SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

(Estd U/S(3) of UGC Act, 1956) Accredited with Grade "A" by NAAC

SCS1613- DISTRIBUTED DATABASE

UNIT 4: RELIABILTY AND PROTECTION

1. Reliability: Basic Concepts
2. Reliability and concurrency Control
3. Determining a Consistent View of the Network
4. Detection and Resolution of Inconsistency
5. Checkpoints and Cold Restart-
6. Distributed Database Administration
7. Catalog Management in Distributed Databases
8. Authorization and Protection

✓**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 robustness of concurrency control
3. Determining the state of the network
4. Detection and resolution of inconsistencies.
5. Checkpoints and cold restart.

•A reliable distributed database management system is one that can continue to process user requests even when the underlying system is unreliable.

•In other words, even when components of the distributed computing environment fail, a reliable distributed DBMS should be able to continue executing user requests without violating database consistency.

- Reliability refers to a system that consists of a set of components. The system has a state, which changes as the system operates. The behavior of the system in providing response to all the possible external stimuli is laid out in an authoritative specification of its behavior.
- The specification indicates the valid behavior of each system state. Any deviation of a system from the behavior described in the specification is considered a failure. For example, in a distributed transaction manager the specification may state that only serializable schedules for the execution of concurrent transactions should be generated. If the transaction manager generates a non-serializable schedule, we say that it has failed.

Database reliability is defined broadly to mean that the database performs consistently without causing problems. More specifically, it means that there is accuracy and consistency of data. For data to be considered reliable, there must be:

Data integrity, which means that all data is the database is accurate and that there is consistency throughout data. Data consistency is defined broadly to include the type and amount of data.

Data safety, which means that only authorized individuals access the database. Data security also includes preventing any type of data corruption and ensuring that data is always accessible. When it comes to data safety, engineers must ensure that data is accessible even in the event of unforeseen circumstances, such as emergencies or natural disasters.

Data recoverability, which means there are effective procedures in place to recover any lost data. This is a key to database reliability, ensuring that even if other safety measures fail, there is a system for recovering data.

The Importance of Database Reliability

•Organizational databases store a broad range of information, including customer information, sales information, financial transactions, vendor information, and employee records.

•This information is essential for maintaining the health of organizations and plays a central role in everything from competitive strategy to daily logistics.

•In many ways, data works as the eyes and ears of the organization, and without it, organizations lack the necessary information to make informed decisions.

•It's the database that makes this information accessible and usable.

•If an organization's database is not reliable, consistent, or accurate, it can lead to making bad or misinformed decisions.

•Further, as the database is a central part of organizational infrastructure, if it goes down, it can lead to substantial issues throughout the organization.

•This means that database reliability is and must remain a central concern for businesses.

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.

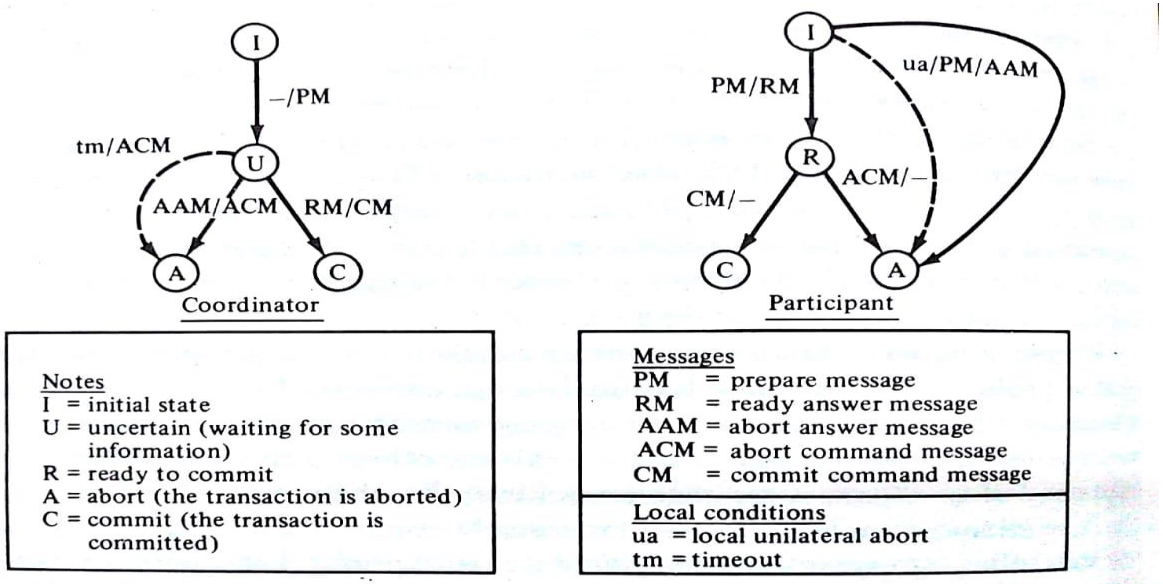


Figure 9.1 State diagrams for the 2-phase-commitment protocol.

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:

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.

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

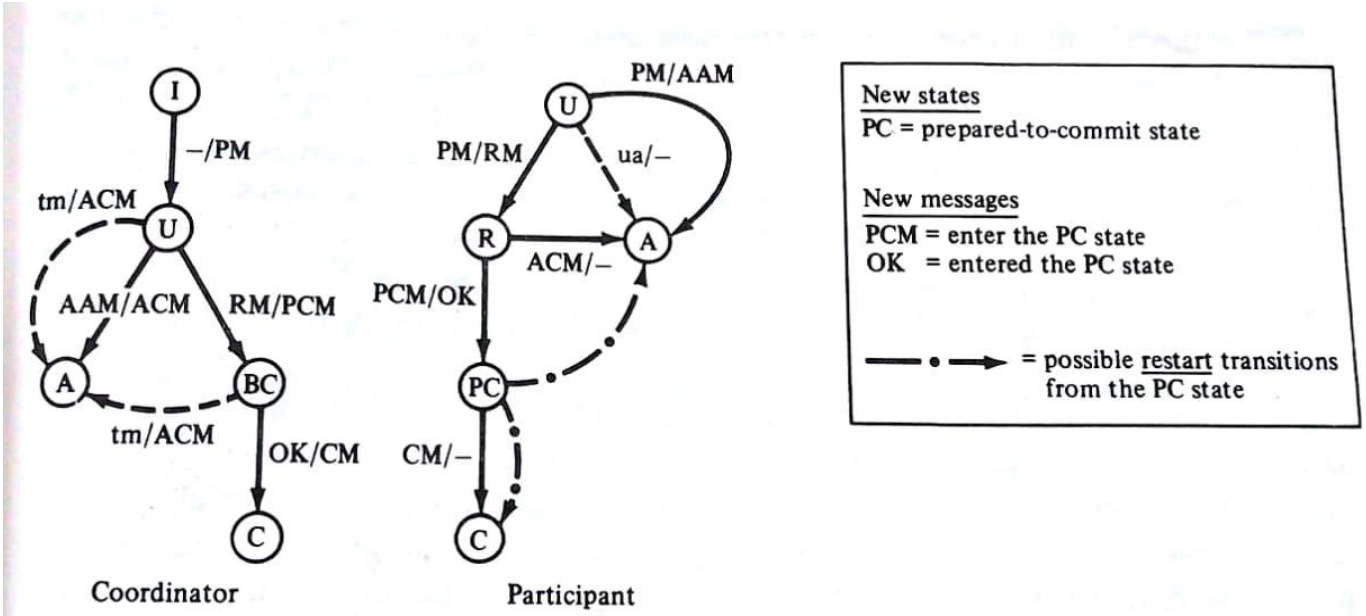


Figure 9.2 State diagrams for the 3-phase-commitment protocol.

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 item 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.

Recovery and Concurrency Control

Concurrency controlling techniques ensure that multiple transactions are executed simultaneously while maintaining the ACID properties of the transactions and serializability in the schedules.

Locking Based Concurrency Control Protocols:

Locking-based concurrency control protocols use the concept of locking data items. A **lock** is a variable associated with a data item that determines whether read/write operations can be performed on that data item. Generally, a lock compatibility matrix is used which states whether a data item can be locked by two transactions at the same time.

Locking-based concurrency control systems can use either one-phase or two-phase locking protocols.

One-phase Locking Protocol

In this method, each transaction locks an item before use and releases the lock as soon as it has finished using it. This locking method provides for maximum concurrency but does not always enforce serializability.

Two-phase Locking Protocol:

In this method, all locking operations precede the first lock-release or unlock operation. The transaction comprise of two phases. In the first phase, a transaction only acquires all the locks it needs and do not release any lock. This is called the expanding or the **growing phase**. In the second phase, the transaction releases the locks and cannot request any new locks. This is called the **shrinking phase**.

Every transaction that follows two-phase locking protocol is guaranteed to be serializable. However, this approach provides low parallelism between two conflicting transactions.

Timestamp Concurrency Control Algorithms

Timestamp-based concurrency control algorithms use a transaction's timestamp to coordinate concurrent access to a data item to ensure serializability. A timestamp is a unique identifier given by DBMS to a transaction that represents the transaction's start time.

These algorithms ensure that transactions commit in the order dictated by their timestamps. An older transaction should commit before a younger transaction, since the older transaction enters the system before the younger one.

Timestamp-based concurrency control techniques generate serializable schedules such that the equivalent serial schedule is arranged in order of the age of the participating transactions.

Some of timestamp based concurrency control algorithms are −

- Basic timestamp ordering algorithm.
- Conservative timestamp ordering algorithm.
- Multi version algorithm based upon timestamp ordering

Timestamp based ordering follow three rules to enforce serializability −

Access Rule − When two transactions try to access the same data item simultaneously, for conflicting operations, priority is given to the older transaction. This causes the younger transaction to wait for the older transaction to commit first.

Late Transaction Rule − If a younger transaction has written a data item, then an older transaction is not allowed to read or write that data item. This rule prevents the older transaction from committing after the younger transaction has already committed.

Younger Transaction Rule − A younger transaction can read or write a data item that has already been written by an older transaction.

Optimistic Concurrency Control Algorithm

In systems with low conflict rates, the task of validating every transaction for serializability may lower performance. In these cases, the test for serializability is postponed to just before commit. Since the conflict rate is low, the probability of aborting transactions which are not serializable is also low. This approach is called optimistic concurrency control technique.

In this approach, a transaction's life cycle is divided into the following three phases −

Execution Phase − A transaction fetches data items to memory and performs operations upon them.

Validation Phase − A transaction performs checks to ensure that committing its changes to the database passes serializability test.

Commit Phase − A transaction writes back modified data item in memory to the disk.

This algorithm uses three rules to enforce serializability in validation phase −

Rule 1 − Given two transactions T_i and T_j, if T_i is reading the data item which T_j is writing, then T_i's execution phase cannot overlap with T_j's commit phase. T_j can commit only after T_i has finished execution.

Rule 2 − Given two transactions T_i and T_j, if T_i is writing the data item that T_j is reading, then T_i's commit phase cannot overlap with T_j's execution phase. T_j can start executing only after T_i has already committed.

Rule 3 − Given two transactions T_i and T_j, if T_i is writing the data item which T_j is also writing, then T_i's commit phase cannot overlap with T_j's commit phase. T_j can start to commit only after T_i has already committed.

Concurrency Control in Distributed Systems

Distributed Two-phase Locking Algorithm

The basic principle of distributed two-phase locking is same as the basic two-phase locking protocol. However, in a distributed system there are sites designated as lock managers. A lock manager controls lock acquisition requests from transaction monitors. In order to enforce co-ordination between the lock managers in various sites, at least one site is given the authority to see all transactions and detect lock conflicts.

Depending upon the number of sites who can detect lock conflicts, distributed two-phase locking approaches can be of three types −

Centralized two-phase locking − In this approach, one site is designated as the central lock manager. All the sites in the environment know the location of the central lock manager and obtain lock from it during transactions.

Primary copy two-phase locking − In this approach, a number of sites are designated as lock control centers. Each of these sites has the responsibility of managing a defined set of locks. All the sites know which lock control center is responsible for managing lock of which data table/fragment item.

Distributed two-phase locking − In this approach, there are a number of lock managers, where each lock manager controls locks of data items stored at its local site. The location of the lock manager is based upon data distribution and replication.

Distributed Timestamp Concurrency Control

In a centralized system, timestamp of any transaction is determined by the physical clock reading. But, in a distributed system, any site's local physical/logical clock readings cannot be used as global timestamps, since they are not globally unique. So, a timestamp comprises of a combination of site ID and that site's clock reading.

For implementing timestamp ordering algorithms, each site has a scheduler that maintains a separate queue for each transaction manager. During transaction, a transaction manager sends a lock request to the site's scheduler. The scheduler puts the request to the corresponding queue in increasing timestamp order. Requests are processed from the front of the queues in the order of their timestamps, i.e. the oldest first

Conflict Graphs

Another method is to create conflict graphs. For this transaction classes are defined. A transaction class contains two set of data items called read set and write set. A transaction belongs to a particular class if the transaction's read set is a subset of the class' read set and the transaction's write set is a subset of the class' write set. In the read phase, each transaction issues its read requests for the data items in its read set. In the write phase, each transaction issues its write requests.

A conflict graph is created for the classes to which active transactions belong. This contains a set of vertical, horizontal, and diagonal edges. A vertical edge connects two nodes within a class and denotes conflicts within the class. A horizontal edge connects two nodes across two classes and denotes a write-write conflict among different classes. A diagonal edge connects two nodes across two classes and denotes a write-read or a read-write conflict among two classes.

The conflict graphs are analyzed to ascertain whether two transactions within the same class or across two different classes can be run in parallel.

Distributed Optimistic Concurrency Control Algorithm:

Distributed optimistic concurrency control algorithm extends optimistic concurrency control algorithm. For this extension, two rules are applied −

Rule 1 − According to this rule, a transaction must be validated locally at all sites when it executes. If a transaction is found to be invalid at any site, it is aborted. Local validation guarantees that the transaction maintains serializability at the sites where it has been executed. After a transaction passes local validation test, it is globally validated.

Rule 2 − According to this rule, after a transaction passes local validation test, it should be globally validated. Global validation ensures that if two conflicting transactions run together at more than one site, they should commit in the same relative order at all the sites they run together. This may require a transaction to wait for the other conflicting transaction, after validation before commit. This requirement makes the algorithm less optimistic since a transaction may not be able to commit as soon as it is validated at a site.

SATHYABAMA

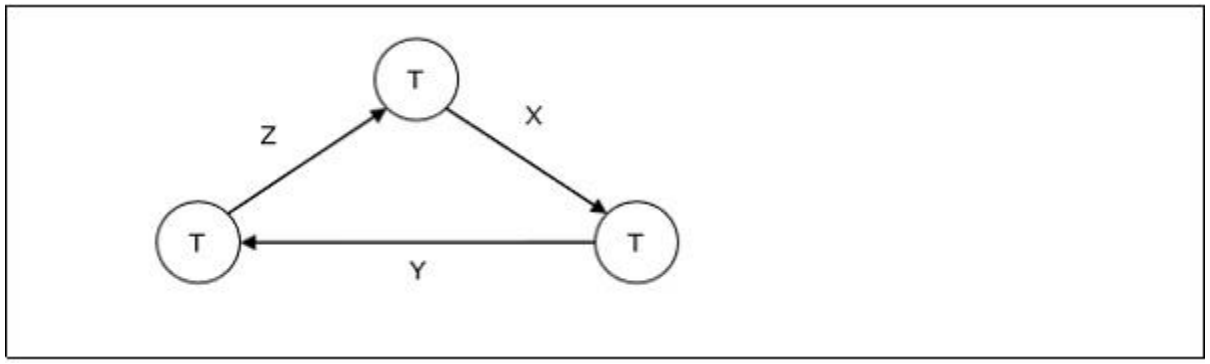
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

(Estd U/S(3) of UGC Act, 1956) Accredited with Grade "A" by NAAC

What are Deadlocks?

Deadlock is a state of a database system having two or more transactions, when each transaction is waiting for a data item that is being locked by some other transaction. A deadlock can be indicated by a cycle in the wait-for-graph. This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.

For example, in the following wait-for-graph, transaction T1 is waiting for data item X which is locked by T3. T3 is waiting for Y which is locked by T2 and T2 is waiting for Z which is locked by T1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing.



Deadlock Handling in Centralized Systems

There are three classical approaches for deadlock handling, namely −

- Deadlock prevention.
- Deadlock avoidance.
- Deadlock detection and removal.

All of the three approaches can be incorporated in both a centralized and a distributed database system.

Deadlock Prevention

The deadlock prevention approach does not allow any transaction to acquire locks that will lead to deadlocks. The convention is that when more than one transactions request for locking the same data item, only one of them is granted the lock.

One of the most popular deadlock prevention methods is pre-acquisition of all the locks. In this method, a transaction acquires all the locks before starting to execute and retains the locks for the entire duration of transaction. If another transaction needs any of the already acquired locks, it has to wait until all the locks it needs are available. Using this approach, the system is prevented from being deadlocked since none of the waiting transactions are holding any lock.

Deadlock Avoidance

•The deadlock avoidance approach handles deadlocks before they occur. It analyzes the transactions and the locks to determine whether or not waiting leads to a deadlock.

•The method can be briefly stated as follows. Transactions start executing and request data items that they need to lock.

•The lock manager checks whether the lock is available. If it is available, the lock manager allocates the data item and the transaction acquires the lock.

•However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not.

•Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

There are two algorithms for this purpose, namely **wait-die** and **wound-wait**. Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows −

Wait-Die − If T1 is older than T2, T1 is allowed to wait. Otherwise, if T1 is younger than T2, T1 is aborted and later restarted.

Wound-Wait − If T1 is older than T2, T2 is aborted and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to wait.

Deadlock Detection and Removal

The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. It does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.

Since there are no precautions while granting lock requests, some of the transactions may be deadlocked. To detect deadlocks, the lock manager periodically checks if the wait-forgraph has cycles. If the system is deadlocked, the lock manager chooses a victim transaction from each cycle. The victim is aborted and rolled back; and then restarted later. Some of the methods used for victim selection are −

Choose the youngest transaction.

Choose the transaction with fewest data items.

Choose the transaction that has performed least number of updates.

Choose the transaction having least restart overhead.

Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

Deadlock Handling in Distributed Systems:

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are **transaction location** and **transaction control**. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

Transaction Location

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

Transaction Control

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like −

One server may be selected as the center of control.

The center of control may travel from one server to another.

The responsibility of controlling may be shared by a number of servers

Distributed Deadlock Prevention

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks −

Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.

In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.

If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

Distributed Deadlock Avoidance

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur –

Conflict between two transactions in the same site.

Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows −

Distributed Wound-Die

If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.

If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.

Distributed Wait-Wait

If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.

If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

Distributed Deadlock Detection

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

Centralized Deadlock Detector − One site is designated as the central deadlock detector.

Hierarchical Deadlock Detector − A number of deadlock detectors are arranged in hierarchy.

Distributed Deadlock Detector − All the sites participate in detecting deadlocks and removing them.

Determining a Consistent View of the Network

•Consistency is the agreement between multiple nodes in a distributed system to achieve a certain value.
•Specifically, it can be divided into strong consistency and weak consistency.
Strong consistency: The data in all nodes is the same at any time. At the same time, you should get the value of key1 in node A and the value of key1 in node B.
Weak consistency: There is no guarantee that all nodes have the same data at any time, and there are many different implementations. The most widely achieved is the ultimate consistency. The so-called final consistency means that the same data on any node is the same at any time, but as time passes, the same data on different nodes always changes in the direction of convergence. It can also be simply understood that after a period of time, the data between nodes will eventually reach a consistent state.

Distributed and consistent application scenarios:
Multi-node provides read and write services to ensure high availability and scalability (ZooKeeper, DNS, redis cluster)

Problems faced by distributed systems:

Message asynchronous asynchronous (asynchronous):

The real network is not a reliable channel, there are message delays, loss, and inter-node messaging can not be synchronized (synchronous)

node-fail-stop: The node continues to crash and will not recover

node down recovery (fail-recover): Node recovery after a period of time, the most common in distributed systems

network partition: There is a problem with the network link, separating N nodes into multiple parts

Byzantine failure (byzantine failure) [2]: Node or downtime or logic failure, even without the card to throw out the interference resolution information

The general premise of distributed system design that needs to meet the consistency is that there is no Byzantine general problem (intranet trusted)

The basic theory of distributed systems, the FLP theorem, is mentioned here, that is, when only the node is down, the availability and strong consistency cannot be satisfied at the same time. Another point of view is CAP theory, namely strong consistency, availability and partition fault tolerance, only two of which can be guaranteed.

There are many agreements to ensure consistency, including 2PC, 3PC, Paxos, raft and PacificA.

2PC:

2-stage lock commit protocol to guarantee the atomicity of operations on multiple data slices. (distributed transaction)

The nodes are divided into coordinators and participants (participat), and the execution is divided into two phases.

Phase 1: The coordinator initiates a proposal to ask whether each participant is accepted. Participate performs transaction operations, writes undo and redo information to the transaction log, and replies yes or no to the coordinator.

Phase 2: The coordinator submits or aborts the transaction according to the feedback of the participant. If the participant is all yes, it is submitted, as long as there is a participant reply no. Participate formally commits or terminates the transaction according to the commit/Rollback information of the coordinator, and releases the occupied resources and returns ack.

Advantages: simple principle and easy implementation

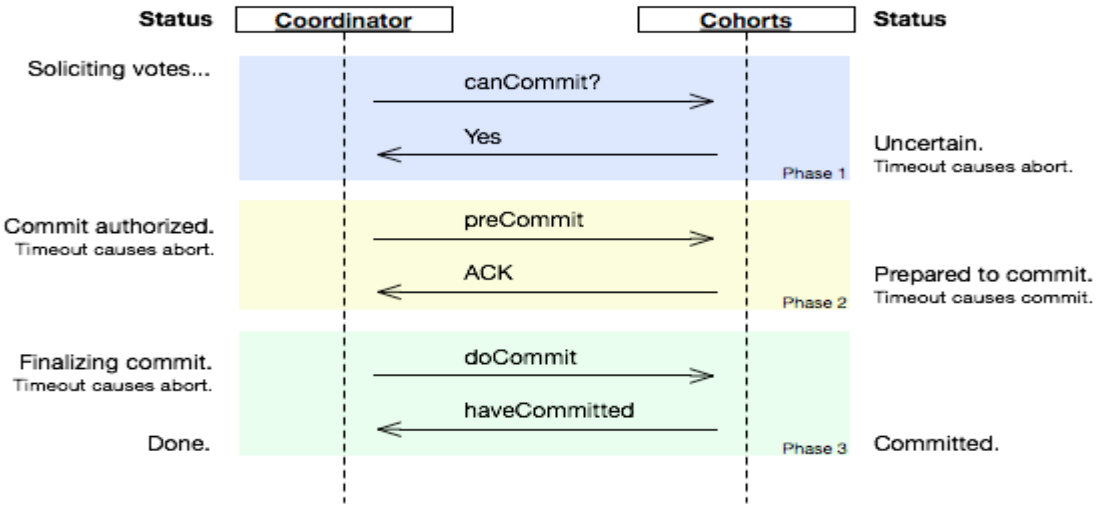
Disadvantages: synchronous blocking, single point problem, data inconsistency (coordinator crashes before sending commit request or network causes part of the consensus does not receive commit, then part of the participant cannot commit transaction), too conservative (if participant is in coordination) If there is a failure during communication, the coordinator can only rely on the timeout mechanism to determine whether the transaction needs to be interrupted.

3PC:

3-stage lock submission protocol to guarantee the atomicity of operations on multiple data slices. (distributed transaction)

Relative to 2PC, divided into inquiry, pre-submission, and submission of 3 stages (resolving blocking, but it is still possible that data is inconsistent)

Process: After receiving the participant's feedback (vote), the coordinator enters phase 2 and sends a prepare to commit command to each participant. The participant can lock the resource after receiving the instruction to submit, but requires the relevant operation to be rolled back. After receiving the acknowledgment (ACK), the coordinator enters phase 3 and commit/abort. Phase 3 of 3PC is no different from phase 2 of 2PC. Coordinator watchdog and status logging are also applied to 3PC.



Paxos algorithm (solving single point problems)

The Paxos algorithm is currently the most important consistency algorithm, and all consistency algorithms are a simplified version of paxos or paxos.

The Paxos algorithm resolves multiple values of the same data to agree on a value. The theoretical basis for proof of correctness: the intersection of any two legal sets (sets consisting of more than half of the nodes) is not empty.

Character:

There are three roles involved in the proposal to the voting process:

Proposer: There may be more than one proponent, and it is responsible for proposing the proposal.

Acceptor: There must be more than one recipient. They vote on the specified proposal, agree to accept the proposal, and disagree.

Learner: Learners, collect proposals accepted by each Acceptor, and form a final proposal based on the principle of minority majority.

In fact, a component in a distributed system can correspond to one or more roles.

Algorithm Description:

* The first stage (Prepare stage)

Proposer:

Select proposal number n and send a prepare request with number n to most Acceptors.

Acceptor:

If the proposal number n received is larger than the number already received, Proposer promises not to accept proposals with a number less than n. If the proposal has been accepted before, the proposal with the highest number among the accepted proposals will be The number is sent to Proposer.

If the proposal number n received is less than the maximum number of proposal numbers it has received.

* The second stage (Accept stage)

Proposer:

First, the response is received, one by one:

If a rejection is received, it will not be processed.

If you receive the consent and also receive a proposal that Acceptor has accepted, make a note of the proposal and the number.

After processing the response, count the number of rejections and consents:

If most reject, prepare for the next proposal.

If most agree, select the proposal with the largest proposal number as the proposal from the proposals accepted by these Acceptors, and use the proposal without the own, and send the Accept message to the Acceptor one by one.

Acceptor:

Whenever a proposal is accepted, the proposal and number are sent to Learner.

Learner:

Record the proposal currently accepted by each Acceptor. If an Acceptor sends multiple proposals in succession, the proposal with the highest number is retained.

Count the number of Acceptors accepted for each proposal. If more than half of them are accepted, a consensus will be formed.

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:

✓There is at each site a **state table** containing an entry for each site. The entry can be up or down. A program can send an inquiry to the state table for state information.

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.

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.

✓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 of inconsistencies.** 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)

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 transaction executed. 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.

Checkpointing

Checkpoint is a point of time at which a record is written onto the database from the buffers. As a consequence, in case of a system crash, the recovery manager does not have to redo the transactions that have been committed before checkpoint. Periodical checkpointing shortens the recovery process.

The two types of checkpointing techniques are −

- Consistent checkpointing
- Fuzzy checkpointing

Consistent Checkpointing

Consistent checkpointing creates a consistent image of the database at checkpoint. During recovery, only those transactions which are on the right side of the last checkpoint are undone or redone. The transactions to the left side of the last consistent checkpoint are already committed and needn't be processed again. The actions taken for checkpointing are −

- The active transactions are suspended temporarily.
- All changes in main-memory buffers are written onto the disk.
- A "checkpoint" record is written in the transaction log.
- The transaction log is written to the disk.
- The suspended transactions are resumed.

If in step 4, the transaction log is archived as well, then this checkpointing aids in recovery from disk failures and power failures, otherwise it aids recovery from only power failures.

Fuzzy Checkpointing

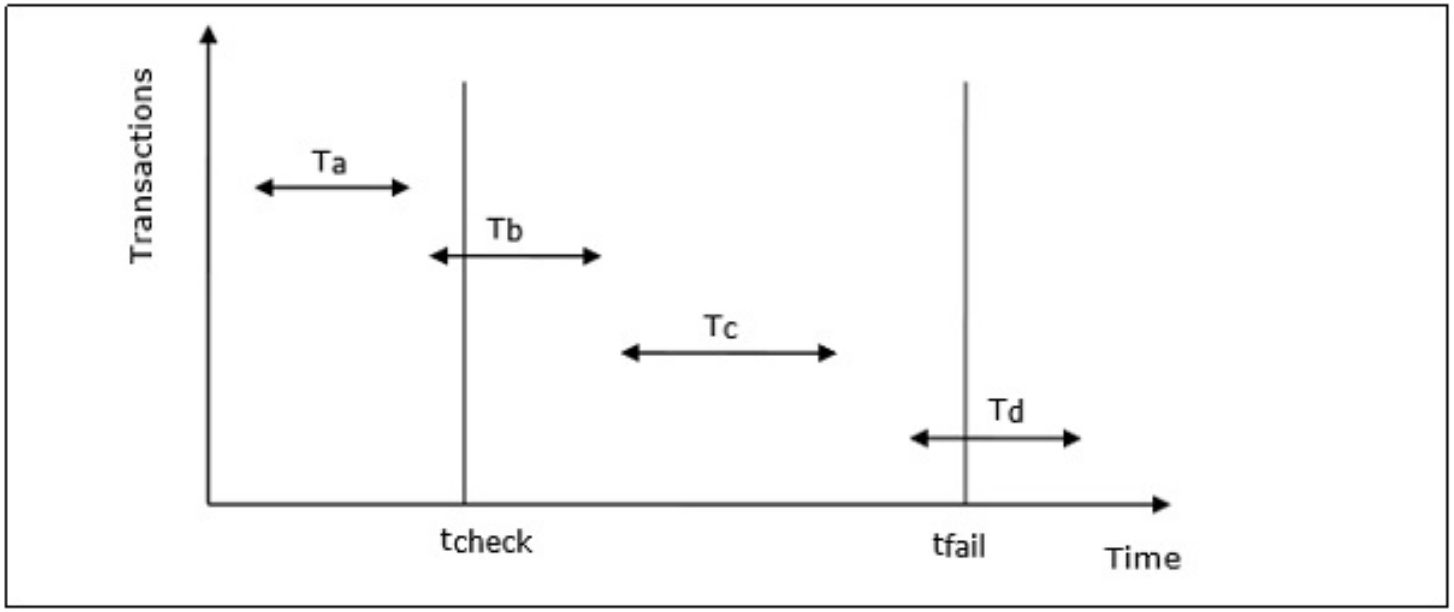
In fuzzy checkpointing, at the time of checkpoint, all the active transactions are written in the log. In case of power failure, the recovery manager processes only those transactions that were active during checkpoint and later. The transactions that have been committed before checkpoint are written to the disk and hence need not be redone.

Example of Checkpointing

Let us consider that in system the time of checkpointing is tcheck and the time of system crash is tfail. Let there be four transactions T_a, T_b, T_c and T_d such that −

- T_a commits before checkpoint.
- T_b starts before checkpoint and commits before system crash.
- T_c starts after checkpoint and commits before system crash.
- T_d starts after checkpoint and was active at the time of system crash.

The situation is depicted in the following diagram −



The actions that are taken by the recovery manager are −

- Nothing is done with T_a.
- Transaction redo is performed for T_b and T_c.
- Transaction undo is performed for T_d.

Transaction Recovery Using UNDO / REDO

Transaction recovery is done to eliminate the adverse effects of faulty transactions rather than to recover from a failure. Faulty transactions include all transactions that have changed the database into undesired state and the transactions that have used values written by the faulty transactions.

Transaction recovery in these cases is a two-step process –

- UNDO all faulty transactions and transactions that may be affected by the faulty transactions.
- REDO all transactions that are not faulty but have been undone due to the faulty transactions.

Steps for the UNDO operation are –

- If the faulty transaction has done INSERT, the recovery manager deletes the data item(s) inserted.
- If the faulty transaction has done DELETE, the recovery manager inserts the deleted data item(s) from the log.
- If the faulty transaction has done UPDATE, the recovery manager eliminates the value by writing the before-update value from the log.

Steps for the REDO operation are –

- If the transaction has done INSERT, the recovery manager generates an insert from the log.
- If the transaction has done DELETE, the recovery manager generates a delete from the log.

In a local database system, for committing a transaction, the transaction manager has to only convey the decision to commit to the recovery manager. However, in a distributed system, the transaction manager should convey the decision to commit to all the servers in the various sites where the transaction is being executed and uniformly enforce the decision. When processing is complete at each site, it reaches the partially committed transaction state and waits for all other transactions to reach their partially committed states. When it receives the message that all the sites are ready to commit, it starts to commit. In a distributed system, either all sites commit or none of them does.

The different distributed commit protocols are –

- One-phase commit
- Two-phase commit
- Three-phase commit

Distributed One-phase Commit

Distributed one-phase commit is the simplest commit protocol. Let us consider that there is a controlling site and a number of slave sites where the transaction is being executed. The steps in distributed commit are –

- After each slave has locally completed its transaction, it sends a "DONE" message to the controlling site.
- The slaves wait for "Commit" or "Abort" message from the controlling site. This waiting time is called **window of vulnerability**.

- When the controlling site receives "DONE" message from each slave, it makes a decision to commit or abort. This is called the commit point. Then, it sends this message to all the slaves.
- On receiving this message, a slave either commits or aborts and then sends an acknowledgement message to the controlling site.

Distributed Two-phase Commit

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows −

Phase 1: Prepare Phase

- After each slave has locally completed its transaction, it sends a "DONE" message to the controlling site. When the controlling site has received "DONE" message from all slaves, it sends a "Prepare" message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a "Ready" message.
- A slave that does not want to commit sends a "Not Ready" message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

Phase 2: Commit/Abort Phase

- After the controlling site has received "Ready" message from all the slaves −
  - The controlling site sends a "Global Commit" message to the slaves.

- When the controlling site receives "Commit ACK" message from all the slaves, it considers the transaction as committed.

- After the controlling site has received the first "Not Ready" message from any slave –
  - The controlling site sends a "Global Abort" message to the slaves.
  - The slaves abort the transaction and send a "Abort ACK" message to the controlling site.
  - When the controlling site receives "Abort ACK" message from all the slaves, it considers the transaction as aborted.

Distributed Three-phase Commit

The steps in distributed three-phase commit are as follows –

Phase 1: Prepare Phase

The steps are same as in distributed two-phase commit.

Phase 2: Prepare to Commit Phase

- The controlling site issues an "Enter Prepared State" broadcast message.
- The slave sites vote "OK" in response.

Phase 3: Commit / Abort Phase

The steps are same as two-phase commit except that "Commit ACK"/"Abort ACK" message is not required.

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 transaction executed. 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:

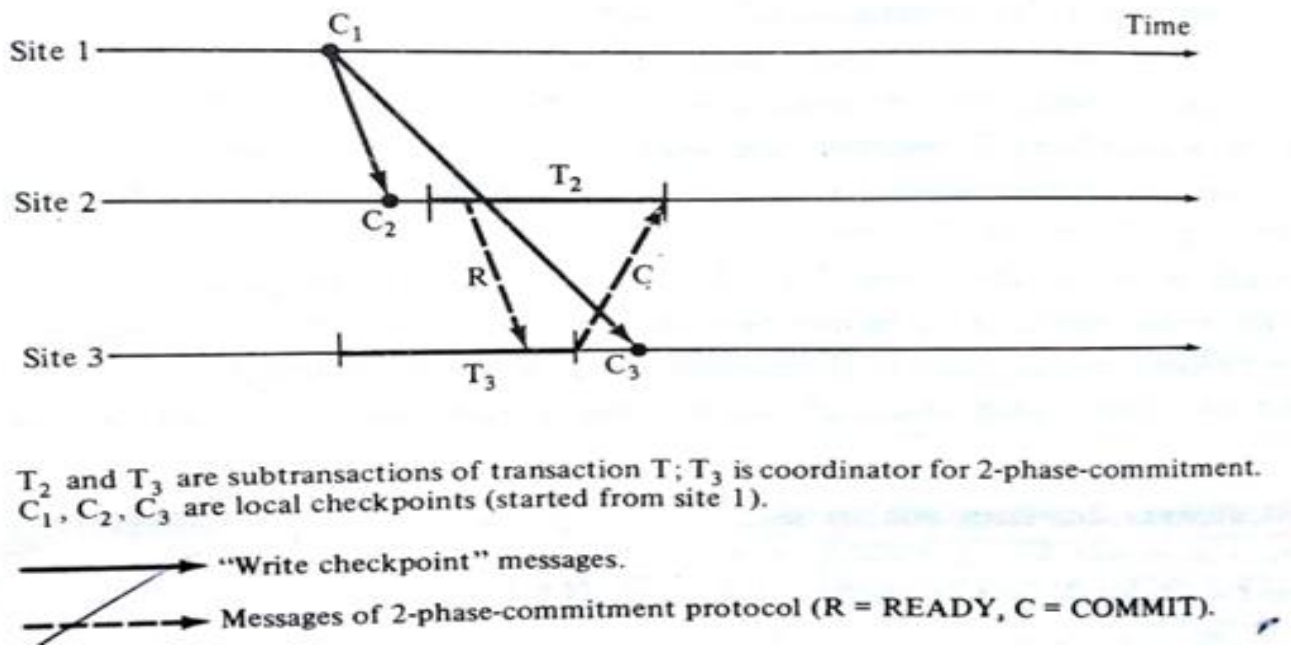
For each transaction T, C contains the updates performed by all subtransactions of T at any site, of it does not contain any of them; in the former case we say that T is contained in C.

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 sub transaction of a transaction T is contained in the local checkpoint at some site, then all other sub transactions 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.



Synchronization problem for global check point

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:

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 sub transaction 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 sub transaction 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 sub transaction to wait; observing the second condition can cause transaction aborts and restarts.

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.

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 sub transactions 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 sub transactions T'_i and T_j'be sub transactions of T'. If at site i the checkpoint of sub transaction T_i proceeds the checkpoint of T_J should precede the checkpoint of sub transaction 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:

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.

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:

Translating applications - Data referenced by applications at different levels of transparency are mapped to physical data (physical images in our reference architecture).

Optimizing applications - Data allocation, access methods available at each site, and statistical information (recorded in the catalogs) are required for producing an access plan.

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.

CONTENT OF CATALOGS

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

Global schema description -It includes the name of global relations and of attributes.

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.

Allocation description - It gives the mapping between fragments and physical images.

Mappings to local names -It is used for binding the names of physical images to the names of local data stored at each site.

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.

Statistics on the database - They include the profiles of the database,

co**nsistency 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:

Assessing the rights of users to perform specific actions on data. The typical actions considered are: read, insert, delete, update, and move.

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.

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,

The centralized approach

The full replication of items 1, 2, and 3 of catalog content and the local allocation of remaining items

The full replication of items 1, 2, 3, 4, and 5 of catalog contentand the local allocation of remaining items

The full replication of all items 5 of catalog content.

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.

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.

Data definition should be performed locally.

Different users should be able, independently, to give the same name to different data.

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:

Systemwide names are unique names given to each object in the system.

They have four components:

The identifier of the user who creates the object

The site of that user

The object name

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.
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 missing user identifier is replaced by the identifier of the current user.
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:

Global replication of a catalog is unacceptable, since this would violate the possibility of autonomous data definition.

No site should be required to maintain catalog information of objects which are not stored or created there.

The name resolution should not require a random search of catalog entries in the network.

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:

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.

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

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:

At the other side of the communication line is the intended site (and not an intruder).

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.

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.

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.

N 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.

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:

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.

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).

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:

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.

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.