SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

**SCHOOL OF COMPUTING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# UNIT – IV – DISTRIBUTED DATABASE – SCS1613

# UNIT IV

# RELIABILITY AND PROTECTION

***Reliability- Basic Concepts- Reliability and concurrency Control- Determining a Consistent View of the NetworkDetection and Resolution of Inconsistency- Checkpoints and Cold Restart- Distributed Database AdministrationCatalog Management in Distributed Databases-Authorization and Protection***

## RELIABILITY

**Reliability** is defined as a measure of the success with which the system conforms to some authoritative specification of its behavior. When the behavior deviates from that which is specified for it, this is called as **Failure**. The reliability of the system is inversely related to the frequency of failures.

The reliability of a system can be measured in several ways, which are based on the incidence of failures. Measures include Mean Time Between Failure(MTBF), Mean Time To Repair(MTTR), and availability, defined as the fraction of the time that the system meets its specification. MTBF is the amount of time between system failures in a network. MTTR is the amount of time system takes to repair the failed systems.

## BASIC CONCEPTS

In a database system application, the highest level specification is application-dependent. It is convenient to split the reliability problem into two separate parts, an application-dependent part and an application-independent part.

We emphasize two aspects of reliability: correctness and availability. It is important not only that a system behave correctly, i.e., in accordance with the specification, but also that it be available when necessary.

In some applications, like banking applications, correctness is an absolute requriment, and errors which may corrupt the consistency of the database cannot be tolerated. Other applications may tolerate the risk of inconsistencies in order to achieve a greater availability.

When a communication network fails the following problems may arise,

1. Commitment of transactions

2. Multiple copies of data and robusteness of concurrency control
3. Determining the state of the network
4. Detection and resolution of inconsistencies.
5. Checkpoints and cold restart.

## NONBLOCKING COMMITMENT PROTOCOLS

A Commitment protocol is called **blocking**if the occurrence of some kinds of failure forces some of the participating sites to wait until the failure is repaired before terminating the transaction. A transaction that cannot be terminated at a site is called **pending** at this site. State diagrams are used for describing the evolution of the coordinator and participants during the execution of a protocol.

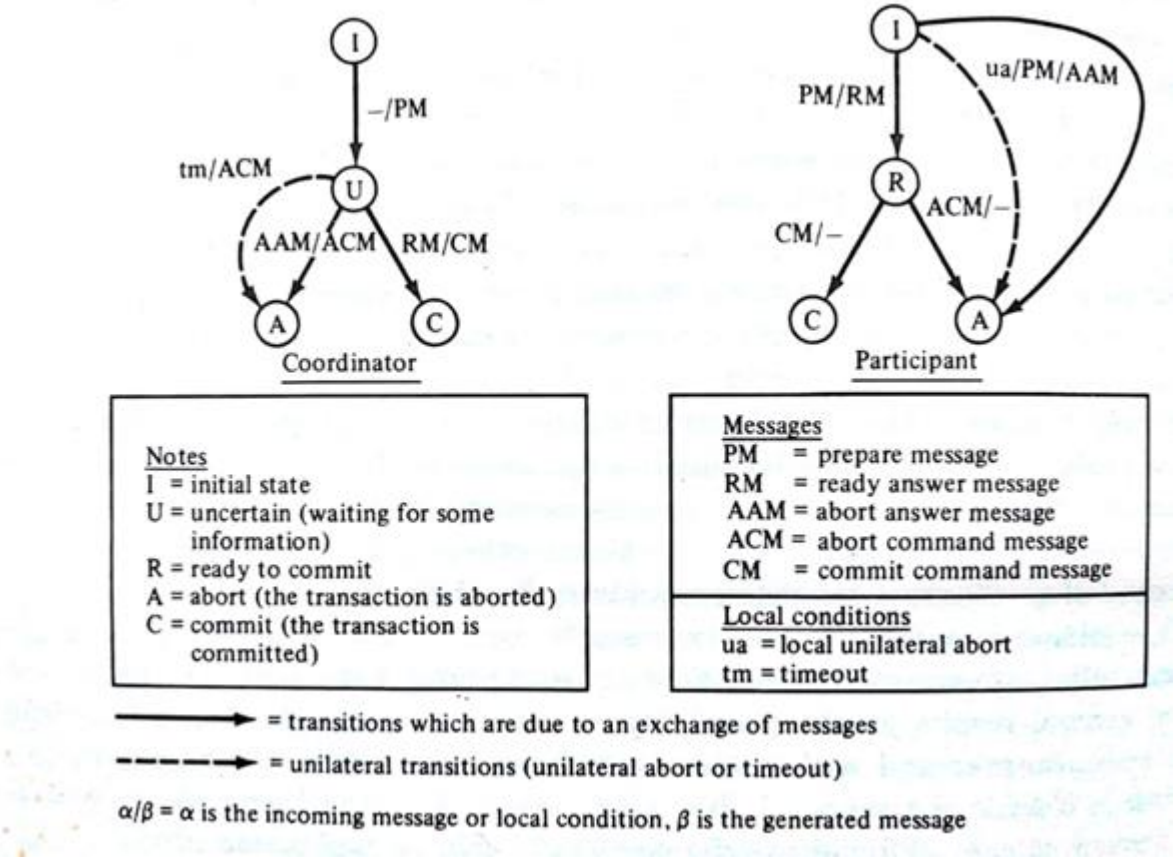



Figure 4.1 two state diagrams for the 2-phase-commitment protocol

The above figure shows the two state diagrams for the 2-phase-commitment protocol without the ACK messages. For each transition, an input message and an output message are indicated. A transition occurs when an input message arrives and causes the output message to be sent.

State information must be recorded into stable storage for recovery purposes. This helps in writing appropriate records in the logs.

Consider a transition from state X to state Y with input I and output O. The following behavior is assumed:

1. The input message I is received.

2. The new state Y is recorded on stable storage.
3. The output message O is sent.

If the site fails between the first and the second event, the state remains X, and the input message is lost. If the site fails between the second and third event, then the site reaches state Y, but the output message is not sent.

## 1. NONBLOCKING COMMITMENT PROTOCLS WITH SITE FAILURE

The termination protocol for the 2-phase-commitment protocol must allow the transactions to be terminated at all operational sites when a failure of the coordinator site occurs. This is possible in the following two cases:

1. At least one of the participants has received the command. In this case, the other participants can be told by this participant of the outcome of the transactions and can terminate it.
2. None of the participants has received the command, and only the coordinator site has crashed, so that all participants are operational. In this case, the participants can elect a new coordinator and resume the protocol.

In above cases, the transactions can be correctly terminated at all operational sites. Termination is impossible when no operational participants has received the command and at least one participant failed, because the operational participants cannot know the failed participant has done and cannot take an independent decision. So, if a coordinator fails termination is impossible.

This problem can be eliminated by modifying the 2-phase-commitment protocol in the 3-phase-commitment protocol.

### The 3-phase-commitment protocol

In this protocol, the participants do not directly commit the transactions during the second phase of commitment, instead they reach in this phase a new prepared-to-commit(PC) state. So an additional third phase is required for actually committing the transactions.

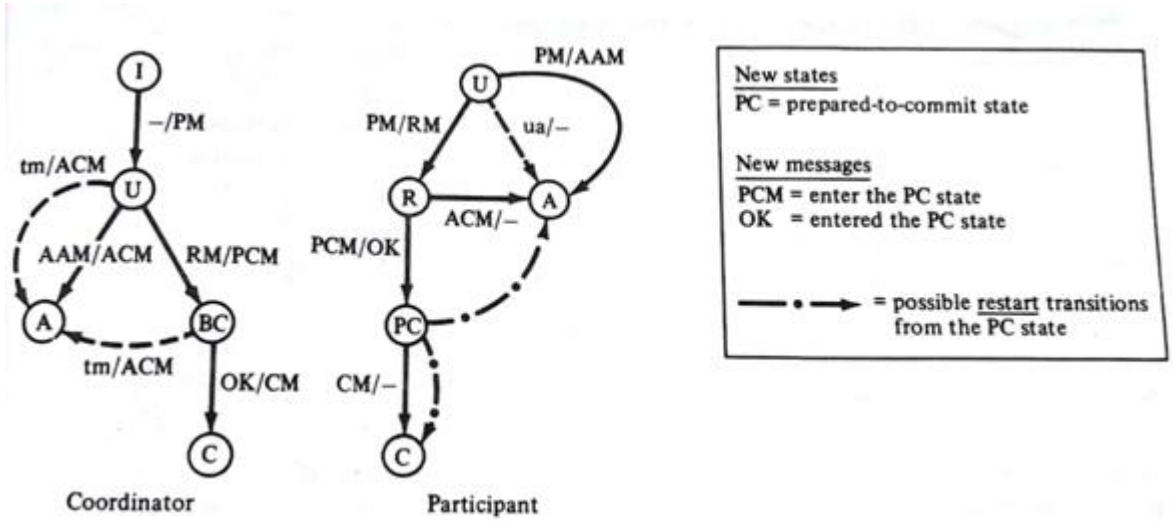



Figure 4.2 two state diagrams for the 3-phase-commitment protocol

This protocol eliminates the blocking problem of the 2-phase-commitment protocol because,

1. If one of the operational participants has received the command and the command was ABORT, then the operational participant can abort the transactions. The failed participant will abort the transaction at restart if it has not done it already.
2. If one of the operational participants has received the command and the command was ENTER-PREPARED-STATE, then all the operational participants can commit the transactions, terminating the second phase if necessary m performing the third phase.
3. If none of the operational participants has received the ENTER-PREPARED-STATE command, 2-phase-commitment protocol cannot be terminated. But with our new protocol, the operational participants can abort the transactions, because the failed participants has not committed. The failed transactions therefore abort the transactions at restart.

This new protocol requires three phases for committing a transaction and two phases for aborting it.

**Termination protocol for 3-phase-commitment**

"If at least one operational participant has not entered the prepared-to-commit state, then the transactions can be aborted. If at least one operational participant has entered the prepared-to-commit state, then the transactions can be safely committed."

Since the above two condition are not mutually exclusive, in several cases the termination protocol can decide whether to commit or abort the transactions. The protocol which always commits the transactions when both cases are possible is called progressive.

The simplest termination protocol is the centralized, nonprogressive protocol. First a new coordinator is elected by the operational participants. Then the new coordinator behaves as follows:

1. If the new coordinator is in the prepared-to-commit state, it issues to all operational participants the command to enter also in this state. When it has received all the OK messages, it issues the COMMIT command.
2. If the new coordinator is in commit state, i.e. it has committed the transactions, it issues the COMMIT command to all participants.
3. If the new coordinator is in the abort state, it issues the ABORT command to all participants.
4. Otherwise, the new coordinator orders all participants to go back to a state previous to the prepared-to-commit and after it has received all the acknowledgements, it issues the ABORT command.

## 2. COMMITMENT PROTOCOLS AND NETWORK PARTITIONS

### Existence of nonblocking protocols for partitions

The main problem of the existence of nonblocking protocols is, some protocol allows independent recovery in case of site failures.

The protocol we design must work as the following example. Suppose that we can build a protocol such that if one site, say site2, fails, then

1. The other site, site1, terminates the transactions.
2. Site2 at restart terminates the transactions correctly without requiring any additional information from site1.

So we make 4 propositions for the nonblocking commitment protocol, they are,

1. Independent recovery protocols exist only for single-site failures; however there exists no independent recovery protocol which is resilient to multiple-site failures.
2. There exists no nonblocking protocol that is resilient to a network partition if messages are lost when the partition occurs.
3. There exist nonblocking protocols which are resilient to a single network partition if all undeliverable messages are returned to the sender.
4. There exists no nonblocking protocol which is resilient to a multiple partition.

**Protocols which can deal with partitions**

It is convenient to allow the termination of the transactions by at least one group of sites, possible the largest group so that blocking is minimized. But it is not possible to determine the largest group, because it does not know the size of the other groups.

There are two approaches to this problem, the primary site approach and the majority approach.

In primary site approach, a site is designated as the primary site and the group that contains it is allowed to terminate the transactions.

In majority approach, only the group which contains a majority of ites can terminate the transactions. Here it is possible that no single group reaches a majority, in this case, all groups are blocked.

### A. Primary Site Approach

If the 2-phase-commitment protocol is used together with a primary site approach, then it is possible to terminate all the transactions of the group of the primary site(the primary group),if and only if the coordinators of all pending transactions belong to this group. This can be achieved by assigning to the primary site the coordinator function for all transactions.

This approach is inefficient in most types of networks and it is very vulnerable to primary site failure. To avoid this condition we can use 3-phase-commitment protocol can be used in primary group.

### B. Majority Approach and Quorum-Based Protocols

The majority approach avoids the disadvantages of the primary site approach. The basic idea is that a majority of sites must agree on the abort or commit of a transaction before the

transaction is aborted or committed. A majority approach requires a specialized commitment protocol. It cannot be applied with the standard 2-phase-commitment.

A straightforward generalization of the basic majority approach consists of assigning different weights to the sites. The protocol which use a weighted majority are called **quorum-based protocols**. The weights which are assigned to the sites are usually called **votes**, since they are used when a site "votes" on the commit or abort of a transaction.

The basic rules of a quorum-based protocol are:

1. Each site I has associated with it a number of votes $V_i$, $V_i$being a positive integer.
2. Let V indicate the sum of the votes of all sites of the network.
3. A transaction must collect a commit quorum $V_c$ before committing.
4. A transaction must collect an abortquorum $V_a$ before aborting.
5. $V_a + V_c > V$.

Rule 5 ensures that a transaction is either committed or aborted by implementing the basic majority idea. In practice, the choice $V_a + V_c = V + 1$ is the most convenient one.

A commitment protocol which implements this rule must guarantee that at one time a number of sites such that the sum of their votes is greater than $V_c$ agree to commit. It means these sites have entered a prepared-to-commit state. Therefore a quorum based commitment protocol can be obtained from the 3-phase-commitment protocol implementing the quorum requirement.

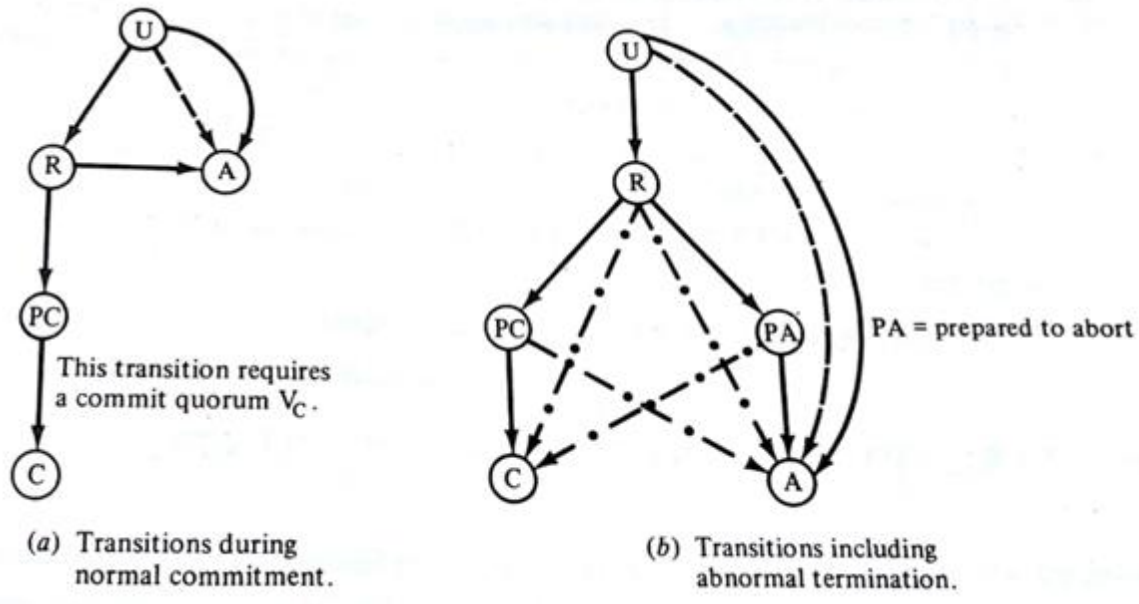



Figure 4.3 Quorum based 3 phase commitment protocol

Termination and restart are more complex in this protocol. So once a site has participated in building a commit (abort) quorum, it cannot participate in an abort (commit) quorum. Since a site cab fail after participating in building a quorum, its participation must be recorded in stable storage.

A centralized termination protocol for the quorum-based 3-phase-commitment has the following structure.

1. A new coordinator is elected.
2. The coordinator collects state information and acts according to the following rules:
   a. If at least one site has committed (aborted), send a COMMIT (ABORT) command to the other sites.
   b. If the number of votes of sites that reached the prepare-to-commit state is greater than or equal to $V_c$ , send a COMMIT command.
   c. If the number of votes of sites in prepare to abort state reaches the abort quorum, send an ABORT command.
   d. If the number of votes of sites that reached the prepare-to-commit state plus the number of votes of uncertain sites is greater than or equal to $V_c$, send a PREPARE-TO-COMMIT command to uncertain sites, and wait for condition 2b to occur.
   e. If the number of votes of sites that reached the prepare-to-abort plus the number of votes of uncertain sites is greater than or equal to $V_a$, send a PREPARE-TO-ABORT command to uncertain sites, and wait for condition 2c to occur.
   f. Otherwise, wait for the repair of some failure.

## RELIABILITY AND CONCURRENCY CONTROL

The problem arises when a failure happens is addressed here. We have to maximize the number of transactions which are executed during this failure by the operational part of the system.

Consider a transaction T having read-set RS(T) and write-set WS(T) and suppose that we want to run Talone, so that no concurrency control is needed. In order to run T it is necessary that at least one copy of each data item x belonging to RS(T) be available. If this elementary necessary condition is not satisfied, T cannot be executed, because it lacks input data. The availability of the data items of the write-set of T is not strictly required if we run T alone during a failure, because a list of deferred updates can be produced which will be applied to the database when the failure is repaired. Deferred updates can be implemented using “spooler” method.

The availability of a system which allows only one transaction to be run during failure is not satisfactory; therefore, concurrency control must be taken in account. The strongest limitations on the execution of transactions in the presence of failures are due to the need for concurrency control.

### A. NONREDUNDANT DATABASES

If the database is nonredundant, then it is very simple to determine which transactions can be executed. Let us consider 2-phase-locking is used for concurrency control. A transaction tries to lock all data items of its readand write-sets before commitment. As there is only one copy of some data item, this copy is either available or not. If the unique copy of some data item of the read or write-set is not available, the transaction cannot commit and must therefore be aborted.

If we assume that only site crashes occur but no partitions, then the availability of the items which belong only one to the write-set is not required, and it is possible to spool the update messages for these items. All transactions which have their read-set available executed completely, including commitment; but the updates affecting sites which are down are stored at spooler sites. When recovery happens, the restart procedures of the failed sites will receive this list of deferred updates and execute them. We consider a crashed site as exclusively locked for the transaction.No other transaction can read the values of data items which are stored here. In the case of partitions the differed updated will cause inconsistent results to be produced- the failure is catastrophic.

In conclusion, if the database is nonredundant, there is not very much to do in order to increase its availability in the presence of failures. Therefore, most reliability techniques consider the case of redundant databases.

## B. REDUNDANT DATABASE

The reasons why redundancy is introduced in a distributed database are twofold:

1. To increase the locality of reads, especially in those applications where the number of reads is much larger than the number of writes
2. To increase the availability and reliability of the system.

We deal here essentially with the second aspect; however, in designing reliable concurrency control methods for replicated data the first goal also should be kept in mind.

There are three main approaches to concurrency control based on 2-phase-locking in a redundant database: write-locks-all, majority locking, and primary copy locking.

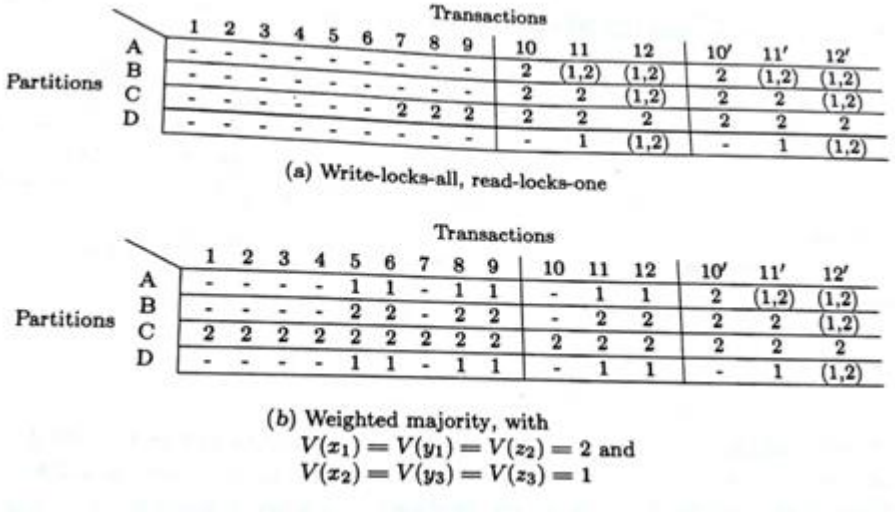
### I. WRITE-LOCK-ALL

For transaction with a small write-set and especially for read-only transactions, the system is much more available than for transaction with a large write-set. For read-only transactions sometimes can run in more than one group ,because if a data item has two copies in two copies in two different groups, then no update transaction can write on it and read-only transaction can use each copy consistently.

If we make the assumption that no partitions occur, but only site crashes, then the same approach can be used as with a nonredundant database i.e., the updates of unavailable copies of data items can be spooled. In this case, the availability of the database for update transactions increases very much. In fact, since only the read-set matters in the case, transaction 1,4and 7 have the same availability as transaction 10; transactions 2, 5 and 8 as transaction 11; and transaction 3,6 and 9 as transaction 12. So the example must be carefully interpreted. The fact that a transaction can run in a given group means now that it can be run if all other sites are down, instead of building separate groups. The high increase in availability is obtained at the risk of catastrophic partitions.

Requests to lock or unlock a data item and the messages of the 2-phase-commitment protocol are required for the control of transactions. Control messages carry information and are short. Data messages contain database information and can be long. With the write locks-all approach, we have:

1. **Benefit** - For each transaction executed at site i having x in its read-set, one lock message and one data message are saved.
2. **Cost** - for each transaction which is not executed at site i and has x in its write-set, one lock message and one data message are required, plus the messages required by the commitment protocol.



Transactions

| Partitions | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 10′ | 11′ | 12′ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | - | - | - | - | - | - | - | - | - | 2 | (1,2) | (1,2) | 2 | (1,2) | (1,2) |
| B | - | - | - | - | - | - | - | - | - | 2 | 2 | (1,2) | 2 | 2 | (1,2) |
| C | - | - | - | - | - | - | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| D | - | - | - | - | - | - | - | - | - | - | 1 | (1,2) | - | 1 | (1,2) |

(a) Write-locks-all, read-locks-one

Transactions

| Partitions | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 10′ | 11′ | 12′ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | - | - | - | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 | 2 | (1,2) | (1,2) |
| B | - | - | - | - | 2 | 2 | - | 2 | 2 | - | 2 | 2 | 2 | 2 | (1,2) |
| C | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| D | - | - | - | - | 1 | 1 | - | 1 | 1 | - | 1 | 1 | - | 1 | (1,2) |

(b) Weighted majority, with $V(x_1) = V(y_1) = V(z_2) = 2$ and $V(x_2) = V(y_3) = V(z_3) = 1$

Figure 4.4 Availability of Transaction

## II. WEIGHTED MAJORITY LOCKING

The pure majority locking approach is not very suitable for our example, because two copies of each data item exist; hence to lock a majority we must lock both. So consider a weighted majority approach, or quorum approach, which adopts the same rules which have been used for quorum-based commitment and termination protocols.

These rules, applied to the locking problem, consist of assigning to each data item x a total number of votes V(x), and assigning votes $V(x_i)$ to each copy $x_i$ in such a way that V(x) is the sum of all $V(x_i)$. A read quorum $V_r(x)$ and a write quorum $V_w(x)$ are then determined, such that:

$$V_r(x) + V_w(x) > V(x)$$
$$V_w(x) > V(x)/2$$

A transaction can read(write)x if it obtains read(write) locks on so many copies of x that the sum of their votes is greater than or equal to $V_r(x)(V_w(x))$. Due to the first condition, all conflicts between read and write operations are determined, because two transactions which perform a read and a write operation on x cannot reach the read and write quorum using two disjoint subsets of copies. Likewise, because of the second condition, all conflicts between write operations are determined. Notice that the second condition can be omitted if transactions read all data items which are written.

Let us assign votes for the copies of data items of Figure in the following way:

$$V(x) = V(y) = V(z) = 3$$
$$V(x1) = V(y1) = V(z2) = 2$$
$$V(x2) = V(y3) = V(z3) = 1$$

With this assignment we can now consider the availability of the system in the case of partitions. We choose the read and write quorums to be 2 for all data items. The availability for the 12 transaction is shown in the figure. The following can be observed:

1. Transaction 1,2,3,4,7 and 10 have all the same availability. They are characterized by the fact that they access all three data items either for reading or for writing or for both. Since the read quorum is equal to the write quorum, it makes no difference whether the data item is read or written from the viewpoint of availability. For the same reason, transaction 5,6,8and 11, which access only data items x and y, have the same availability. Also, transactions 9 and 12 have the same availability of the latter group, because the copy with highest weight for y.
2. The availability for update transactions is grater with the weighted majority approach than with write-locks-all, while the availability for read-only transactions is smaller.
3. With this method , read-only transaction increases their availability if they can read an inconsistent database, i.e., if they do not need to lock items, in fact, columns 10',11,and 12, are the same for the majority approach as for the write-locks-all approach .

With the majority approach it is not reasonable to consider the assumption that partitions do not occur. Notice that if we assume the absence of partitions, then the majority approach is dominated by the write-locks-all approach(an approach is dominated by another one if it is worse under all circumstances). In fact, we have seen that the majority and quorum ideas have been developed essentially for dealing with partitions.

Consider now the locality aspect. A transaction reads a data item x at its site of origin, if a copy is locally available. Hence, also in this case a data message is saved if a local copy is available .However, read locks must be obtained at a number of copies corresponding to the read quorum. Therefore, the addition of a copy of x can also force transactions which read x to request more read locks at remote sites. This additional cost is incurred by transactions which have x in their write-set, which must obtain write locks at a number of sites corresponding to the write quorum. Moreover, they have to send a data message to all the sites where there are copies of x.

It is clear that, considering only data messages, the same advantages and disadvantage exist for the majority and the write-locks-all method. When control message are also considered, then the situation is more complex; however, some of the locality motivations for read-only transaction are lost.

## III.PRIMARY COPY LOCKING

In the primary copy locking approach, all locks for a data item x are requested at the site of the primary copy. We will assume first that also all he read and write operations are performed on this copy; however, write are then propagated to all other copies.

Several enhancements of the primary copy approach exist which it more attractive. The principal ones are:

1. Allowing consistent reads at different copies than the primary, even if real locks are requested only at the primary; this enhances the locality of reads.
2. Allowing the migration of the primary copy if a site crash makes it unavailable; this enhances availability.
3. Allowing the migration of the primary copy depending on its usage pattern. This also enhances the locality aspect.

The first point deserves a comment. In order to obtain consistent reads at different copies from the primary one, we should use the primary copy method for synchronization, but perform the write and read operations according to the "write all/read one" method. In this approach, the locks are all requested at the primary copy. So, at commitment all copies are updated before releasing the write lock. A read can be performed in this way at any copy, obtaining consistent data.

**DETERMINING A CONSISTANT VIEW OF THE NETWORK**

There are two aspects of this problem: Monitoring the state of the network, so that state transitions of a site are discovered as soon as possible, and propagating new state information to all sites consistently. Normally we use timeouts in the algorithms in order to discover if a site was down. The use of timeouts can lead to an inconsistent view of the network. Consider the following example in a 3-site network: site 1 sends a message to site2 requesting an answer. If no answer arrives before a given timeout, site 1 sends assumes that sites 2 is down. If site 2 was just slow, then site 1 has a wrong view of the state of site2, which is inconsistent with the view of site 2 about itself. Moreover, a third site 3 could try the same operation at the same time as site 1, obtain an answer within the timeout, and assume that site 2 is up. So it has different view that site1.

A generalized network wide mechanism is built such that all higher-level programs are provided with the following facilities:

1. There is at each site a **state table**containing an entry for eachsite. The entry can be up or down. A program can send an inquiry to the state table for state information.
2. Any program can set a "watch" on any site, so that it receives an interrupt when the site changes state.

The meaning of the state table and of a consistent view in the presence of partitions failures is defined as follow: A site considers up only those sites with which it can communicate.So all crashed sites and all sites which belong to a different group in case of partitions are considered down.A consistent view can be achieved only between sites of the same group.Incase of a partition there are as many consistent views as there are isolated groups of sites. The consistency requirement is therefore that a site has the same state table as all other sites which are up in its state table.

**I.Monitoring the state of the network**

The basic mechanism for deciding whether a site is up or down is to request a message from it and to wait for a timeout. The requesting site is called controller and the other site is called controlled site. In a generalized monitoring algorithm, instead of having the controller request message from the controlled site, it is more convenient to have the controlled site send I-AM-UP messages periodical to the controller and the controlled site.

Note that if only site crashes are considered, the monitoring function essentially has to detect transitions from up to down states, because the opposite transaction is detected by the site which performs recovery and restart; this site will inform all the others. If, however, partitions also are considered, then the monitoring function has also to determine transitions from down to up states. When a partition is repaired, sites of one group must detect that sites of the other group must detect that sites of the group become available.

Using this mechanism for detecting whether a site is up or down the problem consists of assigning controllers to each site so that the overall message overhead is minimized and the algorithm survives correctly the failure of a controller. The latter requirement is of extreme importance, since in a distributed approach each site is controlled and at the same time performs the function of controller of some other site.

A possible solution is to assign circular ordering to the sites and to assign to each site the function of controller of its predecessor. In the absence of failures, each site periodically sends an I-AM-UP message to its successor and controls that the I-AM-UP message from its predecessor arrives in time. If the I-AM-UP message from the predecessor does not arrive in time, then the controller assumes that the controlled site has failed, updates the state table and broadcasts the updated state table to all other sites.

If the predecessor of a site is down,then the site also has to control its predecessor, and if this one is also down, the predecessor of the predecessor, and so on backward in the ordering until an up site is found is isolated or all other sites have crashed; this does not invalidate the algorithm). In this way, each operational site always has a controller. For example, in site k controls site k-3; i.e., it responsible for discovering that sites k-1 and k-2 recover from down to up. Symmetrically, if the successor of a site is down, then this site has as a controller the first operational site following it in the ordering. For example, site k-3 has site k as controller. Note that in the **FIG sites k-1 and k-2** is not necessarily crashed; they could belong to a different group after a partition.Therefore, the view of the network of sites k and k-3 is not necessarily the “real” state.

**Broadcasting a New State**

Each time that the monitor function detects a stage change, this function is activated. The purpose of this function is to broadcast the new state table so that all sites of the same group have the same state table so that all sites of the same group have the same state table. Since this function could be activated by several sites in parallel, some mechanism is needed to control interference. A possible mechanism is to attach a globally unique timestamp to each new version of a state table. By including the version number of the current state table in the I-am-up message all sites in the same group can check that they have a consistent view.

The site which starts the propagation of a new state table first performs a synchronization step in order to obtain a timestamp and then sends the state table to all sites which have answered.

## DETECTION AND RESOLUTION OF INCONSISTENCY

When a partition of the network occurs, transaction should be run at most in one group of sites if we want to preserve strictly the consistency of the database. In some application it is acceptable to lose consistency in order to achieve more availability. In this case, transaction is allowed to run in all partitions where there is at least one copy of the necessary data. When the failure is repaired, one can try to eliminate the inconsistencies

which have been introduced into the database. For this purpose it is necessary first to discover which portion of the data has become inconsistent, and then to assign to these portions a value which is the most reasonable in consideration of what has happened. The first problem is called the **detection ofinconsistencies.**The second is called the **resolution** of the inconsistencies. While exact solutions can be found for the detection problem, the resolution problem has no general solution, because transaction has serializable way. Therefore the word “reasonable” and not the word”correct” is used for the value which is assigned by the resolution procedure.

**DETECTION OF INCONSISTENCIES**

Let us assume that, during a partition, transaction have been executed in two or more groups of sites, and that independent updates may have been performed on different copies of the same fragment. Let us first observe that the most naïve solution,consisting of comparing the contents of the copies to check that they are identical, is not only inefficient, but also not correct in general. For example consider an airline reservation system. If, during the partition, reservation for the same flight independently on different copies until the maximum number is reached, then all copies might have the same value for the number of reservation; however, the flight would be overbooked in this case.

A correct approach to the detection of inconsistencies can be based on version number .Assume that one of the approaches is used for determining for each data item, the one group of sites which is allowed to operate on it. The copies of the data item which are stored at the sites of this group are called **master copies**; the others are called **isolated copies.**

During normal operation all copies are master copies and are mutually consistent. For each copy an original **version number** and a **current version number** are maintained .Initially the original version number is set to 0, and the current version number is set to 1; only the current version number is incremented each time that an update is performed on the copy. When a partition occurs, the original version number of each isolated copy is set to the value of its current version number. In this way, the original version number is not altered until the partition is repaired. At this time, the comparison of the current and original version numbers of all copies reveals inconsistencies.

Let us consider an example of this method. Assume that copies x1, x2 and x3 of data item x are stored at three different sites. Let V1,V2 and V3 be the version numbers of x1, x2 and x3. Each Vi is in fact a pair with the original and current version number. Initially all three copies are consistently updated .Suppose that one update has been performed, so that the situation is

$$V1=(0,2),\ V2=(0,2),\ V3=(0,2)$$

Now a partition occurs separating x3 from the other two copies. A majority algorithm is used which chooses x1 and x2 as major copies. The version numbers become now

$$V1=(0,2),\ \ V2=(0,2)\ ,\ V3=(2,2)$$

Suppose now that only the master copies are updated during the partitions. The version numbers become

$$V1=(0,3),\ V2=(0,3),\ V3=(2,2)$$

And after the repair it is possible to see that x3 has not been modified, since the current and original version numbers are equal. In this case, no inconsistency has occurred and it is sufficient to perform the updated during the partition. We have

$$V1=(0,2),\ V2=(0,2),\ V3=(2,3)$$

Since the original version number of x3 is equal to the current version number of x1 and x2 , the master copies have not been updated. If there are no other copies, then we can simply apply to the master copies the updates of x3, since the situation is exactly symmetrical to the previous one. If there are other isolated copies, for example x4 with V4=(2,3), we cannot tell whether x4 was updated consistency with x3 even if version numbers are the same, hence we have to assume inconsistency.

Finally , if both the master and the isolated copies have been updated , which also reveals an inconsistency, then the original and the current version number of the isolated copy are different, and the original version number of the isolated copy is also different from the current version number of the master copies; for example

$$V1=(0,3),\ V2=(0,3),\ V3=(2,3)$$

**RESOLUTION OF INCONSISTENCIES**

After a partition has been repaired and an inconsistency has been detected a common value must be assigned to all copies of a same data item. The problem of resolution of inconsistency is the determination of this value.

Since in the different group transaction have been executed without mutual synchronization, it seems correct to assign as a common value the one which would be produced by some serializable execution of these same transactions. However besides the difficulty of obtaining this new value, this is not a satisfactory solution, because the transactions which have been executed have produced effects outside of the system which cannot be undone and cannot be simply ignored.

Note that the transaction requiring the high degree of availability which motivates the acceptances of inconsistencies is exactly those which perform effects outside of the system. For example, take the airline reservation example considered before. The reason for running transaction while the system is partitioned is to tell the customers that flight are available; otherwise , it would be simpler to collect the customer request and to apply them to the database after the failure has been repaired.

However, if overbooking has occurred during the partition, then forcing the system to serializable execution would force the system to perform arbitrary cancellation. From the view point of the application, it might be better to keep the over bookings and let normal user cancellations reduce the number of reservations. A possible way of reducing or eliminating overbooking due to partitions is to assign to each site a number of reservations which is smaller than the total number. This number could be proportional to the size of each group or to some other application dependent value.

The above example shows that the resolution of inconsistencies is in general, application-dependent, and hence within the scope of this book, which deals with generalized mechanisms.

## CHECKPONTS AND RESTART

There are two types for errors: Omission errors and Commission errors. Omission errors occur when a action (commit/abort) is left out of the transactions being executed. Commission errors occur when a action (commit/abort) is incorrectly included in the transactionexecuted. An error of omission in one transaction will be counted as an error in commission in another transaction.

Cold restart is required after some catastrophic failure which has caused the low of log information on stable storage, so that the current consistent state of the database cannot be reconstructed and a previous consistent state must be restored. A previous consistent state is marked by a checkpoint.

In a distributed database, the problem of cold restart is worse than in a centralized one; this is because if one site has to establish an earlier state then all other sites also have to establish earlier states which are consistent with the one of the site, so that the global state of the distributed database as a whole is consistent. This means that the recovery process is intrinsically global, affecting all sites of the database, although the failure which caused the cold restart is typically local.

A consistent global state C is characterized by the following two properties:

1. For each transaction T, C contains the updates performed by all subtransactions of T at any site, of it does not contain any of them; in the former case we say that T is contained in C.
2. If a transaction T is contained in C, then all conflicting transactions which have preceded T in the serialization order are also contained in C.

Property 1 is related to the atomicity to the transactions: either all effects of T or none of them can appear in a consistent state. Property 2 is related to the serializability of transactions: if a conflicting transaction T' has preceded T, then the updates performed by T' have affected the execution of T; Hence, if we keep the effects of T , we must keep also all the effects of T' . Note that durability of transaction cannot be ensured if we are forced to a cold restart; the effect of some transactions is lost.

The simplest way to reconstruct a global consistent state in a distributed database is to use local dumps, local logs, and **global checkpoints**. A global checkpoint is a set of local checkpoints which are performed all sites of the network and are synchronized by the following condition: if a subtransaction of a transaction T is contained in the local checkpoint at some site, then all other subtransactions of T must be contained in the corresponding local checkpoint at other sites.

If global checkpoints are available, the reconstruction problem is relatively easy. First, at the failed site the latest local checkpoint which can be considered safe is determined; this determines which earlier global state has to be reconstructed. Then all other sites are required to reestablish the local states of the corresponding local checkpoints.

The main problem with the above approach consists in recording global checkpoints. It is not sufficient for one site to broadcast a "write checkpoints" message to all other sites, because it is possible that the situation of Fig arises; in this situation, T2 and T3 are

subtransactions of the same transaction T, and the local checkpoint C2 does not contain subtransaction T2, while the local checkpoint C3 contains sub transaction T3, thus violating the basic requirement for global checkpoints. FIGURE shows also that the fact that T performs a 2- phase-commitment does not eliminate this problem, because the synchronization of subtransactions during 2-phase-commitment and of sites during recording of the global checkpoint is independent.

The simplest way to avoid the above problem is to require that all sites become inactive before each other records its local checkpoint. Note that all sites must remain inactive simultaneously,and therefore coordination is required. A protocol which is very similar to 2-phase-commitment can be used for this purpose; a coordinator broadcasts " prepare for checkpoint" to all sites, each site terminates the execution of subtransactions and then answers READY, and then the coordinator broadcasts " perform checkpoint". This type of method is unacceptable in practice because of the inactivity which is required all the sites. A site has to remain inactive not only for the time required to record its checkpoints, but until all other sites have finished their active transactions. Three more efficient solutions are possible:

1. To find less expensive ways to record global checkpoints, so called **loosely synchronized checkpoints.** All sites are asked by a coordinator to record a global checkpoint; however, they are free to perform it within a large time interval. The responsibility of guaranteeing that all subtransaction of the same transaction are contained in the local checkpoints corresponding to the same global checkpoint is left to transaction management. If the root agent of transaction T starts after checkpoint $C_i$ and before checkpoint $C_{i+1}$ , then each other subtransaction at a different site can be started only after $C_i$ has been recorded at its sites and before $C_{i+1}$ has been recorded . Observing the first condition may force a subtransaction to wait; observing the second condition can cause transaction aborts and restarts.
2. To avoid building global checkpoints at all, let the recovery procedure take the responsibility of reconstructing a consistent global state at cold restart. With this approach, the notion of global checkpoint is abandoned. Each site records its local checkpoints independently from other sites, and the whole effort of building a consistent global state is therefore performed by the cold restart procedure.
3. To use the 2-phase-commitment protocol for guaranteeing that the local checkpoints created by each sites are ordered in a globally uniform way. The basic ideas is to modify the 2-phase-commitment protocol so that the check points idea is to modify the 2-phase-commitment protocol so that the checkpoints of all subtransactions which belong to two distributed transaction T and $T^1$ are recorded in the same order at all sites where both transaction T and T' are recorded in the same order at all sites where both transactions are executed. Let $T_i$ and $T_j$ be subtransactions $T'_i$ and $T_j$'be subtransactions of T'. If at site i the checkpoint of subtransaction $T_i$proceeds the checkpoint of $T_J$ should precede the checkpoint of subtransaction $T'_j$.

## DISTRIBUTED DATABASE ADMIBISTRATION

Database administration refers to a variety of activities for the development, control, maintenance, and testing of the software of the database application. Database administration

is not only a technical problem, since it involves the statement of policies under which users can access the database, which is clearly also an organization problem.

The technical aspects of database administration in a distributed environment focus on the following problems:

1. The content and management of the catalogs with this name, we designate the information which is required by the system for accessing the database. In distributed systems, catalogs include the description of fragmentation and allocation of data and the mapping to local names.
2. The extension of protection and authorization mechanisms to distributed systems.

## CATALOG MANAGEMENT IN DISTRIBUTED DATABASES

Catalogs of distributed databases store all the information which is useful to the system for accessing data correctly and efficiently and for verifying that users have the appropriate access rights to them.

Catalogs are used for:

1. **Translating applications** - Data referenced by applications at different levels of transparency are mapped to physical data (physical images in our reference architecture).
2. **Optimizing applications -** Data allocation, access methods available at each site, and statistical information (recorded in the catalogs) are required for producing an access plan.
3. **Executing applications** - Catalog information is used to verify that access plans are valid and that the users have the appropriate access rights.

Catalogs are usually updated when the users modify the data definition.It happens when global relations, fragments, or images are created or moved, local access structures are modified, or authorization rules are changed.

## I. CONTENT OF CATALOGS

Several classifications of the information which is typically stored in distributed database catalogs are possible.

1. **Global schema description -**It includes the name of global relations and of attributes.
2. **Fragmentation description -**In horizontal fragmentation, it includes the qualification of fragments. In vertical fragmentation, it includes the attributes which belong to each fragment.In mixed fragmentation, it includes both the fragmentation tree and the description of the fragmentation corresponding to each nonleaf node of the tree.
3. **Allocation description -** It gives the mapping between fragments and physical images.
4. **Mappings to local names -**It is used for binding the names of physical images to the names of local data stored at each site.
5. **Access method description -**It describes the access methods which are locally available at each site. For instance, in the case of a relational system, it includes the number and types of indexes available.
6. **Statistics on the database** - They include the profiles of the database

7. **Consistency information (protection and integrity constraints) -** It includes information about the users' authorization to access the database, or integrity constraints on the allowed values of data.
   Examples of authorization rules are:
   a. Assessing the rights of users to perform specific actions on data. The typical actions considered are: read, insert, delete, update, move.
   b. Giving to users the possibility of granting to other users the above rights. Some references in the literature also include in the catalog content state information (such as locking or recovery information); it seems more appropriate to consider this information as part of a system's data structure and not of the catalog's content.

## II. THE DISTRIBUTION OFCATALOGS

When catalogs are used for the translation, optimization, and execution of applications, their information is only retrieved.When they are used in conjunction with a change in data definitions, they are updated. In a few systems, statistics are updated after each execution, but typically updates to statistics are batched. In general, retrieval usage is quantitatively the most important, and therefore the ratio between updates and queries is small.

Solutions given to catalog management with and without site autonomy are very different. Catalogs can be allocated in the distributed database in many different ways. The three basic alternatives are:

**Centralized catalogs**

The complete catalog is stored at one site.This solution has obvious limitations, such as the loss of locality of applications which are not at the central site and the loss of availability of the system, which depends on this single central site.

**Fully replicated catalogs**

Catalogs are replicated at each site.This solution makes the read-only use of the catalog local to each site, but increases the complexity of modifying catalogs, since this requires updating catalogs at all sites.

**Local catalogs**

Catalogs are fragmented and allocated in such a way that they are stored at the same site as the data to which they refer.

A practical solution which is used in several systems consists of periodically caching catalog information which is not locally stored. This solution differs from having totally replicated catalogs, because cached information is not kept up-to-date.

If an application is translated and optimized with a different catalog version than the up-to-date one, this is revealed by the difference in the version numbers. This difference can be observed either at the end of compilation, when the access plan is transmitted to remote sites, or at execution time.

In the design of catalogs for Distributed-INGRES, five alternatives were considered,

1. The centralized approach

2. The full replication of items 1, 2, and 3 of catalog content and the local allocation of remaining items
3. The full replication of items 1, 2, 3, 4, and 5 of catalog contentand the local allocation of remaining items
4. The full replication of all items 5 of catalog content.
5. The local allocation of all items with remote "caching"

SDD-1 considers catalog information as ordinary user data; therefore an arbitrary level of redundancy is supported. Security, concurrency, and recovery mechanisms of the system are also used for catalog management.

## III. OBJECT NAMING AND CATALOG MANAGEMENT WITH SITE AUTONOMY

We now turn our attention to the different problems which arise when site autonomy is required. The major requirement is to allow each local user to create and name his or her local data independently from any global control, at the same time allowing several users to share data.

1. Data definition should be performed locally.
2. Different users should be able, independently, to give the same name to different data.
3. Different users at different sites should be able to reference the same data.

In the solution given to these problems in R*prototype, two types of names is used:

1. **System wide names** are unique names given to each object in the system.
   They have four components:
   a. The identifier of the user who creates the object
   b. The site of that user
   c. The object name
   d. The birth site of the object, i.e., the site at which the object was created.

   An example of a systemwide name is

   User_ 1 @San_ Jose.EMP@Zurich

   where the symbol @ is a separator which precedes site names.
   Here, User_ 1 from San Jose has created a global relation EMP at Zurich. The same user name at different sites corresponds to different users (i.e., JohnOSF is not the same as JohnOLA). This allows creating user names independently.
2. **Print names** are shorthand names for systemwide names. Sine in systemwide names a, b, and d part can be omitted, name resolution is made by context, where a context is defined as the current user at the local site.

   a. A missing user identifier is replaced by the identifier of the current user.
   b. A missing user site or object site is replaced by the current site.

It is also possible for each user to define synonyms, which map simple names to systemwide names. Synonyms are created for a specific user at a specific site.Synonym mapping of a simple name to a systemwide name is attempted before name resolution.

Catalog management in R* satisfies the following requirements:

1. Global replication of a catalog is unacceptable, since this would violate the possibility of autonomous data definition.
2. No site should be required to maintain catalog information of objects which are not stored or created there.
3. The name resolution should not require a random search of catalog entries in the network.
4. Migration of objects should be supported without requiring any change in programs.

The above requirements are met by storing catalog entries of each object as follows:

1. One entry is stored at the birth site of the object, until the object is destroyed. If the object is still stored at its birth site, the catalog contains all the information; otherwise, it indicates the sites at which there are copies of the object.
2. One entry is stored at every site where there is a copy of the object.

The catalog content in R* includes relation names, column names and types, authorization rules, low-level objects' names, available access paths, and profiles. R* supports the "caching" of catalogs, using version numbers to verify the validity of cached information.

## AUTHORIZATION AND PROTECTION

### I. Site-to-Site Protection

The first security problem which arises in a distributed database is initiating and protecting intersite communication. When two database sites communicate, it is important to make sure that:

1. At the other side of the communication line is the intended site (and not an intruder).
2. No intruder can either read or manipulate the messages which are exchanged between the sites.

The first requirement can he accomplished by establishing an identification protocol between remote sites. When two remote databases communicate with each other, on the first request they also send each other a password. When two sites decide to share some data they follow R* mechanism.

The second requirement is to protect the content of transmitted messages once the two identified sites start to communicate. Messages in a computer network are typically routed

along paths which involve several intermediate nodes and trans-missions, with intermediate buffering.

The best solution to this problem consists of using cryptography, a standard technique commonly used in distributed information systems, for instance for protecting communications between terminals and processing units. Messages ("plaintext") are initially encoded into cipher messages ("ciphertext") at the sender site, then transmitted in the network, and finally decoded at the receiver site.

## II. User Identification

When a user connects to the database system, they must be identified by the system.The identification is a crucial aspect of preserving security, because if an intruder could pretend to be a valid user, then security would be violated.

In a distributed database, users could identify themselves at any site of the distributed database. However, this feature can be implemented in two ways which both show negative aspects.

1. Passwords could be replicated at all the sites of the distributed database. This would allow user identification to be performed locally at each site, but would also compromise the security of passwords, since it would they easier for an intruder to access them.
2. Users could each have a "home" site where their identification is performed; in this scenario, a user connecting to a different site would be identified by sending a request to the home site and letting this site perform the identification.

A reasonable solution is to restrict each user to identifying themselves at the home site. This solution is consistent with the idea that users seem to be more "static" than, for instance, data or programs. A "pass-through" facility could be used to allow users at remote sites to connect their terminals to their "home" sites in order to identify themselves.

## III. Enforcing Authorization Rules

Once users are properly identified, database systems can use authorization rules to regulate the actions performed upon database objects by them. In a distributed environment, additional problems include the allocation of these rules, which are part of the catalog, and the distribution of the mechanisms used for enforcing them. Two alternative, possible solutions are:

1. Full replication of authorization rules. This solution is consistent with having fully replicated catalogs, and requires mechanisms for distributing online updates to them. But, this solution allows authorization to be checked either at the beginning of compilation or at the beginning of execution.
2. Allocation of authorization rules at the same sites as the objects to which they refer. This solution is consistent with local catalogs and does not incur the update overhead as in the first case.

The second solution is consistent with site autonomy, while the first is consistent with considering a distributed database as a single system.

The authorizations that can be given to users of a centralized database include the abilities of reading, inserting, creating, and deleting object instances (tuples) and of creating and deleting objects (relations or fragments).

## IV. Classes of Users

For simplifying the mechanisms which deal with authorization and the amount of stored information, individual users are grouped into classes, which are all granted the same privileges.

In distributed databases, the following considerations apply to classes of users:

1. A "natural" classification of users is the one which is induced by the distribution of the database to different sites.It is likely that "all users at site x" have some common properties from the viewpoint of authorization. An explicit naming mechanism for this class should be provided.
2. Several interesting problems arise when groups of users include users from multiple sites. Problems are particularly complex when multiple-site user groups are considered in the context of site autonomy. So, mechanisms involve the consensus of the majority or of the totality of involved sites, or a decision made by a higher-level administrator. So, multiple-site user groups contrast with pure site autonomy.