Replication

Jingzhu He

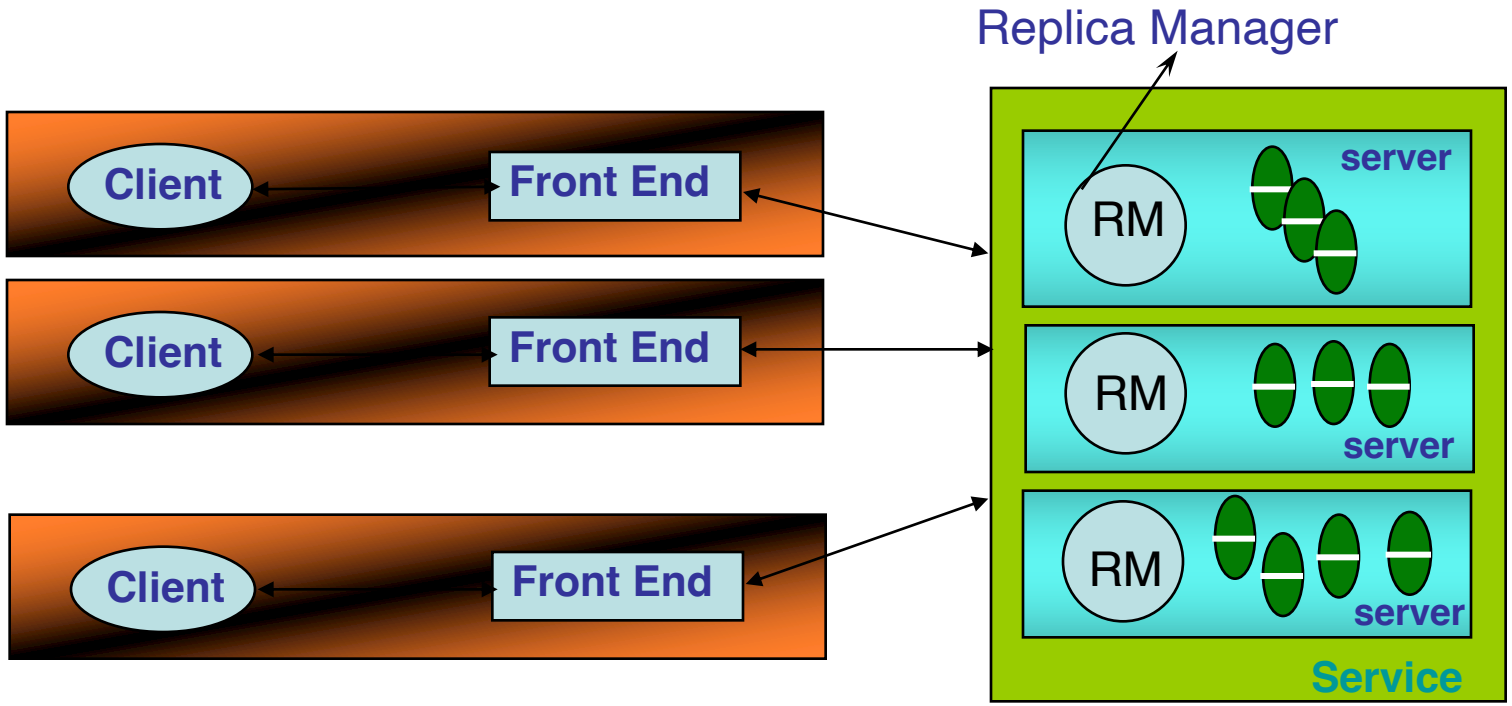
Why Replication?

- Load Balancing
  - Workload is shared between the replicated servers e.g., binding all the server IP addresses to the service's DNS name. A DNS lookup of the site results in one of the servers' IP addresses being returned, in a round-robin fashion.
- Fault Tolerance
  - Under the fail-stop model, if up to *f* of *f+1* servers crash, at least one remains to supply the service.
- Increased Availability
  - Service may not be available when servers fail or overloaded, when the network is partitioned.

P: probability that one server fails= 1 – P= availability of service. e.g. P = 5% => service is available 95% of the time.

P^n: probability that n servers fail= 1 – P^n= availability of service. e.g. P = 5%, n = 3 => service available 99.875% of the time

Basic Replication Mode



- Transparency
  - User/client need not know that multiple physical copies of data exist
- Consistency
  - Data is consistent on all of the replicas (or is in the process of becoming consistent)

Replication Management

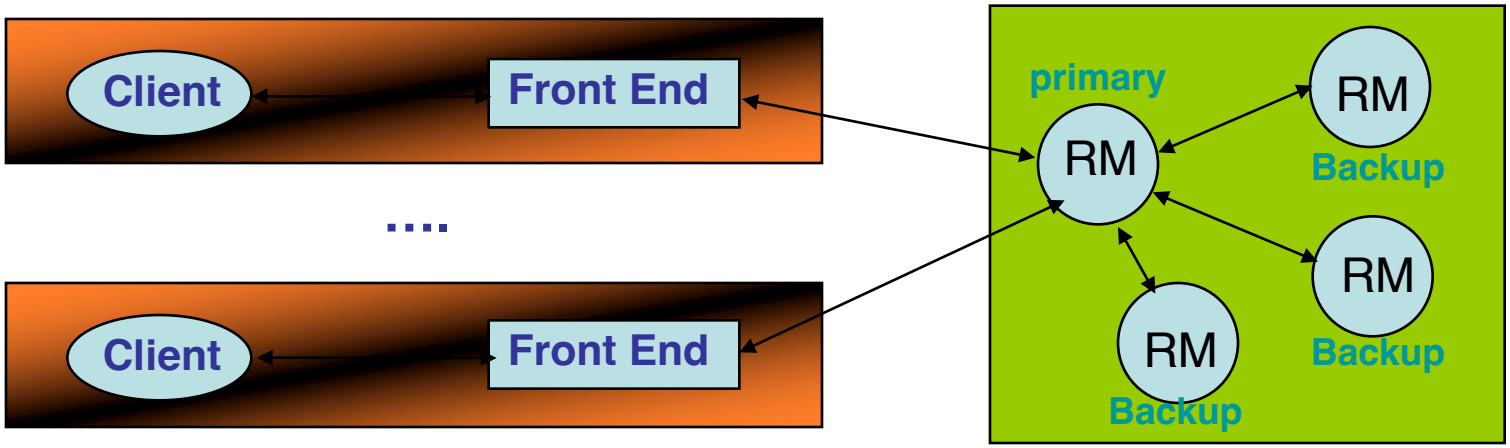
- Request Communication
  - Requests can be made to a single RM or to multiple RMs
- Coordination: The RMs decide
  - whether the request is to be applied
  - the order of requests
    - FIFO ordering: If a FE issues *r* then *r'*, then any correct RM handles *r* and then *r'*.
    - Causal ordering: If the issue of *r* "happened before" the issue of *r'*, then any correct RM handles *r* and then *r'*.
    - Total ordering: If a correct RM handles *r* and then *r'*, then any correct RM handles *r* and then *r'*.
- Execution
  - The RMs execute the request tentatively

Replication Management

- Agreement: The RMs attempt to reach consensus on the effect of the request.
  - E.g., Two phase commit through a coordinator
  - If this succeeds, effect of request is made permanent

- Response
  - One or more RMs responds to the front end.
  - In the case of fail-stop model, the FE returns the first response to arrive.

Passive (Primary-Backup) Replication

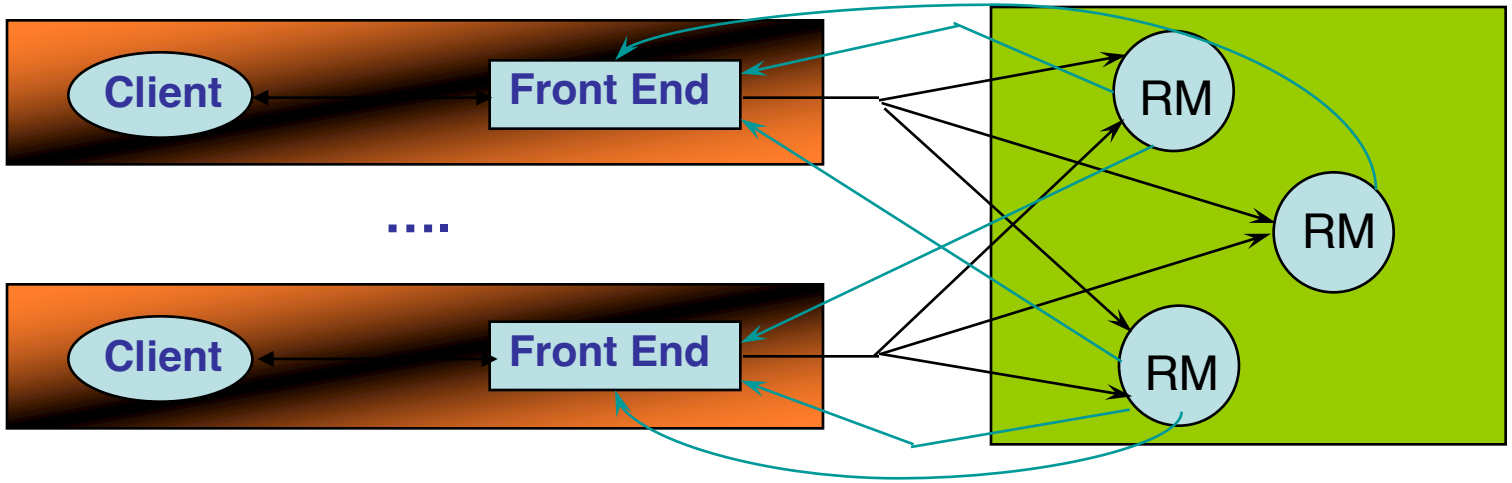


- Request Communication: the request is issued to the primary RM and carries a unique request id.
- Coordination: Primary takes requests atomically, in order, checks id (resends response if not new id.)
- Execution: Primary executes & stores the response
- Agreement: If update, primary sends updated state/result, req-id and response to all backup RMs (1-phase commit enough).
- Response: primary sends to the front end

Fault Tolerance in Passive Replication

- The system implements linearizability, since the primary sequences operations in order.
- If the primary fails, a backup becomes primary by leader election, and the replica managers that survive agree on which operations had been performed at the point when the new primary takes over.
  - The above requirement is met if the replica managers (primary and backups) are organized as a group and if the primary uses view-synchronous group communication to send updates to backups.
- The system remains linearizable after the primary crashes

Active Replication



- Request Communication: The request contains a unique identifier and is multicast to all by a reliable totally-ordered multicast.
- Coordination: Group communication ensures that requests are delivered to each RM in the same order (but may be at different physical times!).
- Execution: Each replica executes the request. (Correct replicas return same result since they are running the same program, i.e., they are *replicated protocols* or *replicated state machines*)
- Agreement: No agreement phase is needed, because of multicast delivery semantics of requests
- Response: Each replica sends response directly to FE

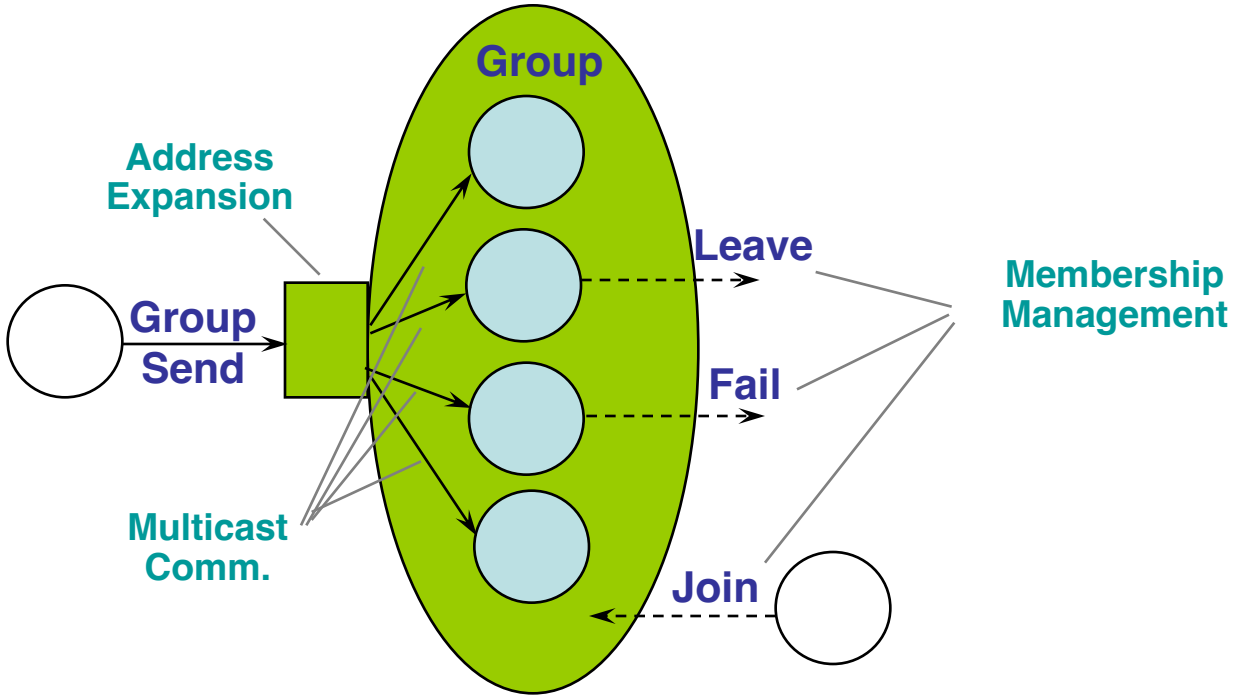
Fault Tolerance in Active Replication

- RMs work as replicated state machines, playing equivalent roles. That is, each responds to a given series of requests in the same way. This is achieved by running the same program code at all RMs.
- If any RM crashes, state is maintained by other correct RMs.
- This system implements sequential consistency
  - The total order ensures that all correct replica managers process the same set of requests in the same order.
  - Each front end's requests are served in FIFO order (because the front end awaits a response before making the next request).
- So, requests are FIFO-total ordered. But if clients are multi-threaded and communicate with one another while waiting for responses from the service, we may need to incorporate causal-total ordering

Eager versus Lazy

- Eager replication
  - Multicast request to all RMs immediately in active replication
  - Multicast results to all RMs immediately in passive replication
- Alternative: Lazy replication
  - Allow replicas to converge eventually and lazily
  - FEs need to wait for reply from only one RM
  - Allow other RMs to be disconnected/unavailable
  - Propagate updates and queries lazily
  - May provide weaker consistency than sequential consistency, but improves performance
  - Concepts also applicable in building disconnected file systems
- Lazy replication can be provided by using the gossip architecture

Group Communication



- Member: process (e.g., RM, server replica)
- Static Groups: group membership is pre-defined
- Dynamic Groups: members may join and leave

Views

- A *group membership service* maintains group *views*, which are lists of current group members.
  - This is NOT a list maintained by one member
  - *Each member maintains its own local view*
- A view $V_p(g)$ is process *p*'s understanding of its group (list of members)
  - Example: $V_{p.0}(g) = \{p\}$, $V_{p.1}(g) = \{p, q\}$, $V_{p.2}(g) = \{p, q, r\}$, $V_{p.3}(g) = \{p,r\}$
- A new group view is disseminated, throughout the group, whenever a member joins or leaves.
  - Member detecting failure of another member multicasts a "view change" message (requires causal-total ordering for multicasts)

Views

- An event is said to occur in a view $v_{p,i}(g)$ *if the event occurs at p, and at the time of event occurrence, p has delivered* $v_{p,i}(g)$ *but has not yet delivered* $v_{p,i+1}(g)$.

- Messages sent out in a view *i* need to be delivered in that view to *all* members in the group ("What happens in the View, stays in the View")

Views

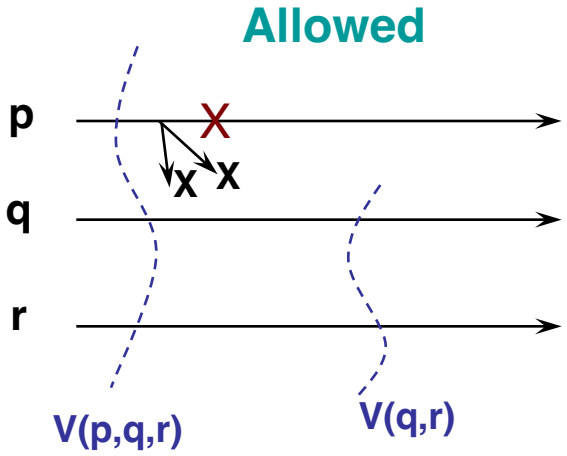
- Requirements for view delivery
  - Order: If *p* delivers $v_i(g)$ and then $v_{i+1}(g)$, then no other process *q* delivers $v_{i+1}(g)$ before $v_i(g)$.
  - Integrity: If *p* delivers $v_i(g)$, then *p* is in $v_i(g)$.
  - Non-triviality: if process q joins a group and becomes reachable from process p, then eventually, q will always be present in the views that delivered at p.
    - *Exception: partitioning of group. Solutions to partitioning:*
      - *Primary partition: allow only majority partition to proceed*
      - *Allow any and all partitions to proceed*
      - *Choice depends on consistency requirements.*
    - *Ignore partitions for the rest of the lecture.*

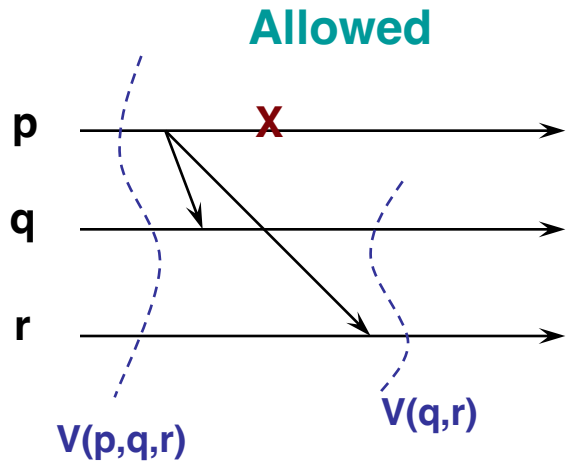
View Synchronous Communication

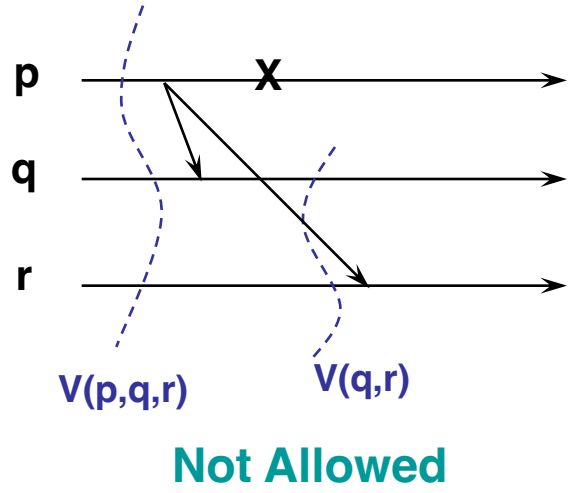
- View Synchronous Communication = Group Membership Service + Reliable multicast
- The following guarantees are provided for multicast messages:
  - Integrity: If *p* delivered message *m*, *p* will not deliver *m* again. Also $p \in group\ (m)$.
  - Validity: Correct processes always deliver all messages. That is, if p delivers message m in view *v(g)*, and some process q $\in$ *v(g)* does not deliver m in view *v(g),* then the next view *v'(g)* delivered at *p* will not include *q.*
  - Agreement: Correct processes deliver the same set of messages in any view.

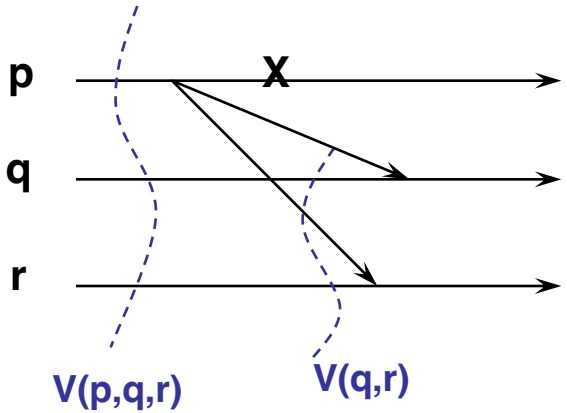
    if *p* delivers *m* in *V*, and then delivers *V'*, then all processes in $V \cap V'$ deliver *m* in view *V*
  - All View Delivery conditions (Order, Integrity and Non-triviality conditions, from last slide) are satisfied
- "What happens in the View, stays in the View"

Example: View Synchronous Communication





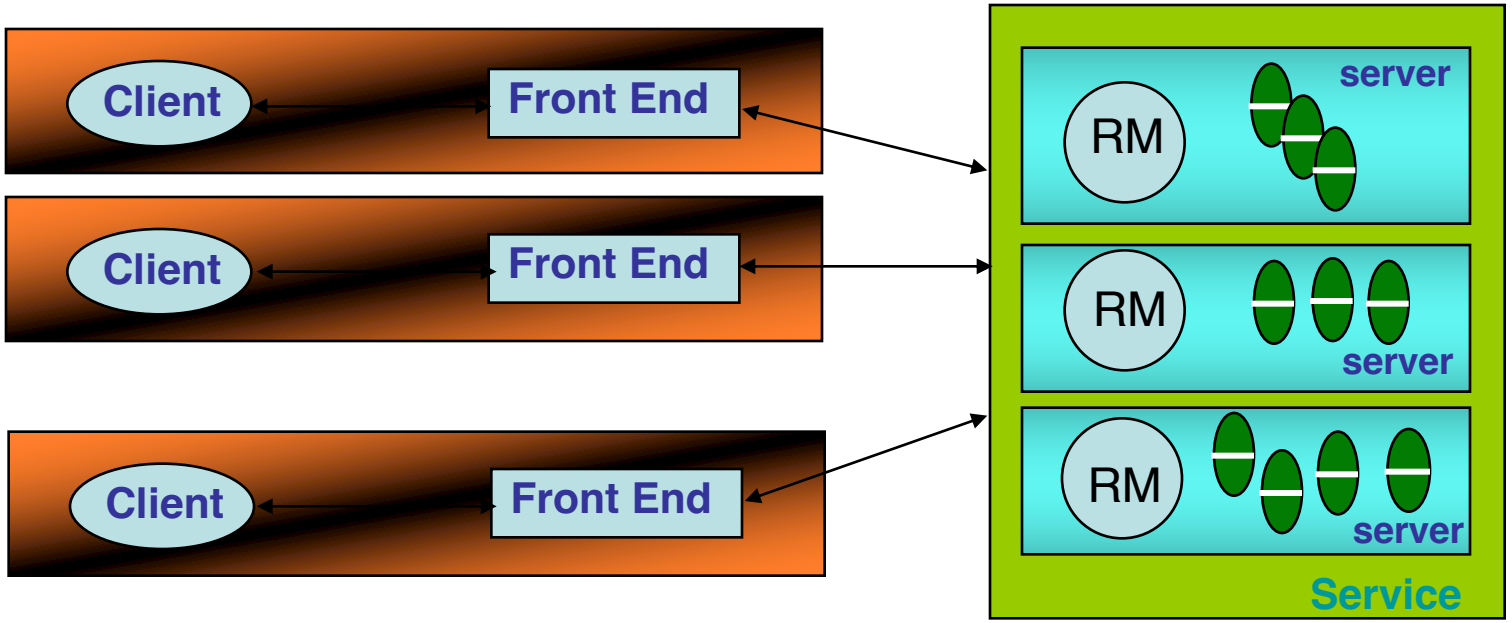




State Transfer

- When a new process joins the group, state transfer may be needed (at view delivery point) to bring it up to date
  - "state" may be list of all messages delivered so far (wasteful)
  - "state" could be list of current server object values (e.g., a bank database) – could be large
  - Important to optimize this state transfer
- View Synchrony = "Virtual Synchrony"
  - "Provides an abstraction of a synchronous network that hides the asynchrony of the underlying network from distributed applications
  - But does not violate FLP impossibility (since can partition)
- Used in ISIS toolkit (NY Stock Exchange)

Back to Replication



Need consistent updates to all copies of an object

- Linearizability
- Sequential Consistency

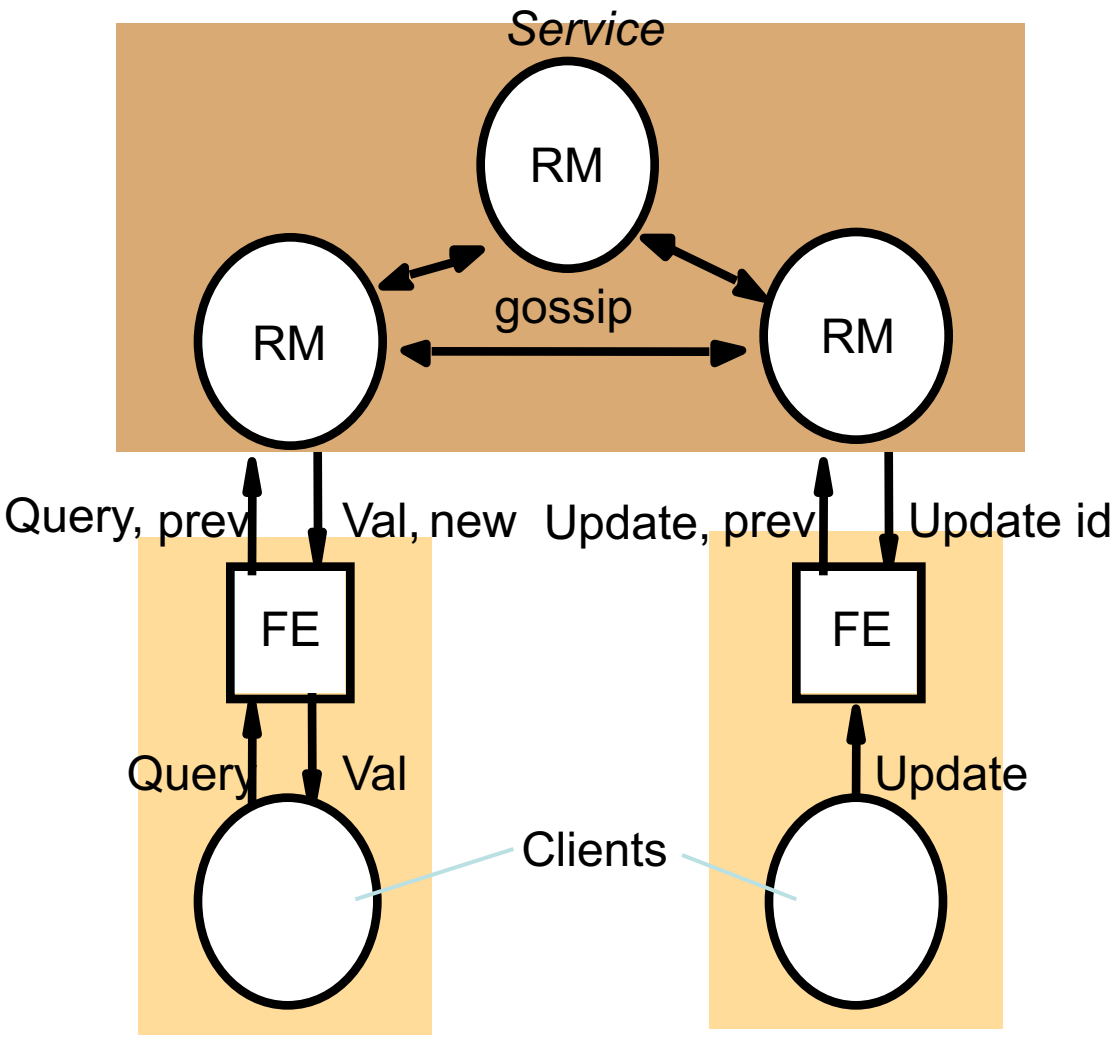
Gossiping Architecture

- The RMs exchange “gossip” messages
  (1) periodically and (2) among each other.

Gossip messages convey updates they have each received from clients, and serve to achieve anti-entropy (convergence of all RMs).

- Objective: provisioning of highly available service. Guarantee:
  - Each client obtains a consistent service over time: in response to a query, an RM may have to wait until it receives “required” updates from other RMs. The RM then provides client with data that at least reflects the updates that the client has observed so far.
  - Relaxed consistency among replicas: RMs may be inconsistent at any given point of time. Yet all RMs eventually receive all updates and they apply updates with ordering guarantees. Can be used to provide sequential consistency.

Query and Update Operations in a Gossip Service

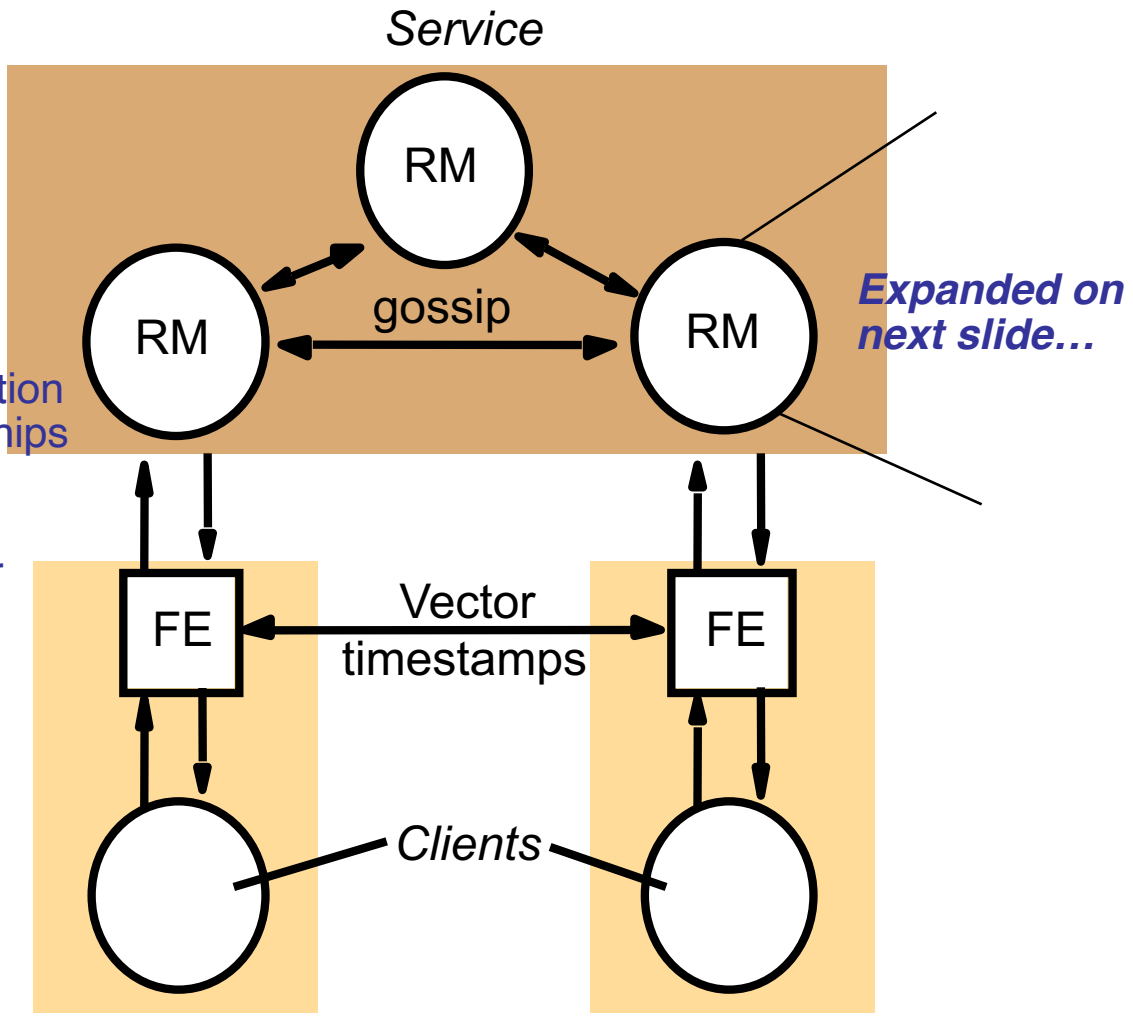


Various Timestamps

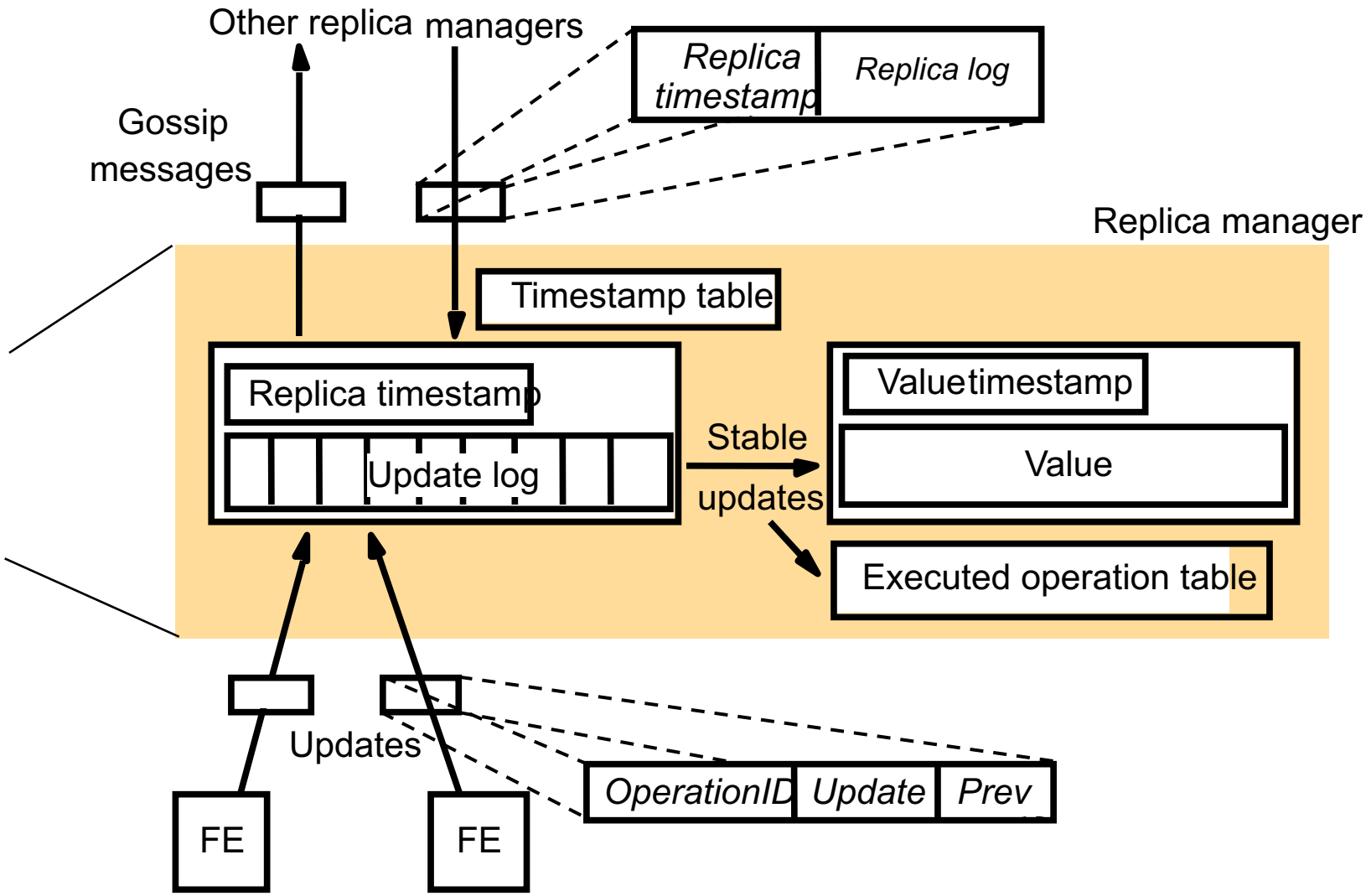
- Vector timestamp: the timestamp contains an entry for each RM
- Prev: Each front end keeps a vector timestamp
  - reflects the latest data values accessed by FE
- Replies to FE:
  - Query operation: a new timestamp, *new*.
  - Update operation: a timestamp, *update id*.
- Each returned timestamp is *merged* with the FE's previous timestamp to record the data that has been observed by the client.

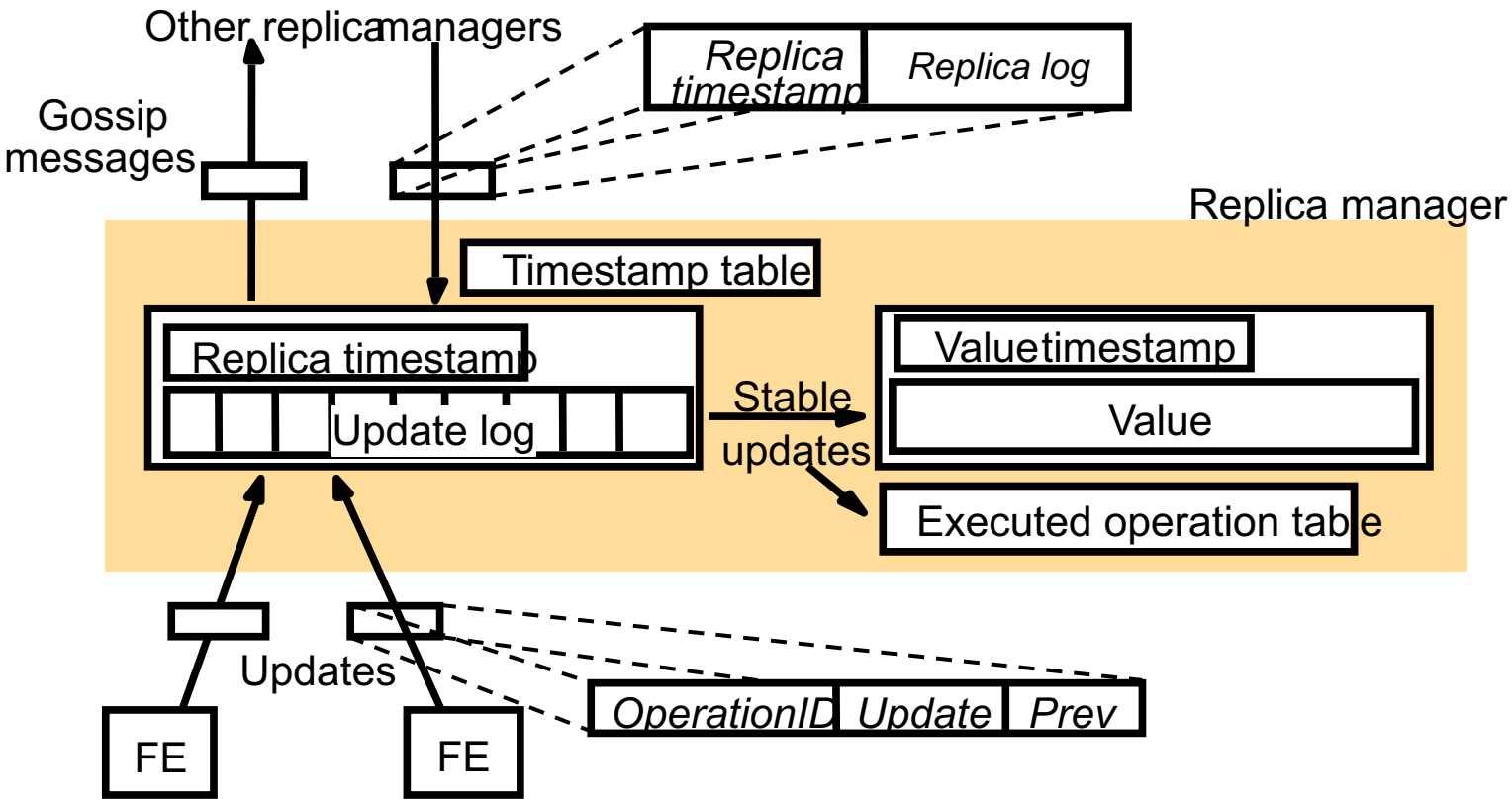
Front ends Propagate Their Timestamps

Since client-to-client communication can also lead to causal relationships between operations applied to services, the FE piggybacks its timestamp on messages to other clients.

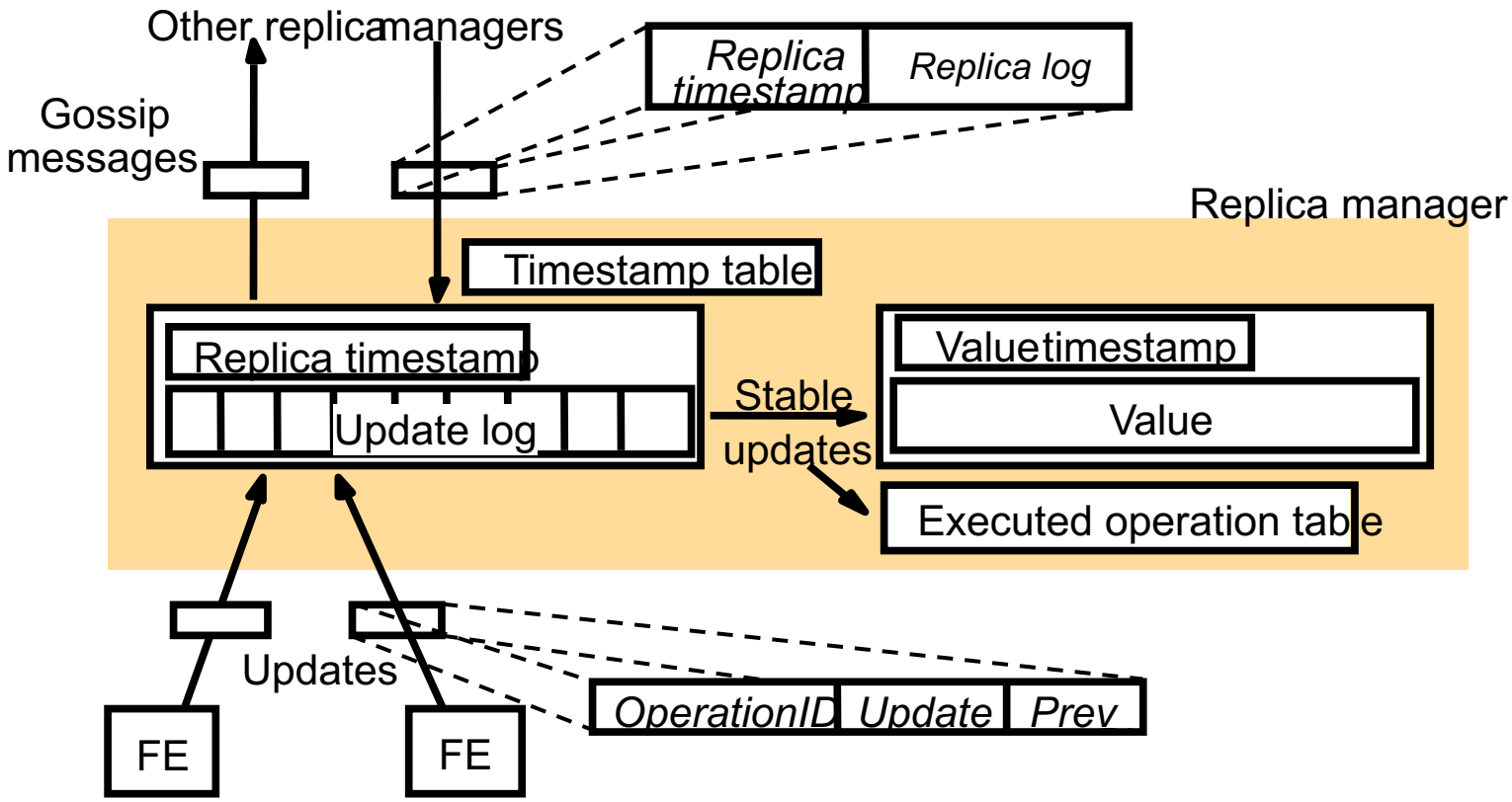


A Gossip Replica Manager

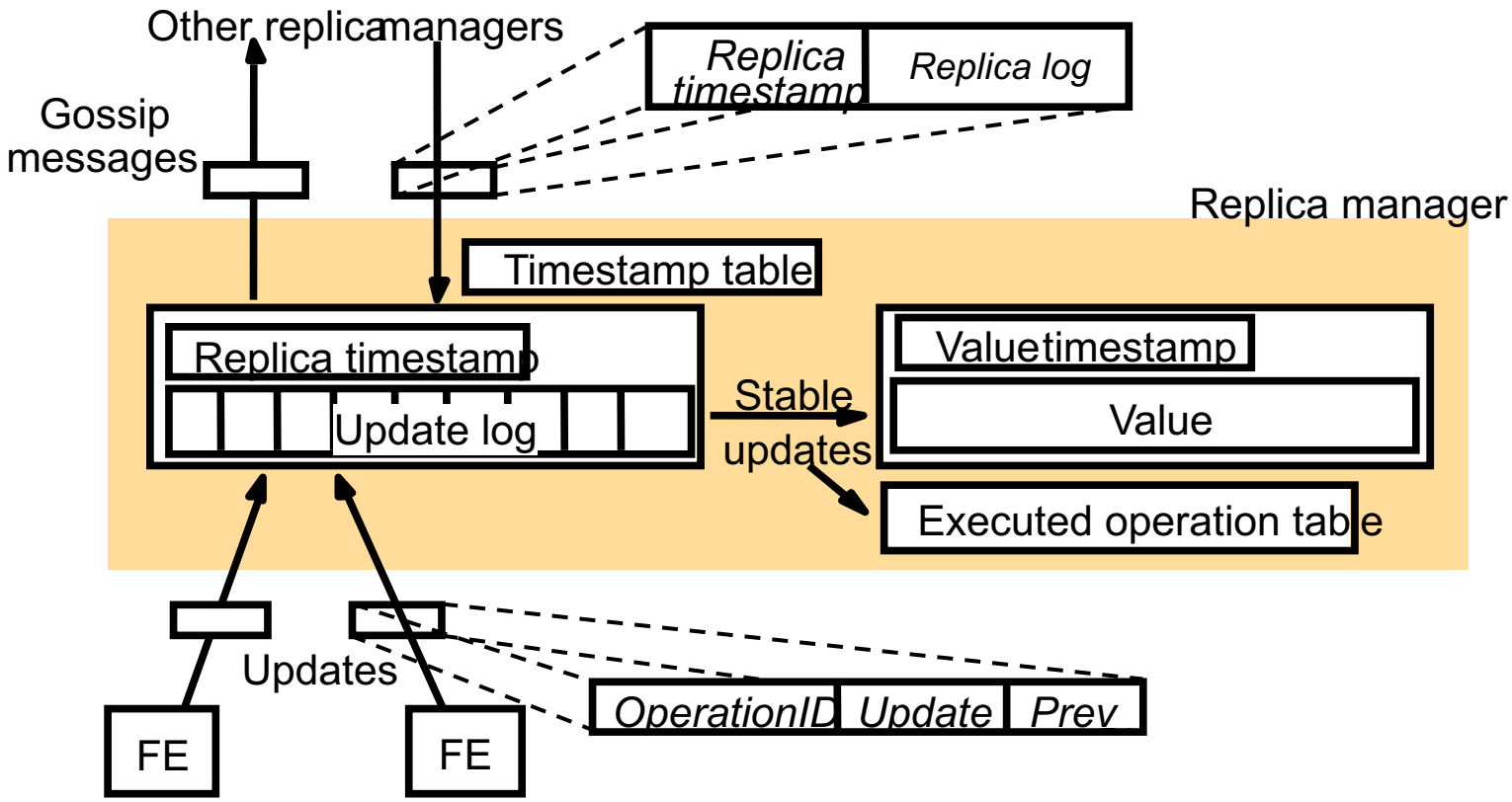




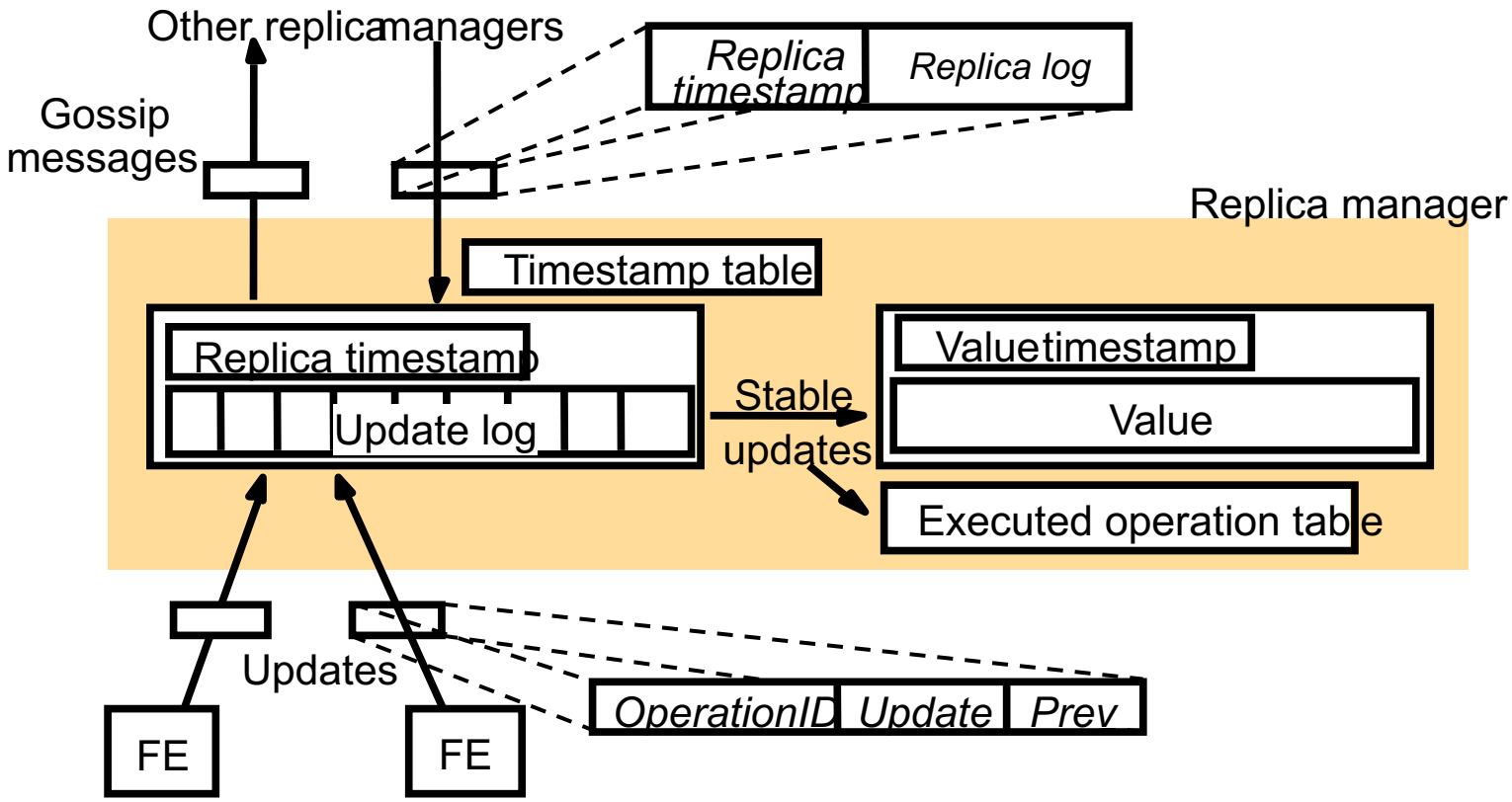
- Value: value of the object maintained by the RM.
- Value timestamp: the timestamp that represents the updates reflected in the value. Updated whenever an update operation is applied.



- Update log: records all update operations as soon as they are received, until they are reflected in Value.
  - Keeps all the updates that are not stable, where a stable update is one that has been received by all other RMs and can be applied consistently with its ordering guarantees.
  - Keeps stable updates that have been applied, but cannot be purged yet, because no confirmation has been received from all other RMs.
- Replica timestamp: represents updates that have been accepted by the RM into the log.



- Executed operation table: contains the FE-supplied ids of updates (stable ones) that have been applied to the value.
  - Used to prevent an update being applied twice, as an update may arrive from a FE and in gossip messages from other RMs.
- Timestamp table: contains, for each other RM, the latest timestamp that has arrived in a gossip message from that other RM.



- The *ith* element of a vector timestamp held by RM_i corresponds to the number of updates received from FEs by RM_i
- The *jth* element of a vector timestamp held by RM_i (*j* not equal to *i*) equals the number of updates received by RM_j and forwarded to RM_i *in gossip messages.*

Query Operations

- A query request *q* contains the operation, *q.op*, and the timestamp, *q.prev*, sent by the FE.
- Let *valueTS* denote the RM's value timestamp, then *q* can be applied if

  q.prev <= *valueTS*
- The RM keeps *q* on a hold back queue until the condition is fulfilled.
  - *If valueTS is (2,5,5) and q.prev is (2,4,6), then one update from RM_3 is missing.*
- Once the query is applied, the RM returns

  new ← *valueTS*

  to the FE (along with the value), and the FE merges *new* with its timestamp.

Update Operations

- Each update request *u* contains
  - The update operation, *u.op*
  - The FE's timestamp, *u.prev*
  - A unique id that the FE generates, *u.id*.

- Upon receipt of an update request, the RM *i*
  - Checks if u has been processed by looking up u.id in the executed operation table and the update log.
  - If not, increments the *i*-th element in the replica timestamp by 1 to keep track of the number of updates directly received from FEs.
  - Places a record for the update in the RM's log.

    logRecord := <i, ts, u.op, u.prev, u.id>

    where ts is derived from u.prev by replacing u.prev's ith element by the ith element of its replica timestamp.
  - Returns *ts* back to the FE, which merges it with its timestamp.

Update Operation (Cont'd)

- The stability condition for an update *u* is

  u.prev <= valueTS

  i.e., All the updates on which this update depends have already been applied to the value.
- When the update operation *u* becomes stable, the RM does the following
  - *value := apply(value, u.op)*
  - *valueTS := merge(valueTS, ts) (update the value timestamp)*
  - *executed := executed U {u.id} (update the executed operation table)*

Exchange of Gossiping Messages

- A gossip message *m* consists of the log of the RM, *m.log*, and the replica timestamp, *m.ts*.
  - Replica timestamp contains info about non-stable updates

- An RM that receives a gossip message has three tasks:
  - (1) Merge the arriving log with its own.
    - Let *replicaTS* denote the recipient RM's replica timestamp. A record *r* in *m.log* is added to the recipient's log unless *r.ts* <= *replicaTS*.
    - *replicaTS* ← *merge(replicaTS, m.ts)*
  - (2) Apply any updates that have become stable but not been executed (stable updates in the arrived log may cause some pending updates become stable)
  - (3) Garbage collect: Eliminate records from the log and the executed operation table when it is known that the updates have been applied everywhere.

Selecting Gossip Partners

- **The frequency with which RMs send gossip messages depends on the application.**
- **Policy for choosing a partner to exchange gossip with**:
  - Random policies: choose a partner randomly (perhaps with weighted probabilities)
  - Deterministic policies: a RM can examine its timestamp table and choose the RM that is the furthest behind in the updates it has received.
  - Topological policies: arrange the RMs into an overlay graph. Choose graph edges based on small round-trip times (RTTs), e.g., ring or Chord.
  - Each has its own merits and drawbacks. The ring topology produces relatively little communication but is subject to high transmission latencies since gossip has to traverse several RMs.

Summary

- Replication management
  - Request communication, Coordination, Execution, Agreement, Response
- Replication approaches
  - Passive v.s. active replications
  - Lazy v.s. eager update
- Replica group management
  - View maintenance
- Gossip architecture