "Failure Recovery in a Distributed Data Base System," in Chu, Wesley W. and Chen, Peter P. (Eds.) Tutorial: Centralized and Distributed Data Base Systems, New York: The Institute of

Electrical and Electronics Engineers,Inc., 1979.

[STON79] Stonebraker, Michael R. "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES," IEEE Transactions on Software Engineering, V (May, 1979), 188-194.

[STON80] Stonebraker, Michael R. "Homogeneous Distributed Data Base Systems," in Draffan, I.W. and Poole, F. Distributed Data Bases. Cambridge: Cambridge University Press, 1980.

[TANE81] Tanenbaum, Andrew. Computer Networks. New Jersey: Prentice-Hall, Inc., 1981.

[THOM79] Thomas, R. H. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Data Bases," ACM Transactions on Database Systems, IV (June, 1979), 180-209.

[THOM80] Thomas, R. H. "Process Structure Alternatives Toward a Distributed INGRES," in Delobel, C. and Litwin, W. (Eds.). Distributed Data Bases. Amsterdam: North Holland Publishing Co., 1980.

[WAHB81] Wah, Benjamin W. Data Management on Distributed Databases. Ann Arbor: UMI Research Press, 1981.

[WALT80a] Walter, Bernd. "Concepts for a Robust Distributed Data Base System," in Delobel, C. and Litwin, W. (Eds.). Distributed Data Bases. Amsterdam: North Holland Publishing Co., 1980.

[WALT80b] Walter, Bernd. "Strategies for Handling Transactions in Distributed Data Base Systems During Recovery," Sixth International Conference on Very Large Data Bases, (1980), 384-389.

[WILM83] Wilms, P.F., Linsay, B.G. and Selinger, P. "I Wish I Were Over There: Distributed Execution Protocols for Data Definition in R*," ACM SIGMOD International Conference on Management of Data, XIII (May, 1983), 238-242.