## What is Replication in Distributed System?

In a distributed system data is stored is over different computers in a network. Therefore, we need to make sure that data is readily available for the users. **Availability** of the data is an important factor often accomplished by data replication. ***Replication is the practice of keeping several copies of data in different places.***

**Why do we require replication?**

The first and foremost thing is that it makes our system more stable because of node replication. It is good to have replicas of a node in a network due to following reasons:

•If a node stops working, the distributed network will still work fine due to its replicas which will be there. Thus it increases the fault tolerance of the system.

•It also helps in load sharing where loads on a server are shared among different replicas.

•It enhances the availability of the data. If the replicas are created and data is stored near to the consumers, it would be easier and faster to fetch data.

**Types of Replication**

•Active Replication

•Passive Replication

**Active Replication:**

•The request of the client goes to all the replicas.

•It is to be made sure that every replica receives the client request in the same order else the system will get inconsistent.

•There is no need for coordination because each copy processes the same request in the same sequence.

•All replicas respond to the client’s request.

**Advantages:**

•It is really simple. The codes in active replication are the same throughout.

•It is transparent.

•Even if a node fails, it will be easily handled by replicas of that node.

**Disadvantages:**

•It increases resource consumption. The greater the number of replicas, the greater the memory needed.

It increases the time complexity. If some change is done on one replica it should also be done in all others.

**Passive Replication:**

- The client request goes to the primary replica, also called the main replica.
- There are more replicas that act as backup for the primary replica.
- Primary replica informs all other backup replicas about any modification done.
- The response is returned to the client by a primary replica.
- Periodically primary replica sends some signal to backup replicas to let them know that it is working perfectly fine.
- In case of failure of a primary replica, a backup replica becomes the primary replica.

**Advantages:**

- The resource consumption is less as backup servers only come into play when the primary server fails.
- The time complexity of this is also less as there’s no need for updating in all the nodes replicas, unlike active replication.

**Disadvantages:**

- If some failure occurs, the response time is delayed.

## The Gossip Architecture

- The gossip architecture is a framework for providing a highly available service which replicates data close to the points where groups of clients request it.
- RMs exchange "**gossip**" messages periodically in order to convey the updates they have each received from clients.
- A gossip service provides two basic types of operation:

  **queries** are read-only operations; and
  **updates** modify but do not read the state.
- A front end can send a query or update to any RM that it chooses (based on availability and response time).
- RM may be temporarily unable to communicate with each other due to failures. Nevertheless the system guarantees a form of consistency called **relaxed consistency**.

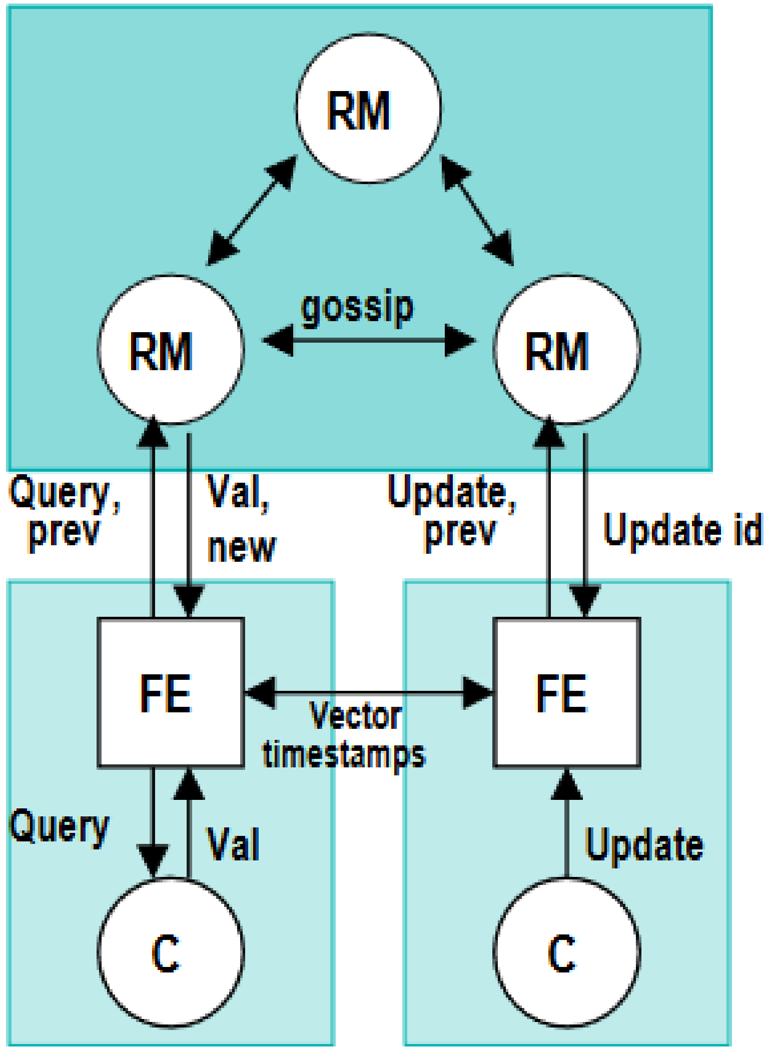
## Gossip System Guarantees

**Each client obtains a consistent service over time:** Responding to a query a RM provides a client with data that reflects at least the updates that client has observed so far.

Note that this is guaranteed even though a client may communicate with different RM at different times.

**Relaxed consistency between replicas:** All RM eventually receive all updates and they apply updates with ordering guarantees that make the replicas sufficiently similar to meet the needs of the application.

This is weaker than sequential consistency and may mean that clients observe stale data. It is supported by **causal update ordering**: if the issue of request $r$ *happened-before* the issue of request $r'$ then any correct RM handles $r$ before $r'$. However stronger ordering conditions may be applied at higher operational costs.



The gossip service front end handles client operations using an application-specific API and turns them into gossip operations (queries or updates). RM updates are **lazy** in the sense that gossip messages may be exchanged only occasionally. Each front end keeps a **vector timestamp** *prev*, reflecting the latest data values accessed by the client/front end. When clients communicate directly they piggyback their vector timestamps, which are then merged.

## Processing requests

1. *Request:* The front end sends the request to a RM. Queries are usually synchronous but updates are asynchronous: the client continues as soon as the request is passed to the front end and the front end propagates the request in background.
2. *Update response:* The RM replies to the front end as soon as it has received an update.
3. *Coordination:* The RM that receives a request does not process it until it can apply the request according to the required ordering constraints. This may involve receiving updates from other replica managers, in gossip messages.
4. *Execution:* The RM executes the request.
5. *Query response:* If the request is a query the RM responds at this point.
6. *Agreement:* The RMs only coordinate via gossip messages.

## Gossip Replica Manager: main components

**Value** This records the application state as maintained by this RM.

**Value timestamp** This represents the updates currently reflected in the value.

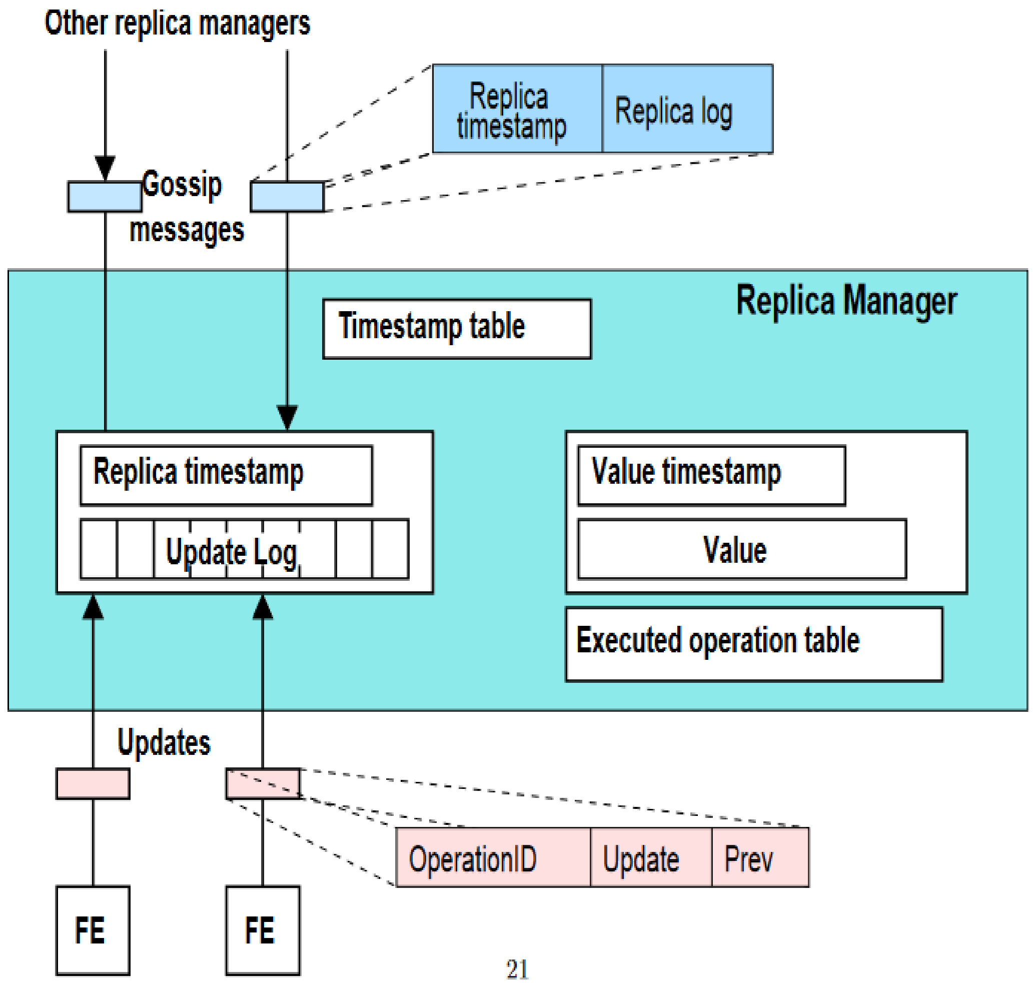
**Update log** All update operations are recorded in the log as soon as they arrive although they may be applied in a different order. An update is termed **stable** if it can be applied consistently with the desired ordering. An arriving update is stored until it is stable and can be applied. It then remains in the log until the RM receives confirmation that all other RMs have received the update.

**Replica timestamp** This timestamp represents those updates that have been accepted by the RM (not necessarily applied).

**Executed operation table** Maintained to prevent an update being applied twice and checked before an update is added to the log.

**Timestamp table** Contains a vector timestamp for each other RM, derived from gossip messages.

### Gossip Replica Manager



In a gossip system, a front end has a timestamp (3,5,7) representing the data it has received from members of a group of three replica managers. The tree replica managers have vector timestamps (4,2,8), (4,5,6) and (4,5,8), respectively. Which replica managers could immediately satisfy a query from the front end, and what would the resultant timestamp of the front end be? Which could incorporate an update from the front end immediately?

Answer: The only replica manager that can satisfy a query from this front end is the third, with timestamp (4,5,8). The others have not yet processed at least one update seen by the front end.
The resultant time stamp of the front end will be (4,5,8).
Similarly, only the third replica manager could incorporate an update from the front-end immediately

Given:

- Front end timestamp: (3, 5, 7)
- Replica Manager 1 timestamp: (4, 2, 8)
- Replica Manager 2 timestamp: (4, 5, 6)
- Replica Manager 3 timestamp: (4, 5, 8)

1. **Querying**:
    1. To satisfy a query from the front end, the front end timestamp must be less than or equal to the vector timestamp of at least one replica manager for each corresponding component.
    2. Comparing the front end timestamp with each replica manager's timestamp:
        1. Front end timestamp: (3, 5, 7)
        2. Replica Manager 1: (4, 2, 8)
        3. Replica Manager 2: (4, 5, 6)
        4. Replica Manager 3: (4, 5, 8)
    3. The front end timestamp (3, 5, 7) is less than or equal to the timestamp of Replica Manager 3 (4, 5, 8) for all components.
    4. Therefore, only Replica Manager 3 can immediately satisfy a query from the front end.
    5. The resultant timestamp of the front end after the query would be (4, 5, 8).

**2.Updating**:

2. To incorporate an update from the front end immediately, the front end timestamp must be greater than or equal to the vector timestamp of at least one replica manager for each corresponding component.
3. Comparing the front end timestamp with each replica manager's timestamp:
    2. Front end timestamp: (3, 5, 7)
    3. Replica Manager 1: (4, 2, 8)
    4. Replica Manager 2: (4, 5, 6)
    5. Replica Manager 3: (4, 5, 8)
4. The front end timestamp (3, 5, 7) is less than the timestamp of all replica managers for all components.
5. Therefore, none of the replica managers can immediately incorporate an update from the front end.

In summary:

•Only Replica Manager 3 can immediately satisfy a query from the front end, and the resultant timestamp of the front end would be (4, 5, 8) after the query.

•None of the replica managers could immediately incorporate an update from the front end.

Why Gossip-based system is not appropriate for updating replicas in near-real time? Provide an alternative
approach.
Answer: Due to lazy approach to update propagation, a gossip-based system is inappropriate for up-dating replicas with near-real time. A multicast-based system will be more appropriate for this case.

In a replication system, the total number of servers is 4. 2 servers have an independent probability p = 0.3 of failing each, the 3rd server has p = 0.5 and the last one has p = 1. What is the availability of an object stored at each of these servers?
Availability in a replication system refers to the probability that an object can be accessed despite failures of some of the servers.
The expression (1 − p1 × p2 × p3 × p4) represents the probability that at least one server among four servers remains operational, given their individual failure probabilities p1, p2, p3, and p4.
- p1=0.3 (probability of failure for Server 1 and Server 2)
- p2=0.3p2=0.3 (probability of failure for Server 1 and Server 2)
- p3=0.5p3=0.5 (probability of failure for Server 3)
- p4=1p4=1 (probability of failure for Server 4)

Let's calculate the availability using the given expression:

Availability $=1-p1 \times p2 \times p3 \times p4$
Substituting the given values:
Availability $=1-(0.3 \times 0.3 \times 0.5 \times 1)$
Availability $=1-(0.045)$
Availability $=0.955$
So, the availability of the system, considering the failure probabilities of individual servers, is approximately 0.955 or 95.5%. This means there's a 95.5% probability that at least one server remains operational

3. In a multi-user game, the players move figures around a common scene. The state of the game is replicated at the players' workstations and at a server, which contains services controlling the game overall, such as collision detection. Updates are multicast to all replicas.
   i. The figures may throw projectiles at one another and a hit debilitates the unfortunate recipient for a limited time. What type of update ordering is required here? Hint: consider the 'throw', 'collide' and 'revive' events.
   ii. The game incorporates magic devices which may be picked up by a player to assist them. These devices are available to all players at first and they have to race to get them. What type of ordering should be applied to the pick-up-device operation?

ii. The event of the collision between the projectile and the figure, and the event of the player being debilitated (which, we may assume, is represented graphically) should occur in causal order.

Moreover, changes in the velocity of the figure and the projectile chasing it should be causally ordered.

Assume that the workstation at which the projectile was launched regularly announces the projectile's coordinates, and that the workstation of the player corresponding to the figure regularly announces the figure's coordinates and announces the figure's debilitation. These announcements should be processed in causal order.

iii. If two players move to pick up a piece at more-or-less the same time, only one should succeed and the identity of the successful player should be agreed at all workstations.

Therefore total ordering is required.

(FYI) The most promising architecture for a game such as this is a peer group of game processes, one at each player's workstation. This is the architecture most likely to meet the real-time update propagation requirements; it also is robust against the failure of any one workstation (assuming that at least two players are playing at the same time). (More details later on further lectures)