UNIT-VI

UNIT-VI: Transactions & Replications: Introduction, System Model and Group Communication, Concurrency Control in Distributed Transactions, Distributed Dead Locks, Transaction Recovery; Replication-Introduction, Passive (Primary) Replication, Active Replication.

Introduction:

Transaction: A transaction is any kind of action involved in conducting business, or an interaction between people. When you go to the bank, fill out a form, and deposit your pay check, you make a *transaction*.

An important business deal can be called a transaction, particularly the buying or selling of goods, but you can call any exchange with another person a transaction. There are transactions involving money, ideas, and even e-mail.

Distributed Transaction: A **distributed transaction** is a database transaction in which two or more network hosts are involved. Usually, hosts provide **transactional resources**, while the **transaction manager** is responsible for creating and managing a global transaction that encompasses all operations against such resources. Distributed transactions, as any other transactions, must have all four ACID (atomicity, consistency, isolation, durability) properties, where atomicity guarantees all-or-nothing outcomes for the unit of work (operations bundle).

Databases are common transactional resources and, often, transactions span a couple of such databases. In this case, a distributed transaction can be seen as a database transaction that must be synchronized (or provide ACID properties) among multiple participating databases which are distributed among different physical locations. The isolation property (the I of ACID) poses a special challenge for multi database transactions, since the (global) serializability property could be violated, even if each database provides it (see also global serializability). In practice most commercial database systems use strong strict two phase locking (SS2PL) for concurrency control, which ensures global

serializability, if all the participating databases employ it. (see also commitment ordering for multidatabases.)

A common algorithm for ensuring correct completion of a distributed transaction is the two-phase commit (2PC). This algorithm is usually applied for updates able to commit in a short period of time, ranging from couple of milliseconds to couple of minutes.

There are also long-lived distributed transactions, for example a transaction to book a trip, which consists of booking a flight, a rental car and a hotel. Since booking the flight might take up to a day to get a confirmation, two-phase commit is not applicable here, it will lock the resources for this long. In this case more sophisticated techniques that involve multiple undo levels are used. The way you can undo the hotel booking by calling a desk and cancelling the reservation, a system can be designed to undo certain operations (unless they are irreversibly finished).

Replication: Replication is the process of making a replica (a copy) of something. A replication is a copy. The term is used in fields as varied as microbiology, knitwear (replication of knitting patterns), and information distribution (CD-ROM replication).

On the Internet, a Web site that has been replicated in its entirety and put on another site is called a mirror site.

Using the groupware product, Lotus Notes, replication is the periodic electronic refreshing (copying) of a database from one computer server to another so that all users in the Notes network constantly share the same level of information.

System Model and Group Communication:

System Models: An architectural **model** of a **distributed system** defines the way in which the components of the **system** interact with each other and the way in which they are mapped onto an underlying network of computers. E.gs. include the client-server **model** and the peer process **model**.

Distributed System Models is as follows:

1. Architectural Models

2. Interaction Models
3. Fault Models

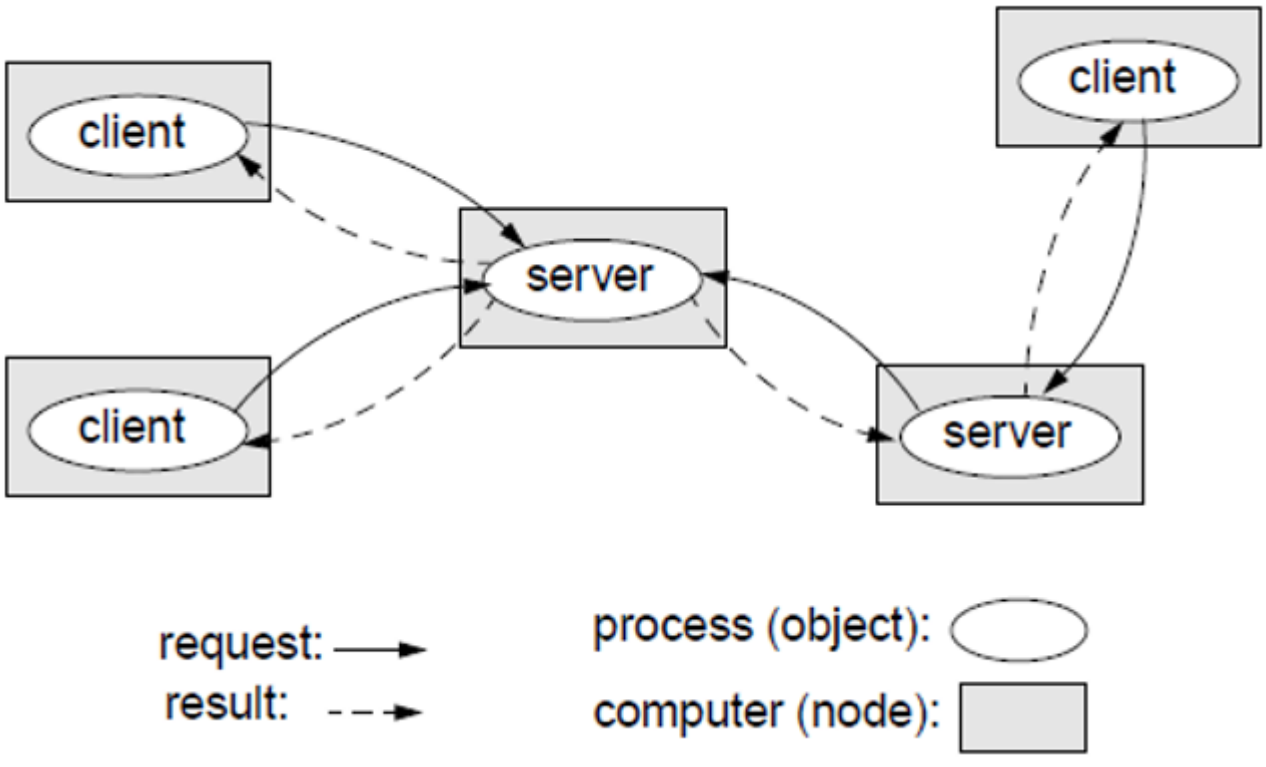
1. Architectural Models

Architectural model describes responsibilities distributed between system components and how are these components placed.

a) Client-server model

☞ The system is structured as a set of processes, called servers, that offer services to the users, called clients.

- The client-server model is usually based on a simple request/reply protocol, implemented with send/receive primitives or using remote procedure calls (RPC) or remote method invocation (RMI):
- The client sends a request (invocation) message to the server asking for some service;
- The server does the work and returns a result (e.g. the data requested) or an error code if the work could not be performed.

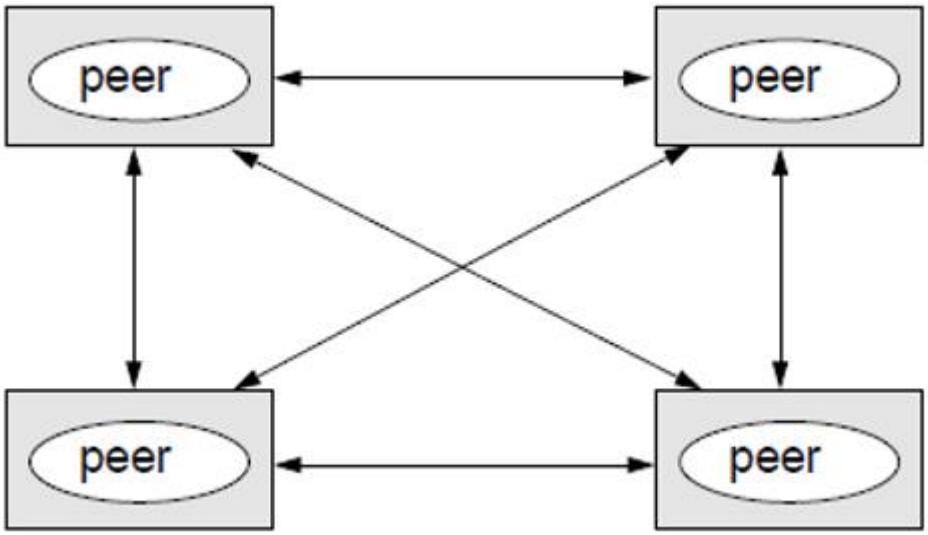


A server can itself request services from other servers; thus, in this new relation, the server itself acts like a client.

b) Peer-to-peer

☞ All processes (objects) play similar role.

- Processes (objects) interact without particular distinction between clients and servers.
- The pattern of communication depends on the particular application.
- A large number of data objects are shared; any individual computer holds only a small part of the application database.
- Processing and communication loads for access to objects are distributed across many computers and access links.
- This is the most general and flexible model.



- Peer-to-Peer tries to solve some of the above
- It distributes shared resources widely -> share computing and communication loads.

☞ Problems with peer-to-peer:

- High complexity due to
  - Cleverly place individual objects
  - retrieve the objects
  - maintain potentially large number of replicas.

2.Interaction Model

Interaction model are for handling time i. e. for process execution, message delivery, clock drifts etc.

- Synchronous distributed systems

Main features:

- Lower and upper bounds on execution time of processes can be set.
- Transmitted messages are received within a known bounded time.
- Drift rates between local clocks have a known bound.

Important consequences:

1. In a synchronous distributed system, there is a notion of global physical time (with a known relative precision depending on the drift rate).
2. Only synchronous distributed systems have a predictable behaviour in terms of timing. Only such systems can be used for hard real-time applications.
3. In a synchronous distributed system, it is possible and safe to use timeouts in order to detect failures of a process or communication link.

☞ It is difficult and costly to implement synchronous distributed systems.

- Asynchronous distributed systems

☞ Many distributed systems (including those on the Internet) are asynchronous. - No bound-on process execution time (nothing can be assumed about speed, load, and reliability of computers). - No bound-on message transmission delays (nothing can be assumed about speed, load, and reliability of interconnections) - No bounds on drift rates between local clocks.

Important consequences:

1. In an asynchronous distributed system, there is no global physical time. Reasoning can be only in terms of logical time (see lecture on time and state).
2. Asynchronous distributed systems are unpredictable in terms of timing.
3. No timeouts can be used.

☞ Asynchronous systems are widely and successfully used in practice.

In practice timeouts are used with asynchronous systems for failure detection.

However, additional measures have to be applied in order to avoid duplicated messages, duplicated execution of operations, etc.

3. Fault Models

☞ Failures can occur both in processes and communication channels. The reason can be both software and hardware faults.

☞ Fault models are needed in order to build systems with predictable behavior in case of faults (systems which are fault tolerant).

☞ such a system will function according to the predictions, only as long as the real faults behave as defined by the "fault model".

Group Communication: A group is a collection of processes that act together in some system or user specified way.

• The key property that all groups have that when a message is sent to the group; itself all the members of the group receive it.

• It is a form of one to many communications.

1) One – to – many:

a. Group management

b. Group Addressing

c. Message delivery to receiving process

d. Atomic multicast

e. Group communication primitives

• Single sender and multiple receiver are also known as multicast communication

a. Group Management

• In case of group communication, the communicating processes forms a group.

• Such a group may of either of 2 types.

• Closed group is the one, in which only member can send message, outside process cannot send message to the group as whole but can send to single member of a group.

• Open group is one in which any process in the system can send the message to group as a whole.

b. Group Addressing

• Two level naming schemes is used for group addressing.

• Multicast address is a special n/w address, packet is delivered automatically here to all m/cs listening to address.

• It broadcast address the all m/cs has to check if packet is intended for it or else simply discard so broadcast is less efficient.

• If network doesn't support any addressing among two then one to one communication is used.

• First two methods send single packet but one to one sends many, creating much traffic.

c. Message delivery to receiving process

• User application uses high level group names in program.

• Centralized group server maintains a mapping of high-level group names to their low level names, when sender sends message to the group. With high level name, kernel of server m/c asks the group servers low level name and list of process identifiers.

• When packet reaches m/c kernel, that m/c extract list of process IDs and forwards message to those processes belonging to its own m/c.

• If none of ID's is matching packet is discarded.

d. Atomic Multicast

• All or nothing property all reliable form requires atomic multicast facility.

• Message passing system should support both atomic and non- atomic multicast facility.

• It should provide flexibility to sender to specify whether atomicity is required or not.

• It is difficult to implement atomic multicast if sender or receiver fails.

• Solution to this is fault tolerated atomic multicast protocol.

• In this protocol, each message has message identifier field to distinguish message and one field to indicate data message in atomic multicast message.

e. Group communication primitives

• In one to one and one to many send or has to specify destination address and pointer to data so some send primitives can be used for both.

• But some systems use send group primitives because

1) It simplifies design and implementation

2) It provides greater flexibilities.

2) Many – to – one:

• Multiple sender single receiver

• Single receiver may be selective or non-selective

• Selective receiver specifies unique sender message exchange takes place only if the sender sends the message.

• Non-selective receiver specifies set of senders if any from that sends message then message exchange takes place.

• No determinism issue needs to be handled here.

• Rest all factors are same as for one – to – many communications.

3) Many –to – many:

• One to many, many to one, all issues are induced in this.

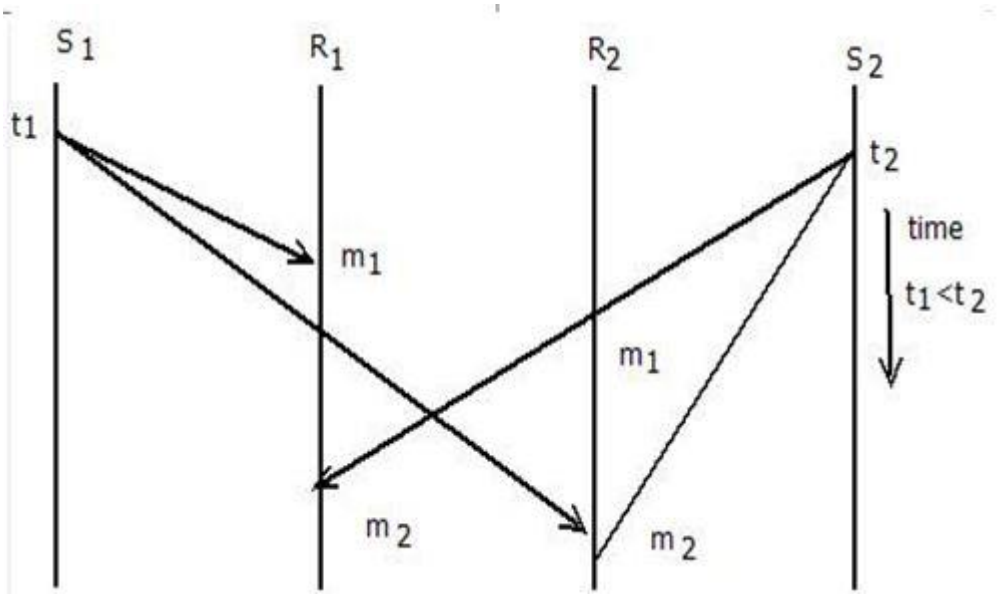
• Ordered message delivery/ event ordering.

1) Absolute ordering

2) Consistent ordering

3) Casual

1) Absolute Ordering



• It ensures that all message delivered to all receiver in exact in which they are sent.

• Used to implement is to use global timestamps as message id.

• System is assumed to have clock synchronization and when sender message timestamps is taken as id of message and embedded in message.

• Kernel of receiver m/c, saves all incoming message in separate queue.

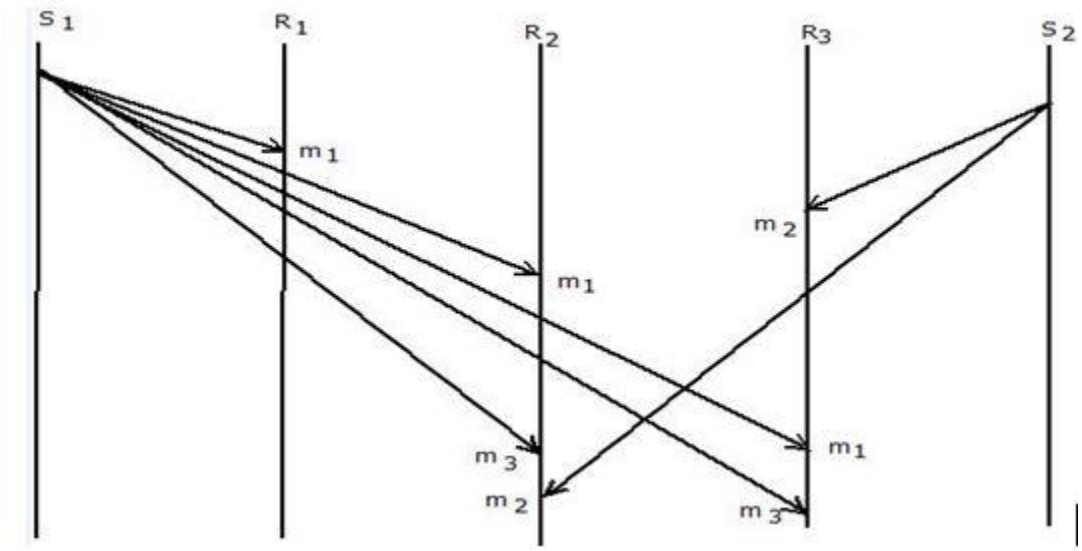
• A sliding window is used to periodically deliver message from receiver i.e fixed time interval is selected as window size.

• Window size and properly chosen by considering maximum time for message to kernel from one m/c to other in n/w.

2) Casual Ordering

• For some application weaker semantics acceptable.

• Such semantic is called casual ordering.



• This semantics ensures that is the sending one message is casually related to event of sending another message, two messages are delivered to all receiver in correct order, otherwise these may be delivered in any order.

• Two message sending events are said to be casually related if they are correlated by happened before relation.

• Two message sending events are casually related if there are any possibilities of second one being influenced in a way by first one.

Concurrency Control in Distributed Systems:

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

In this chapter, we will study the various approaches for concurrency control.

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.

a) 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.

b) Two-phase Locking Protocol

In this method, all locking operations precede the first lock-release or unlock operation. The transaction comprises 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.
- Multiversion 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.

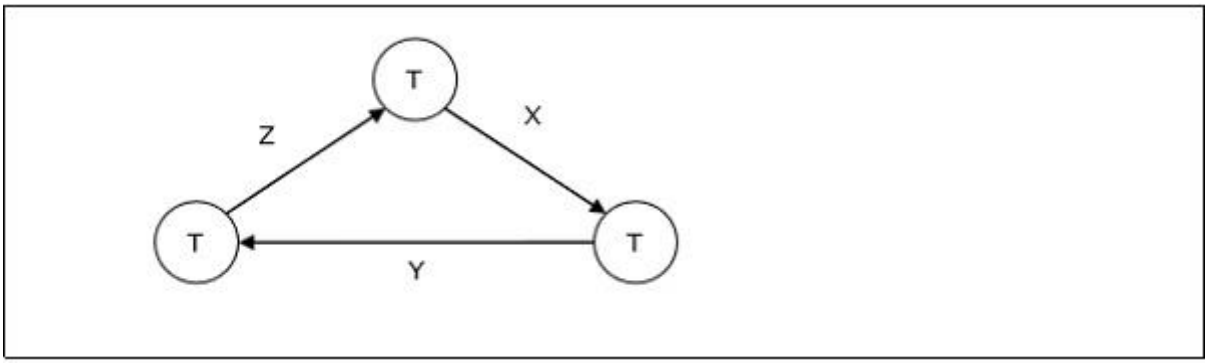
Rule 3 − Given two transactions T_i and T_j, if T_i is writing the data item which T_j is also writing, then T_i's commit phase cannot overlap with T_j's commit phase. T_j can start to commit only after T_i has already committed.

Distributed Dead Locks:

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 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.

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 centre of control, like:

- One server may be selected as the centre of control.
- The centre of control may travel from one server to another.
- The responsibility of controlling may be shared by a number of servers.

1) Distributed 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.

2) Distributed Deadlock Avoidance

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.

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.

3) **Distributed Deadlock Detection**

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.

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.

Transaction Recovery:

Transaction recovery is done to eliminate the adverse effects of faulty transa ctions 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.

Types of Transaction Recovery

Recovery information is divided into two types:

- Undo (or Rollback) Operations
- Redo (or Cache Restore) Operations

Ingres performs both online and offline recovery, as described in Recovery Modes (see page Recovery Modes).

Undo Operation:

Undo or transaction backout recovery is performed by the DBMS Server. For example, when a transaction is aborted, transaction log file information is used to roll back all related updates. The DBMS Server writes the Compensation Log Records (CLRs) to record a history of the actions taken during undo operations.

Redo Operation

A Redo recovery operation is database-oriented. Redo recovery is performed after a server or an installation fails. Its main purpose is to recover the contents of the DMF cached data pages that are lost when a fast-commit server fails. Redo recovery is performed by the recovery process. Redo recovery precedes undo recovery.

Redo Operation in a Cluster Environment

In an Ingres cluster environment where all nodes are active, the local recovery server performs transaction redo/undo for a failed DBMS server on its node, just like in the non-cluster case. The difference in a cluster installation is that if the recovery process (RCP) dies on one node, either because of an Ingres failure, or a general failure of the hardware, an RCP on another node will take responsibility for cleaning up transactions for the failed nodes.

Distributed Replication:

Introduction:

In the distributed systems research area replication is mainly used to provide fault tolerance. The entity being replicated is a process. ... Deterministic means that, given the same initial state and a request sequence, all processes will produce the same response sequence and end up in the same final state.

Replication: Replication in computing involves sharing information so as to ensure consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.

Replication involves storing copies of data at multiple computers. There are several benefits of using this technique:

- Performance enhancement - more load (e.g., client requests) can be tolerated because workload is shared among several processes. In addition, latency can be reduced by replicating data closer to the user. Unfortunately, benefits are reduced if data is read/write.
- Increased availability - replication helps a service to overcome individual server failures. If a server will fail with probability p, the number of servers needed to provide a given level of service is Avail = $1-p^n$.
- Fault tolerance - similar to availability, but ensures correctness in addition to availability. For example, if one server of a group of n servers provides bad information, the others can outvote the incorrect server and provide correct data to the client.

Computer scientists further describe replication as being either:

- **Active replication:** which is performed by processing the same request at every replica
- **Passive replication:** which involves processing every request on a single replica and transferring the result to the other replicas

Passive(primary) Replication:

In the passive replication model, front ends interact with a single, primary replica manager. The primary replica manager responds to requests, and sends updates to

several secondary replica managers. In the event that the primary fails, a secondary replica manager can take its place. The sequence of events is as follows:

1. Request: The front end issues the request, containing a unique identifier, to the primary replica manager.
2. Coordination: The primary takes each request atomically, in the order in which receives it. It checks the unique identifier; in case it has already executed the request and if so it simply re-sends the response.
3. Execution: The primary executes the request and stores the response.
4. Agreement: If the request is an update then the primary sends the updated state, the response, and the unique identifier to all the backups. The backups send an acknowledgement.
5. Response: The primary responds to the front end, which hands the response back to the client.

We communicate updates while ensuring that we can tolerate a failure of the primary replica before, during, and after updating? This is a group communication problem. However, we need the ability to manage group membership so that we can be certain that everyone has received all updates.

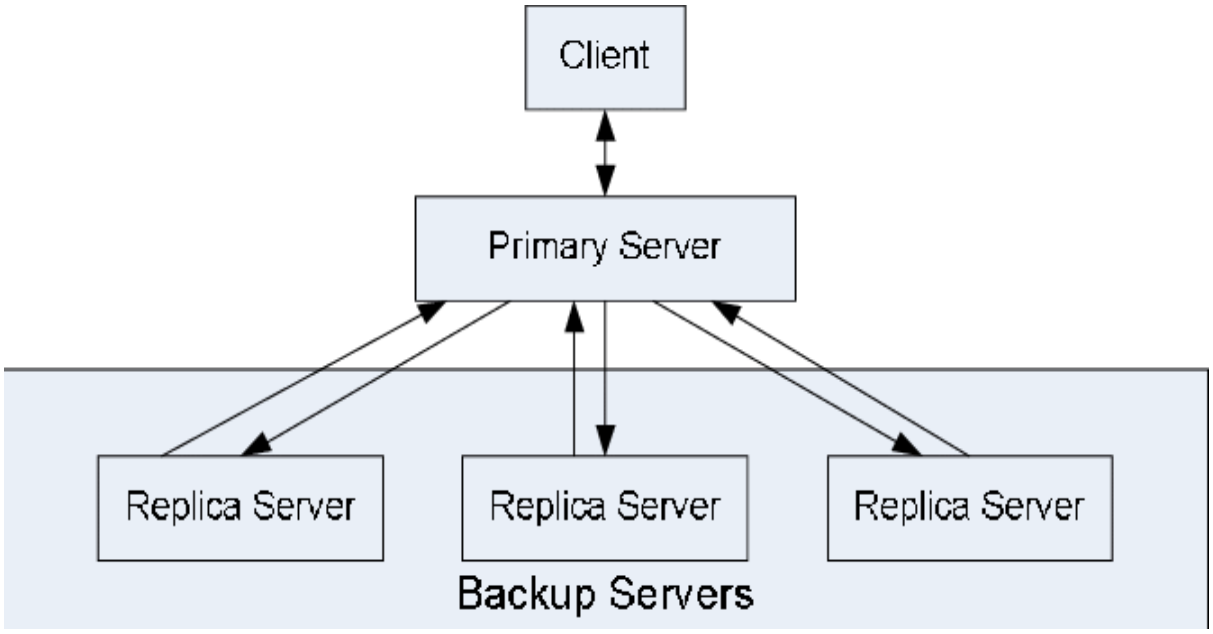


Fig: Passive Replication

A group membership process must:

- Provide an interface for membership changes.
- Provide a failure detector.
- Notify members of membership changes.
- Perform address expansion to ensure that messages sent to the group reach all replicas.

Using this service, we can provide view-synchronous communication. View-synchronous communication is an extension of reliable multicast. It uses a view, which is a list of the processes currently belonging to the group. When membership changes, a new view is sent to all members. All messages that originate in a given view must be delivered before a new view is delivered. It is a bit like a cut in the message timeline.

Implementing view-synchronous communication is costly and requires several rounds of communication for each multicast. This can also lead to delays in responding the client -- a clear disadvantage of passive replication. However, passive replication can tolerate n crash failures if n+1 replica is present. It is also very easy from the front end's point of view since it only communicates with a single server.

Finally, because all replicas have the same record of updates, it is easy for a secondary replicato take over a failed primary replica.

Active Replication:

In the active replication model, a front-end multicast a request to all replicas. The sequence of events is as follows:

1. Request: The front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. The front end is assumed to fail by crashing at worst. It does not issue the next request until it has received a response.
2. Coordination: The group communication system delivers the request to every correct replica manager in the same (total) order.
3. Execution: Every replica manager executes the request. Since they are state machines and since requests are delivered int he same total order, correct replica

managers all process the request identically. The response contains the client's unique request identifier.

4. Agreement: No agreement phase is needed, because of the multicast delivery semantics.
5. Response: Each replica manager sends it response to the front end. The number of replies that the front-end collects depends upon the failure assumptions and on the multicast algorithm. If, for example, the goal is to tolerate only crash failures and the multicast satisfies uniform agreement and ordering properties, then the front end passes the first response to arrive back to the client and discards the rest.

If the goal of the system is to tolerate byzantine failures, to tolerate f failures the system must use 2f+1 replica and the front end must wait until it collects f+1 identical response.

Implementing totally ordered, reliable multicast is equivalent to the consensus problem. It cannot be done in an asynchronous system unless we use a technique such as failure detectors. In addition, passive replication may be slow from the client's point of view.

Fig: Active Replication