DAD
Desenvolvimento de Aplicações Distribuídas

Lazy Replication
for high availability

Summary

- **Replication as a means to achieve high availability**
  - propagation and scheduling of operations
  - consistency
- **Example of sytems:**
  - Gossip
  - Bayou

- Gossip

The Gossip Architecture

- **the Gossip architecture is a framework for implementing highly available services**
  - data is **replicated** close to the location of clients
  - replica managers (RMs) periodically exchange '**gossip**' messages containing **updates**
- **Gossip service provides two types of operations**
  - **queries** - read only operations
  - **updates** - modify (but do not read) the state
- **FE (front-end) sends queries and updates to any chosen RM**
  - one that is **available** and gives reasonable response times

The Gossip Architecture

- **The system gives two guarantees (even if RMs are temporarily unable to communicate)**
  - each **client** gets a **consistent service over time** (i.e. **data** obtained **reflects** the **updates seen** by the client, even if the client uses different RMs)
    - **Vector timestamps** are used – with one entry per RM (**matrix clock in RMs**).
  - **relaxed consistency** between replicas.
    - All RMs **eventually** receive all updates.
    - RMs use **ordering guarantees** to suit the needs of the application (generally **causal ordering**).
    - Client may observe **stale data**.
- **Note that:**
  - while the Gossip architecture can be used to achieve sequential consistency (with additional total ordering), it is primarily intended to deliver **weaker consistency guarantees**.
  - Two clients may **observe different replicas**; and a client may observe **stale data**.

Gossip Processing of Queries and Updates

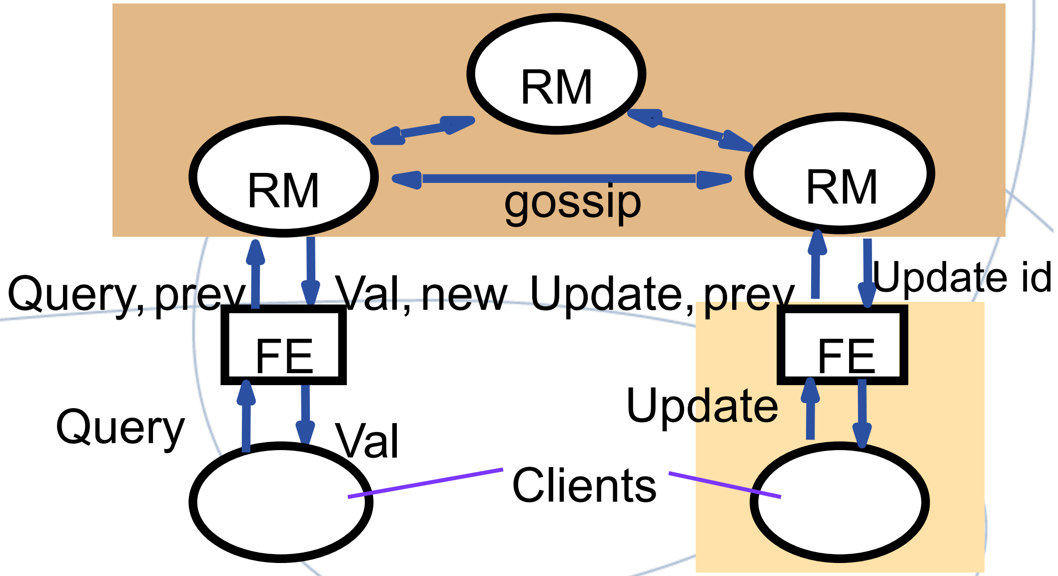
- **The phases in performing a client request are:**
  - **request**
    - FEs normally use the same RM and **may be blocked** on **queries**
    - **update** operations **return** to the client as **soon** as the operation is passed to the FE
  - **update response**
    - the RM **replies as soon** as it has **received** the update
  - **coordination**
    - the RM **waits** to **apply** the request until the **ordering constraints** apply.
    - this may involve **receiving updates** from other RMs in **gossip messages.**
  - **execution**
    - the RM **executes** the request
  - **query response**
    - if the request is a **query** the RM now **replies**
  - **agreement**
    - RMs **update** one another by ***exchanging*** **gossip** messages (**lazily**)
      - e.g. when **several updates** have been **collected**
      - or when an RM **discovers** it is **missing** an **update** (that it needs to process a request) that was sent to one of its peers.

Queries, Updates and Propagation in Gossip

- **Each FE keeps a vector timestamp that reflects the version of the latest data values accessed by the FE (and therefore accessed by the client):**
  - It is denoted **prev**
  - It contains an entry for every RM
  - The FE sends it in every request message to a RM, together with a description of the query or update operation itself

- **When a RM returns a value as a result of a query operation:**
  - It supplies a new vector timestamp called **new**
  - Because the replicas may have been updated since the last operation
- **When an update operation occurs, the RM returns:**
  - A vector timestamp called **update**
  - This vector is **unique** to the update

- **Each returned timestamp is:**
  - **Merged** (vector clock rules) with the FE's **previous** timestamp to **record** the **version** of the replicated data that has been **observed by the client**

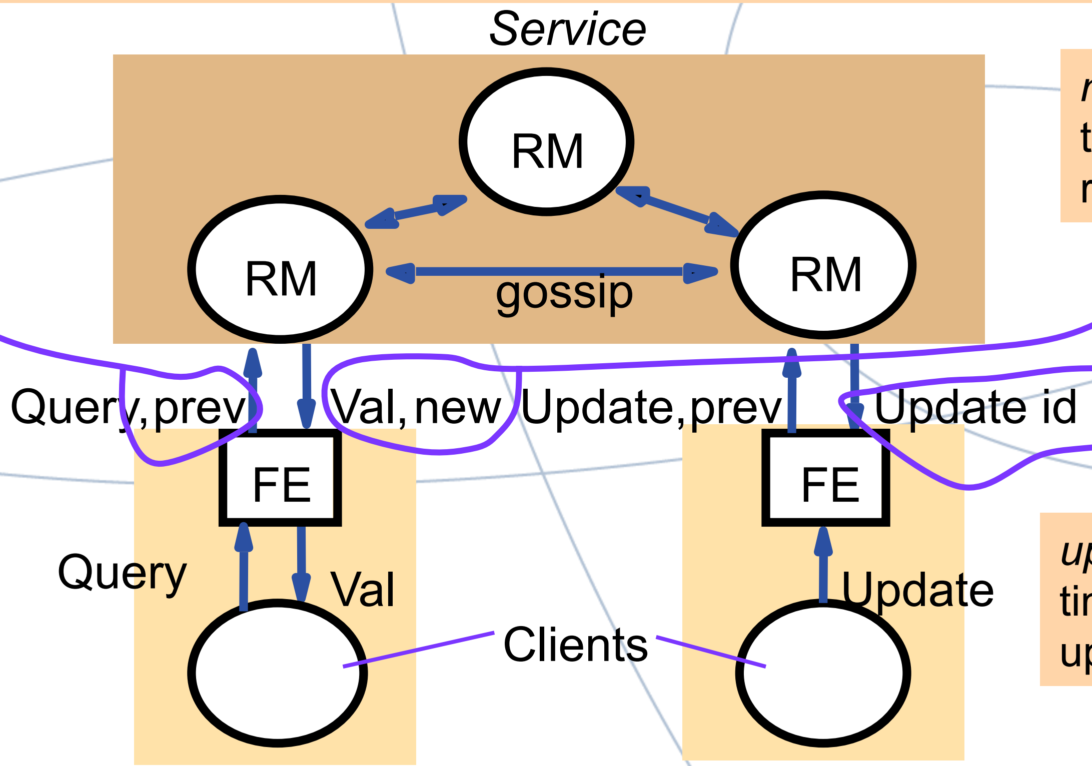


- **Clients exchange data by accessing the same gossip service and by communicating directly with one another:**
  - FEs **piggyback** their vector timestamps on messages to other clients
  - The recipients **merge** them with their own timestamps so that **causal relationships** can be **inferred correctly.**

Queries and Updates in Gossip

- **The service consists of a collection of RMs that exchange gossip messages**
- **Queries and updates are sent by a client via an FE to an RM**

prev is a vector timestamp for the latest version seen by the FE (and client)



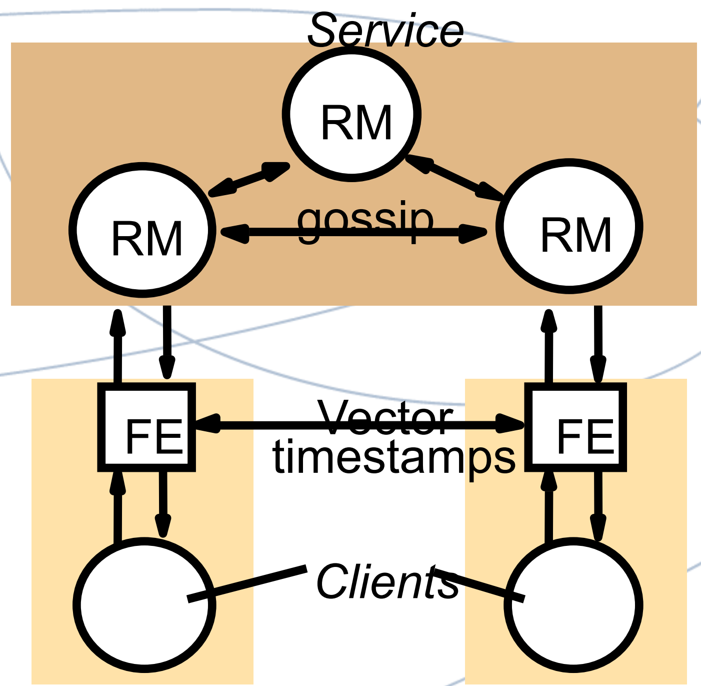
new is the vector timestamp of the resulting value, *val*

update id is the vector timestamp of the update

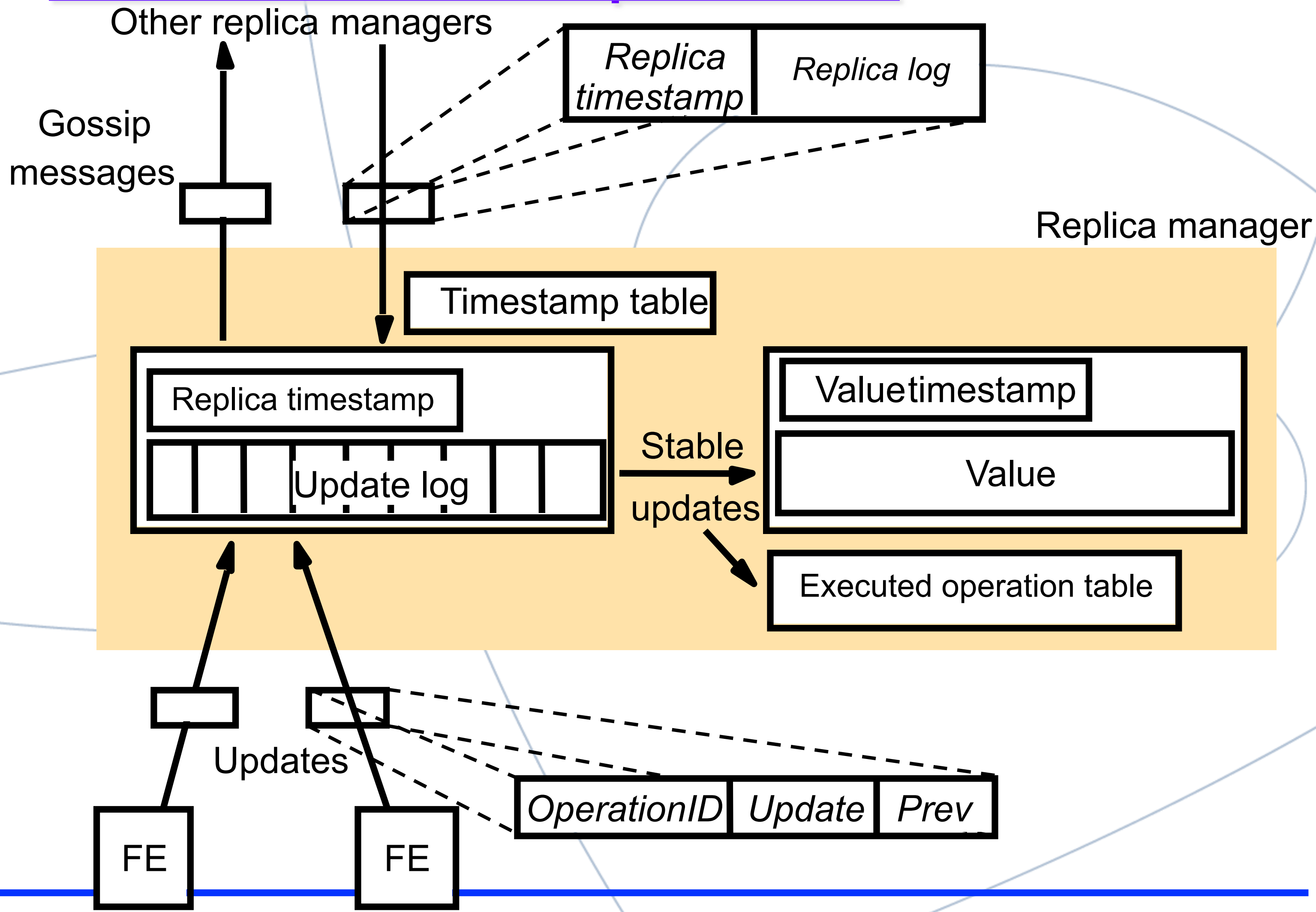
Propagation of Timestamps in Gossip

- **Front ends propagate their timestamps whenever clients communicate directly**
- **each FE keeps a vector timestamp of the latest value seen (prev)**
  - which it sends in every request
  - clients communicate with one another via FEs which pass vector timestamps

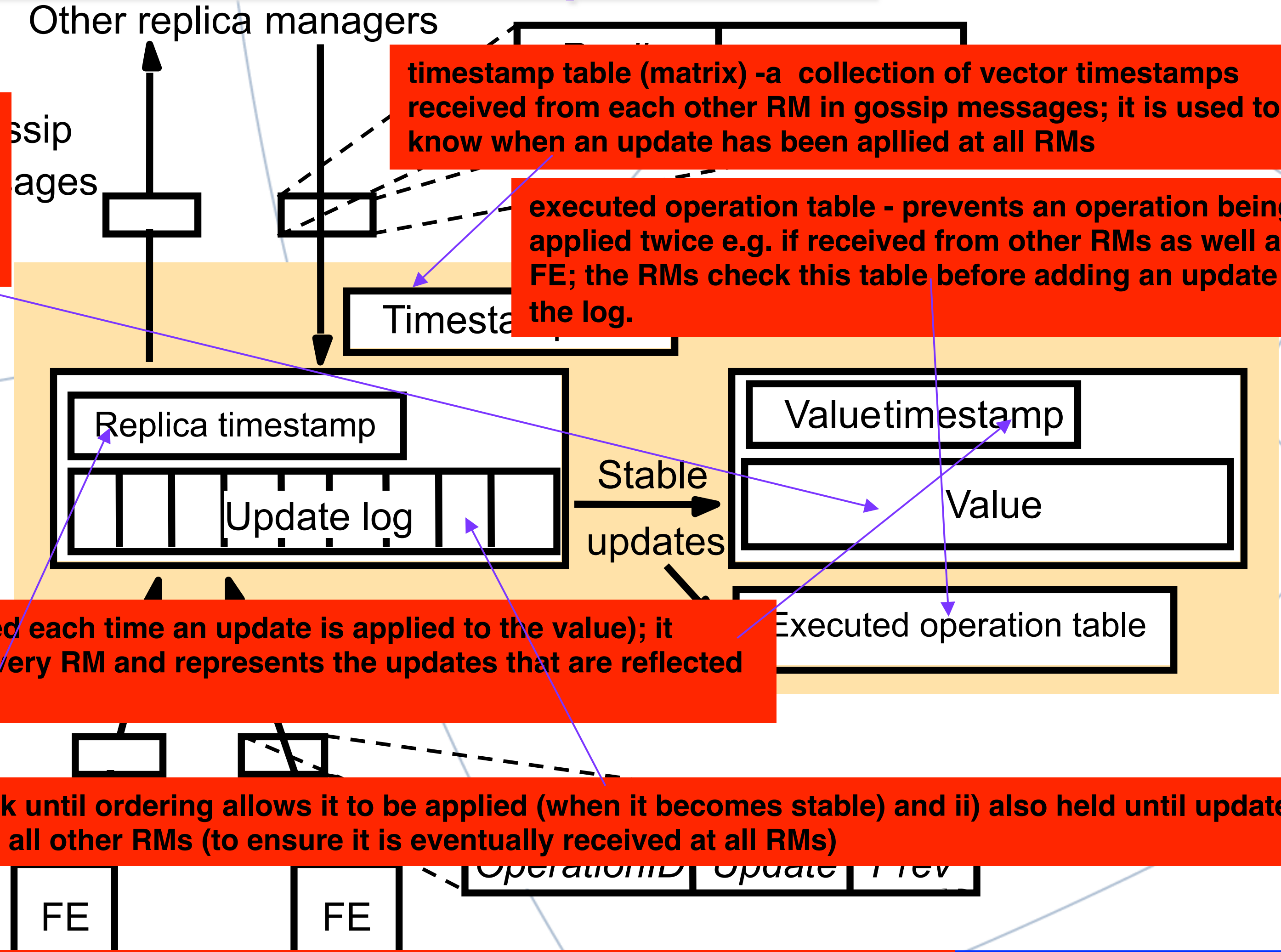
client-to-client communication can lead to causal relationships between operations.



Gossip Replica Manager: main state components



Gossip Replica Manager: main state components



Value: application state (each RM is a state machine); we are only talking about one value here

timestamp table (matrix) -a collection of vector timestamps received from each other RM in gossip messages; it is used to know when an update has been apllied at all RMs

executed operation table - prevents an operation being applied twice e.g. if received from other RMs as well as FE; the RMs check this table before adding an update to the log.

value timestamp (updated each time an update is applied to the value); it contains one entry for every RM and represents the updates that are reflected in the value

update log: i) held-back until ordering allows it to be applied (when it becomes stable) and ii) also held until updates have been received by all other RMs (to ensure it is eventually received at all RMs)

replica timestamp - indicates updates accepted by RM in log (different from value's timestamp if some updates are not yet stable)

Processing of Query Operations

- **Vector timestamp held by RM_i consists of:**
  - ith element holds updates received from FEs by that RM
  - jth element holds updates received by RM_j and propagated to RM_i in gossip messages
- **RMs are numbered 0, 1, 2,…**
  - e.g. in a gossip system with 3 RMs a value of **(2,4,5)** at RM_0 means that the value there reflects the first 2 updates accepted from FEs at RM_0, the first 4 at RM_1 and the first 5 at RM_2.

Processing of Query Operations (1/3)

- **Query operations contain q.prev:**
  - **q.prev** reflects the latest version of the value that the FE has read or submitted as an update
    - thus, the task of the RM is to return a value that is at least as recent as this

Processing of Query Operations (2/3)

- If valueTS is the replica's value timestamp, then q can be applied to the replica's value (recall it's a query operation) if q.prev ≤ valueTS
- failing this, the RM keeps q on the list of pending query operations until the above condition is fulfilled; the RM can wait for the missing updates that will arrive in gossip messages or initiate them
  - e.g. if valueTS = (2,5,5) and q.prev = (2,4,6) - an update from RM_2 is missing in valueTS
  - this means that the FE that submitted q must have contacted a different RM previously for it to have seen this update, which the RM has not seen

Processing of Query Operations (3/3)

- Once the query can be applied, the RM returns valueTS as new the timestamp for the FE:
  - the FE merges new with its vector timestamp – frontEndTS = merge(frontEndTS, new)
  - The update at RM_1 that the FE has not seen before the query (q.prev has 4 where the RM has 5) will be reflected in the update to frontEndTS (and potentially in the value returned, depending on the query)

Processing of Update Operations (1/3)

- **A FE sends update operation u.op, u.prev, u.id to RM i**
  - A FE can send a **request** to **several RMs**, using **same id**
- **When RM i receives an update request, it checks whether it is new, by looking for the id in its executed operation table and its log**
  - The RM **discards** the update if it **has already seen** it
- **if it is new, the RM i**
  - **increments** by 1 the **ith element** of its **replica timestamp**,
  - assigns a **unique** vector **timestamp** ts to the update
  - and **stores** the update in its log
  - **logRecord = <i, ts, u.op, u.prev, u.id>**

- **The timestamp <u>ts</u> is calculated from <u>u.prev</u> by <u>replacing</u> <u>its ith element</u> by the ith element of the <u>replica timestamp</u> (which it has just incremented):**
  - This action makes ts <u>unique</u>, thus ensuring that all system components will correctly record whether or not they have observed the update

Processing of Update Operations (2/3)

- **The RM returns ts to the FE, which merges it with its vector timestamp**
- **The stability condition for an update u is similar to that of queries:**
  - **u.prev ≤ valueTS**
  - This condition states that all the updates on which this update depends – that is, all the updates that have been observed by the front end that issued the update – have already been applied to the value *(causality enforcement)*
  - If this condition is not met at the time the update is submitted, it will be checked again when gossip messages arrive

Update operations are processed in causal order

Processing of Update Operations (3/3)

- **When the stability condition has been met for an update record r, the RM applies the update to the value and updates the value timestamp and the executed operations table:**
  - In other words, the **RM applies** the operation **u.op** to the value, **updates valueTS** and **adds u.id** to the executed operation table.
  - value = apply (value, r.u.op)
  - valueTS = merge (valueTS, r.ts)
  - executed = executed ∪ {r.u.id}

- **Application of the update to the value**
- **The update's timestamp is merged with that of the value**
- **The update's operation identifier is added to the set of identifiers of operations that have been executed (which is used to check for repeated operation requests)**

Gossip Messages

- **an RM uses entries in its timestamp table to estimate which updates another RM has not yet received**
  - The timestamp **table** contains a **vector timestamp for each other replica, collected from gossip messages**
- **gossip message m contains:**
  - Its log **m.log**, and
  - Its replica timestamp **m.ts**
- **an RM receiving gossip message m has the following main tasks**
  - Let replicaTS denote the recipient's replica timestamp
  - **add the arriving log to its own** (omit those entries with m.ts ≤ replicaTS, in which case it is already in the log or it has been applied and the entry discarded)
  - The RM **merges the timestamp** of the **incoming gossip message with its own replicaTS**, so that it corresponds to the additions to the log:
    - **replicaTS = merge (replicaTS, m.ts)**

Gossip Messages

- **When new updates have been merged into the log, the RM collects the set S of any updates in the log that are now stable**
  - These can be **applied** to the value but care must be taken over the order in which they are apllied so that the **happened-before relation is observerd**
  - The RM **sorts updates** in the set according to the **partial order <= between vector timestamps**
  - It then **applies the updates in this order, smallest first**, i.e. each $r \in S$ is applied only when there is no $s \in S$ such that **s.prev < r.prev**

Gossip Messages

- **The RM then looks for records in the log that can be discarded:**
  - **remove redundant entries from the log** when it is known that they have been applied by all RMs
  - If the **gossip message was sent by RM_j** and if tableTS is the table of replica timestamps of the RMs, then the RM sets **tableTS[j] = m.ts**
  - The RM can now **discard any record** r in the log for an **update that has been received <u>everywhere</u>**
  - That is, if c is the RM that created the record, then we require **<u>for all</u> RMs i**:
    - **tableTS[i][c] >= r.ts[c]**

Discussion of Gossip Architecture

- **the Gossip architecture is designed for highly available services**
  - it uses a **lazy form of replication** in which **RMs update one another** from time to time by means of **gossip messages**
  - it **allows clients** to make **updates** to local replicas **while partitioned**
  - **RMs exchange updates** with one another **when reconnected**
- **clients with access to a single RM can work when other RMs are inaccessible**
  - but **it is not suitable** for data such as **bank accounts** (this requires sequential consistency)
  - it is **inappropriate** for updating replicas in **real time** (e.g. a conference)

Discussion of Gossip Architecture

- **scalability**
  - as the number of **RMs grow**, so does the **number of gossip messages**
  - for R RMs, the number of messages per request (2 for the request and the rest for gossip) = **2 + (R-1)/G**
    - G is the number of updates per gossip message
    - **increase G** and **improve number of gossip messages**, but **make latency worse** for applications where **queries are more frequent than updates**,
    - **use some read-only replicas**, which are **updated only by gossip** messages
      - read-only replicas do not need to propagate gossips
      - they do not need to be tracked in the VCs

Additional types of update operations

- **In addition to Causal ordering, Gossip supports two additional types of semantics for update operation:**
  - Forced update : performed in the same order (relative to other forced updates) at all replicas.
  - Immediate update : performed at all replicas in the same order relative to **all** other operations.

Forced Update

- Use the primary to assign a global unique identifier.
- The primary carries out a two phase protocol for updates.

Two phase protocol

- **Upon receiving an update, the primary sends it to all other replicas.**
- **Upon receiving responses from the majority of the backups,**
  - this way a majority of RM record which update is next in sequence before the operation can be applied
  - the primary commit the update by insert the record to its log.
- **Backups know the commitment from gossip messages.**

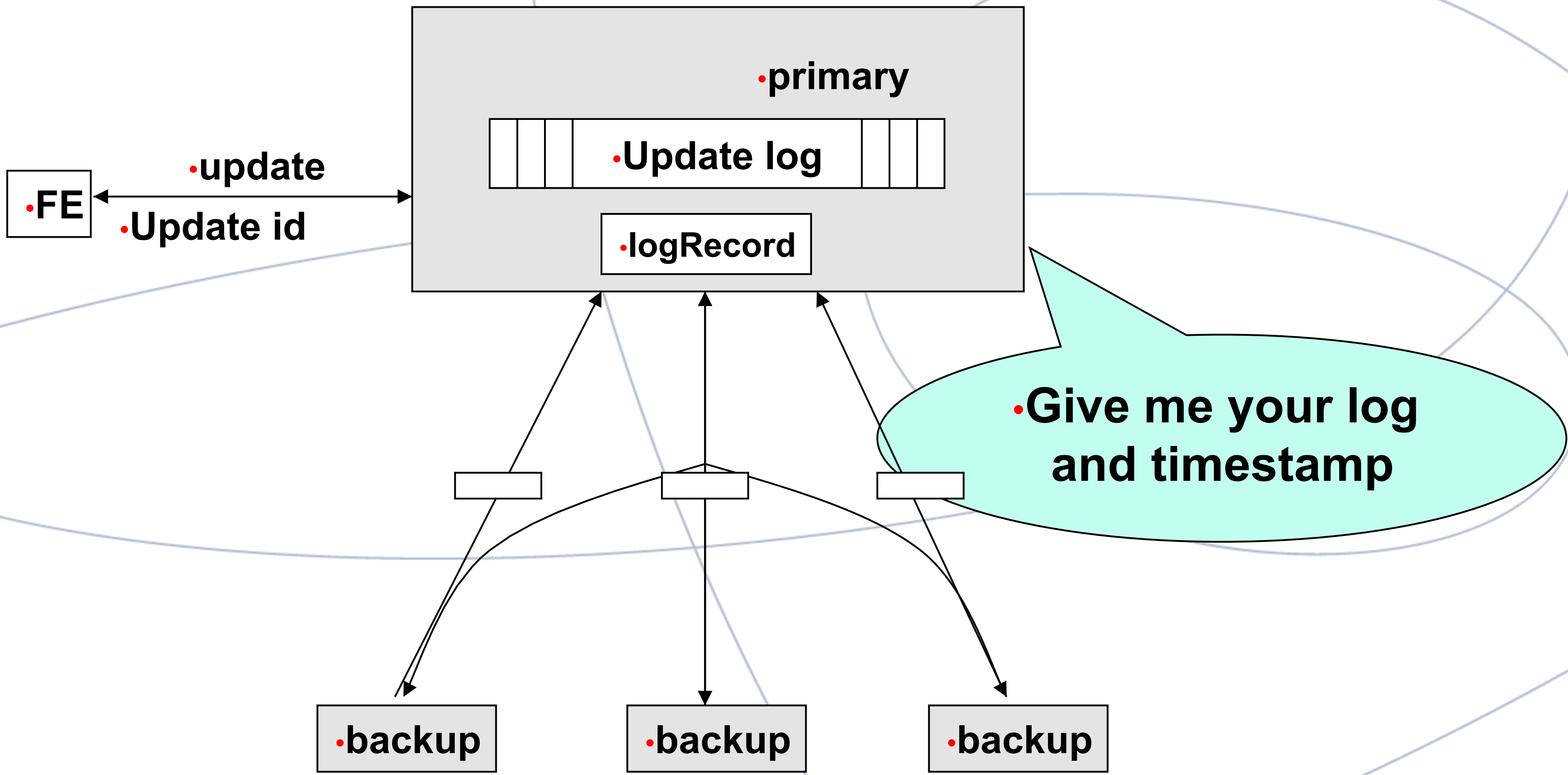
Fail Recovery

- **New coordinator informs participants about the failure.**
- **Participants inform coordinator about most recent forced updates**
- **Coordinator assigns UID based on the largest it knows after the majority of replicas has responded.**
  - Assume perfect failure detection…
  - can be revised to use consensus algorithms that work even with unreliable failure detectors (e.g., Paxos)

Immediate Update

- **Primary use 3 phase protocol.**
  - Pre-prepare :
    - additional phase required to discover which causally ordered updates have to precede the current immediate update
  - Prepare:
    - as for forced updates
  - Commit
    - as for forced updates

3 phase protocol



Summary

- **Replication as a means to achieve high availability**
  - propagation and scheduling of operations
  - consistency
- **Example of sytems:**
  - Gossip
  - Bayou