Replication

Marco Aiello

RuG: Distributed Systems

Introduction to replication

Replication of data :- the maintenance of copies of data at multiple computers

- replication can provide the following
- performance enhancement
  - e.g. several web servers can have the same DNS name and the servers are selected in turn. To share the load.
  - replication of read-only data is simple, but replication of changing data has overheads
- fault-tolerant service
  - guarantees correct behaviour in spite of certain faults (can include timeliness)
  - if *f* of *f*+1 servers crash then 1 remains to supply the service
- availability is hindered by
  - server failures
    - replicate data at failure- independent servers and when one fails, client may use another. Note that caches do not help with availability(they are incomplete).
  - network partitions and disconnected operation
    - Users of mobile computers deliberately disconnect, and then on re-connection, resolve conflicts

Requirements for replicated data

- Replication transparency
  - clients see logical objects (not several physical copies)
    - they access one logical item and receive a single result
- Consistency
  - specified to suit the application,
    - e.g. when a user of a diary disconnects, their local copy may be inconsistent with the others and will need to be reconciled when they connect again. But connected clients using different copies should get consistent results.

System model

- each *logical* object is implemented by a collection of *physical* copies called *replicas*
  - the replicas are not necessarily consistent all the time (some may have received updates, not yet conveyed to the others)

State machine

- applies operations atomically
- its state is a deterministic function of its initial state and the operations applied to them
- all replicas start identical and carry out the same operations
- Its operations must not be affected by clock readings etc.

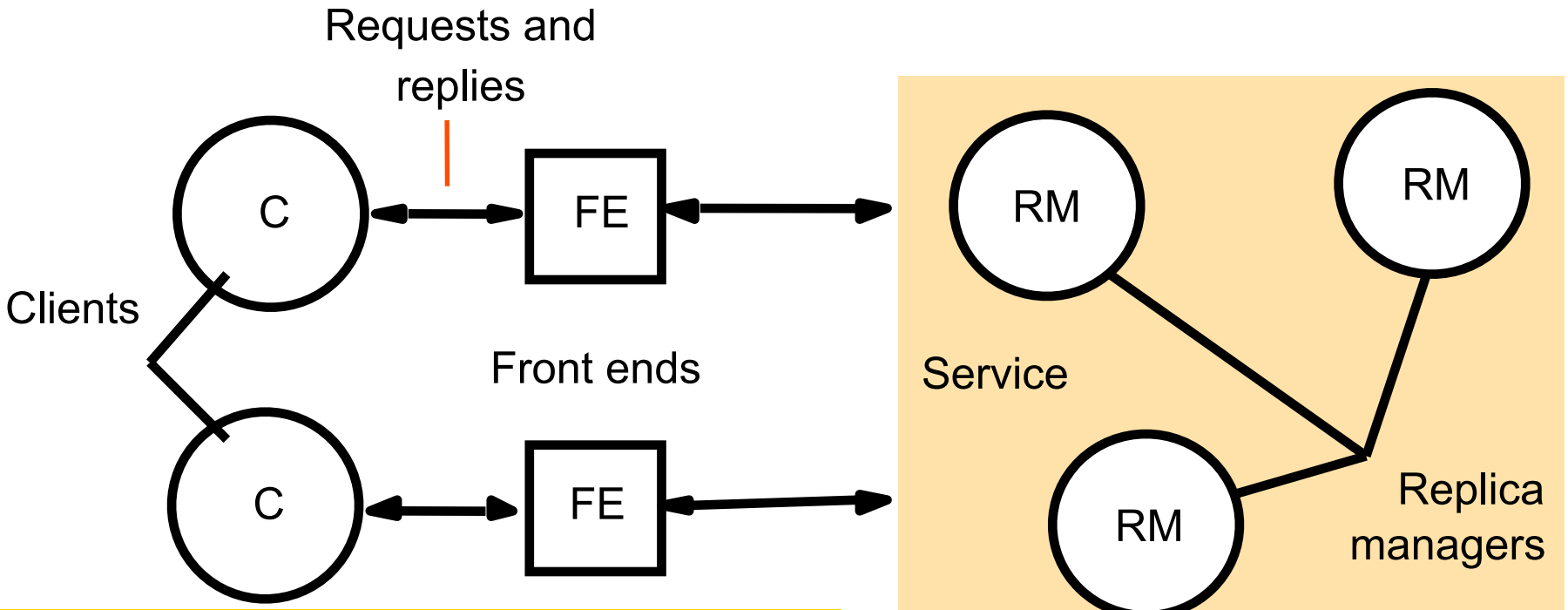
  - RMs apply operations to replicas recoverably
    - i.e. they do not leave inconsistent results if they crash
  - objects are copied at all RMs unless we state otherwise
  - static systems are based on a fixed set of RMs
  - in a dynamic system: RMs may join or leave (e.g. when they crash)
  - an RM can be a *state machine*, which has the following properties:

A basic architectural model for the management of replicated data

A collection of RMs provides a service to clients

Clients see a service that gives them access to logical objects, e.g., bank accounts, diaries, which are in fact replicated at the RMs

Clients request operations: those without updates are called *read-only* requests the others are called *update* requests (they may include reads)



Clients request are handled by front ends. A front end makes replication transparent.

Five phases in performing a request

- issue request
  - the FE either
    - sends the request to a single RM that passes it on to the others
    - or multicasts the request to all of the RMs (in state machine approach)

Total ordering: if a correct RM handles r before *r'*, then any correct RM handles *r* before *r'*

  - the RMs decide whether to apply the request, and decide on its ordering relative to other requests (according to FIFO, causal or total ordering)
- execution
  - the RMs execute the request (sometimes tentatively, i.e., they can undo it)
- agreement
  - RMs *agree* on the effect of the request, .e.g perform 'lazily' or immediately
- response
  - one or more RMs reply to FE. e.g.
    - for high availability give first response to client.
    - one or many replica managers may send response.

RMs agree - i.e. reach a consensus as to effect of the request. In Gossip, all RMs eventually receive updates.

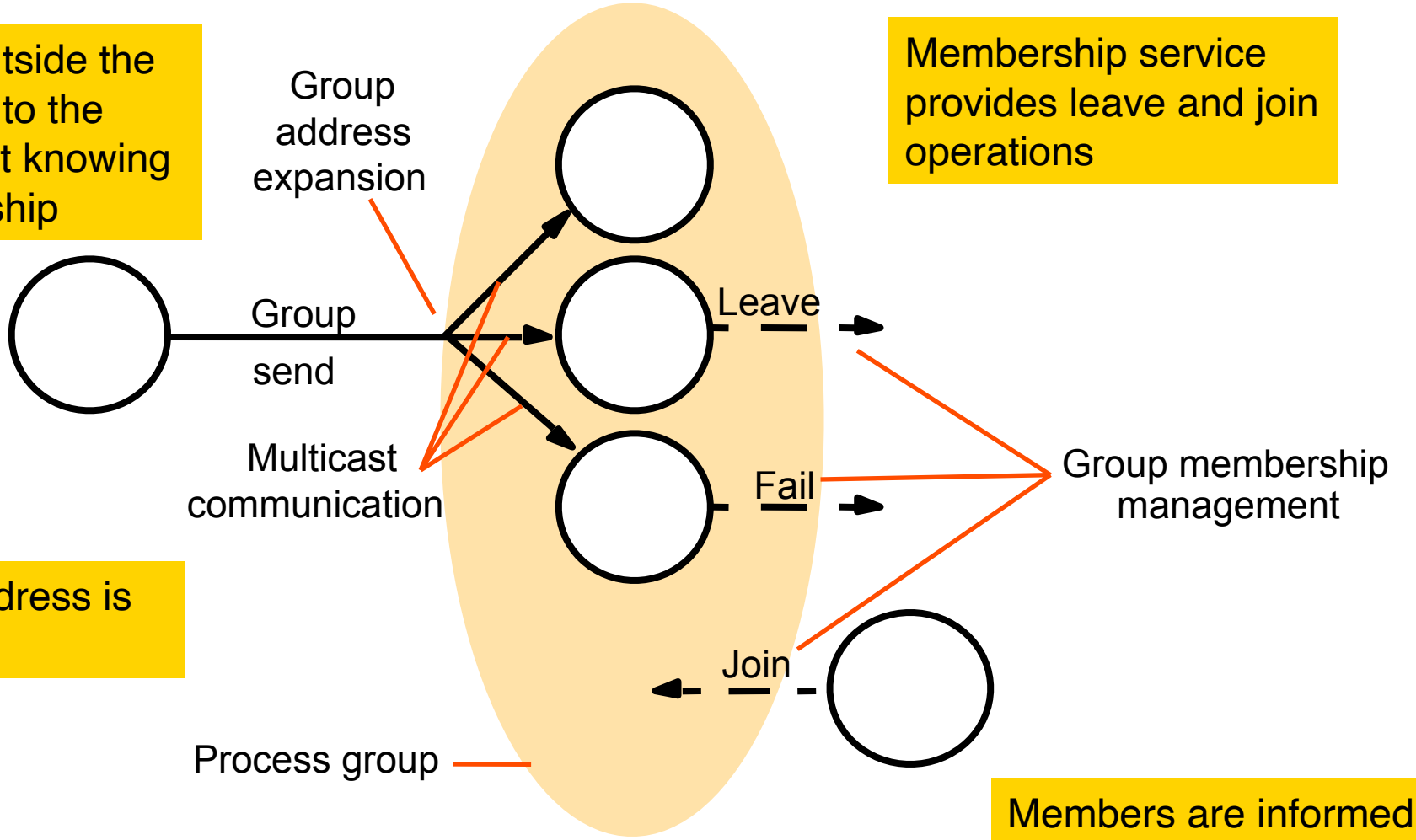
Group communication

We require a membership service to allow dynamic membership of groups

- process groups are useful for managing replicated data
  - but replication systems need to be able to add/remove RMs
- group membership service provides:
  - interface for adding/removing members
    - create, destroy process groups, add/remove members. A process can generally belong to several groups.
  - implements a failure detector
  - which monitors members for failures (crashes/communication),
    - and excludes them when unreachable
  - notifies members of changes in membership
  - expands group addresses
    - multicasts addressed to group identifiers, rather than list of processes
    - coordinates delivery when membership is changing
- e.g. IP multicast allows members to join/leave and performs address expansion, but does not inform about current membership in group & multicast delivery not coordinated with membership changes.

Services provided for process groups

A process outside the group sends to the group without knowing the membership



Membership service provides leave and join operations

The group address is expanded

Failure detector notes failures and evicts failed processes from the group

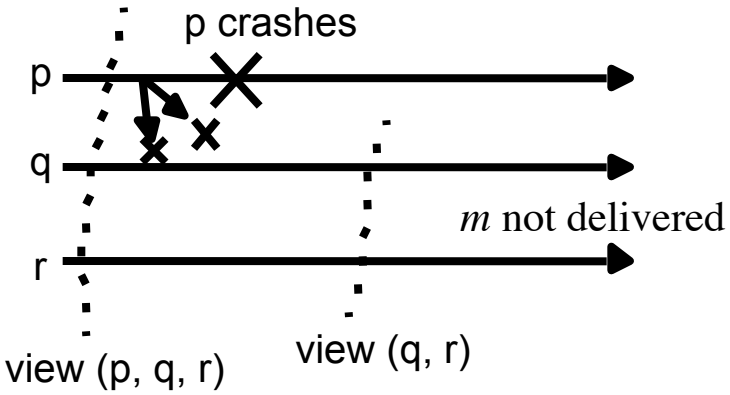
Members are informed when processes join/ leave

Membership services

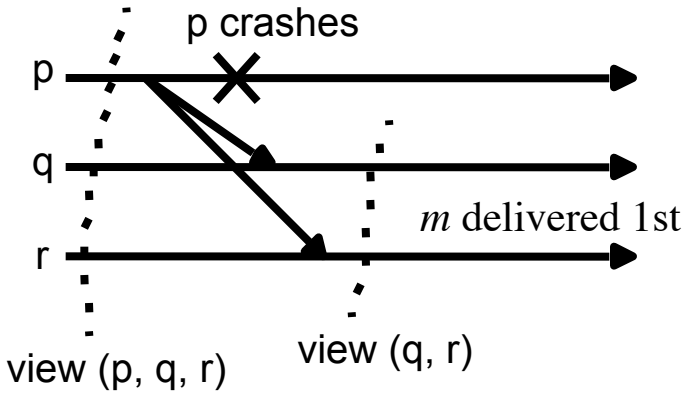
- A full membership service maintains *group view*s, which are lists of group members, ordered e.g. as members join group.
- A new group view is generated each time a process joins or leaves the group.
- *View delivery* The idea is that processes can 'deliver views' (like delivering multicast messages) when membership changes.
  - ideally we would like all processes to get the same information in the same order relative to the messages.
- *view synchronous group communication* with reliability.
  - all processes agree on the ordering of messages and membership changes,
  - a joining process can safely get state from another member.
  - or if one crashes, another will know which operations it had already performed
  - If a process delivers a message *m*, then it will not deliver it again.

View-synchronous group communication

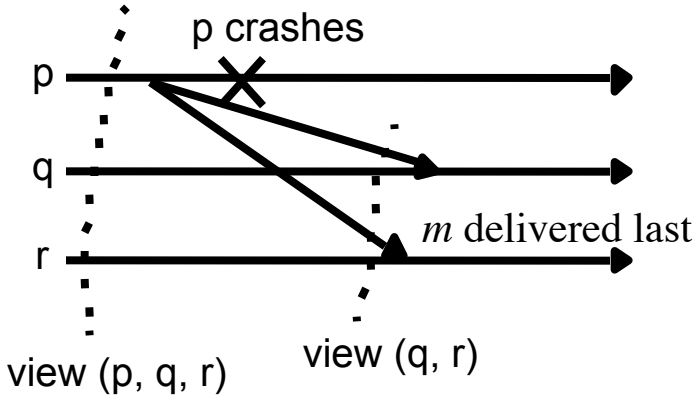
a (allowed).



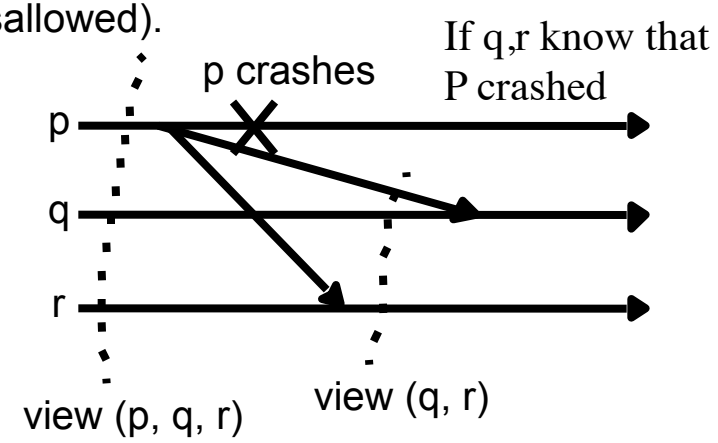
b (allowed).



c (disallowed).



d (disallowed).



Fault-tolerant services

- provision of a service that is correct even if *f* processes fail
  - by replicating data and functionality at RMs
  - assume communication reliable and no partitions
  - RMs are assumed to behave according to specification or to crash
  - intuitively, a service is correct if it responds despite failures and clients can't tell the difference between replicated data and a single copy
  - but care is needed to ensure that a set of replicas produce the same result as a single one would.

Example of a naive replication system

Client 1:	Client 2:
$setBalance_B(x,1)$	
$setBalance_A(y,2)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

RMs at *A* and *B* maintain copies of *x* and *y*

- clients use local RM when available, otherwise the other one
- RMs propagate updates to one another after replying to client

- initial balance of x and y is $0
  - client 1 updates X to x = 1 at B (local) then finds B has failed, so uses A y = 2
  - client 2 reads balances at A (local)
    - as client 1 updates y after x, client 2 should see $1 for x
  - not the behaviour that would occur if A and B were implemented at a single server
- Systems can be constructed to replicate objects without producing this anomalous behaviour.

–The real-time requirement means clients should receive up-to-date information

- but may not be practical due to difficulties of synchronizing clocks
- a weaker criterion is sequential consistency

- The correctness criteria for replicated objects are defined by referring to a virtual interleaving which would be correct

a replicated object service is *linearizable* if *for any execution* there is some interleaving of clients' operations such that:

–the interleaved sequence of operations meets the specification of a (single) correct copy of the objects

–the order of operations in the interleaving is consistent with the real time at which they occurred

– For any set of client operations there is a virtual interleaving (which would be correct for a virtual single image of the shared objects).
– Each client sees a view of the objects that is consistent with this single image, that is, the results of clients operations make sense within the interleaving
  - the bank example did not make sense: if the second update is observed,the first update should be observed too.

linearizability is not intended to be used with transactional replication systems

Sequential consistency

- a replicated shared object service is sequentially consistent if for any execution there is some interleaving of clients' operations such that:
  - the interleaved sequence of operations meets the specification of a (single) correct copy of the objects
  - the order of operations in the interleaving is consistent with the program order in which each client executed them

the following is sequentially consistent but not linearizable

Client 1:	Client 2:
$setBalance_B(x,1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y,2)$	

this is possible under a naive replication strategy, even if neither *A* or *B* fails -

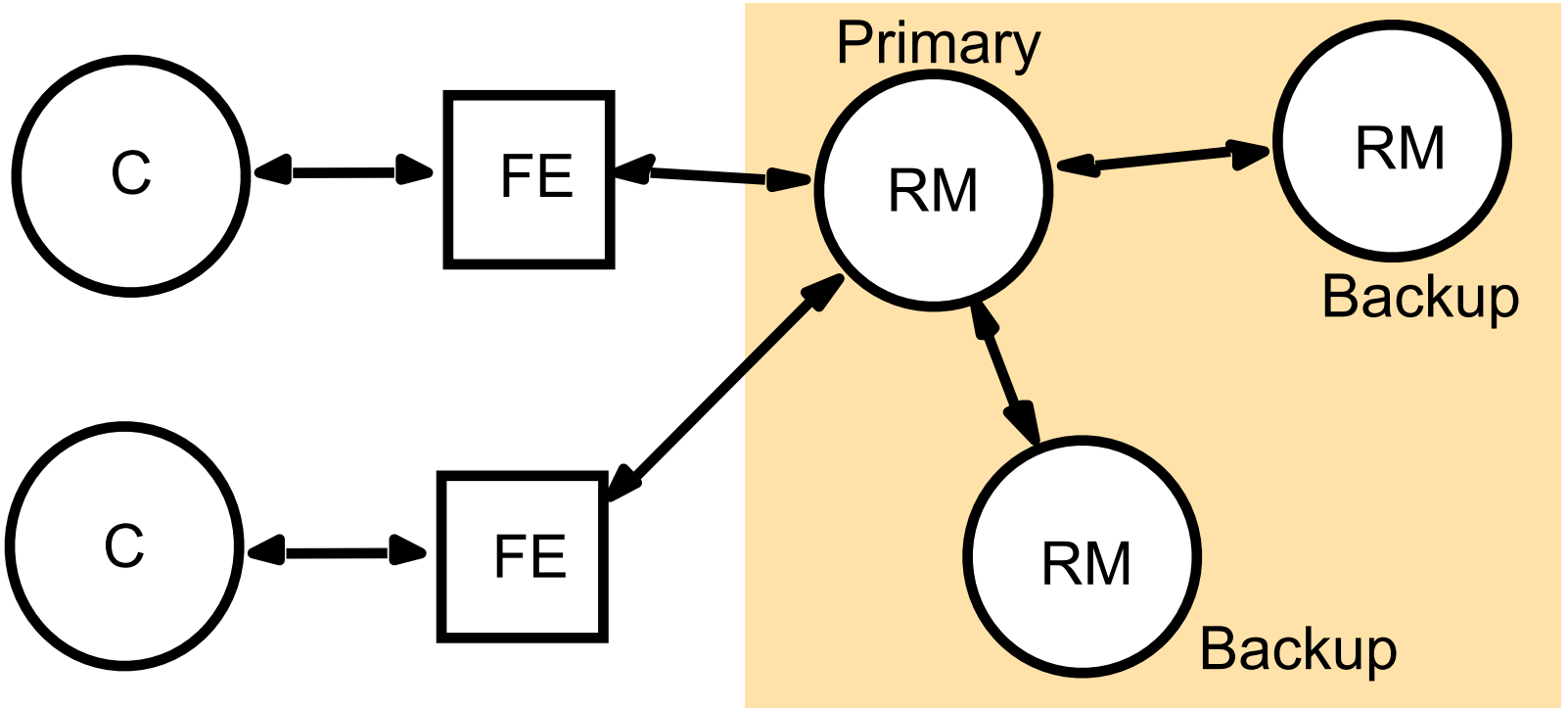
the update at *B* has not yet been propagated to *A* when client 2 reads it

but the following interleaving satisfies both criteria for sequential consistency :

$getBalance_A(y) \rightarrow 0$; $getBalance_A(x) \rightarrow 0$; $setBalance_B(x,1)$; $setBalance_A(y,2)$

The FE has to find the primary, e.g. after it crashes and another takes over

The passive (primary-backup) model for fault tolerance



- There is at any time a single primary RM and one or more secondary (backup, slave) RMs
- FEs communicate with the primary which executes the operation and sends copies of the updated data to the result to backups
- if the primary fails, one of the backups is promoted to act as the primary

Passive (primary-backup) replication. Five phases.

- The five phases in performing a client request are as follows:
- 1. Request:
  - a FE issues the request, containing a unique identifier, to the primary RM
- 2. Coordination:
  - the primary performs each request atomically, in the order in which it receives it relative to other requests
  - it checks the unique id; if it has already done the request it re-sends the response.
- 3. Execution:
  - The primary executes the request and stores the response.
- 4. Agreement:
  - If the request is an update 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 FE, which hands the response back to the client.

Passive (primary-backup) replication (discussion)

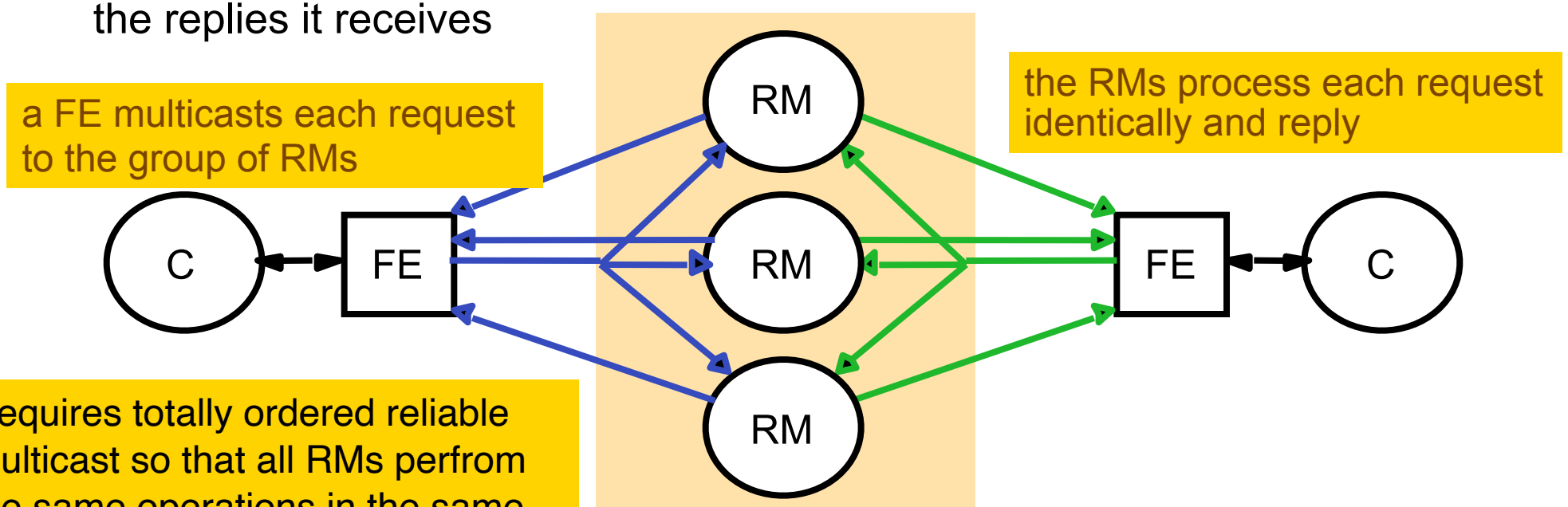
- This system implements linearizability, since the primary sequences all the operations on the shared objects
- If the primary fails, the system is linearizable, if a single backup takes over exactly where the primary left off, i.e.:
  - the primary is replaced by a unique backup
  - surviving RMs agree which operations had been performed at take over
- view-synchronous group communication can achieve this
  - when surviving backups receive a view without the primary, they use an agreed function to calculate which is the new primary.
  - The new primary registers with name service
  - view synchrony also allows the processes to agree which operations were performed before the primary failed.
  - E.g. when a FE does not get a response, it retransmits it to the new primary
  - The new primary continues from phase 2 (coordination) -uses the unique identifier to discover whether the request has already been performed.

Discussion of passive replication

- To survive *f* process crashes, *f*+1 RMs are required
- To design passive replication that is linearizable
  - View synchronous communication has relatively large overheads
  - Several rounds of messages per multicast
  - After failure of primary, there is latency due to delivery of group view
- variant in which clients can read from backups
  - which reduces the work for the primary
  - get sequential consistency but not linearizability
- Sun NIS uses passive replication with weaker guarantees
  - Weaker than sequential consistency, but adequate to the type of data stored
  - achieves high availability and good performance
  - Master receives updates and propagates them to slaves using 1-1 communication. Clients can uses either master or slave
  - updates are not done via RMs - they are made on the files at the master

Active replication for fault tolerance

- the RMs are *state machines* all playing the same role and organised as a group.
  - all start in the same state and perform the same operations in the same order so that their state remains identical
- If an RM crashes it has no effect on performance of the service because the others continue as normal
- It can tolerate byzantine failures because the FE can collect and compare the replies it receives



Requires totally ordered reliable multicast so that all RMs perfrom the same operations in the same order

Active replication - five phases in performing a client request

- Request
  - FE attaches a unique *id* and uses *totally ordered reliable multicast* to send request to RMs. FE can at worst, crash. It does not issue requests in parallel
- Coordination
  - the multicast delivers requests to all the RMs in the same (total) order.
- Execution
  - every RM executes the request. They are state machines and receive requests in the same order, so the effects are identical. The *id* is put in the response
- Agreement
  - no agreement is required because all RMs execute the same operations in the same order, due to the properties of the totally ordered multicast.
- Response
  - FEs collect responses from RMs. FE may just use one or more responses. If it is only trying to tolerate crash failures, it gives the client the first response.

Active replication - discussion

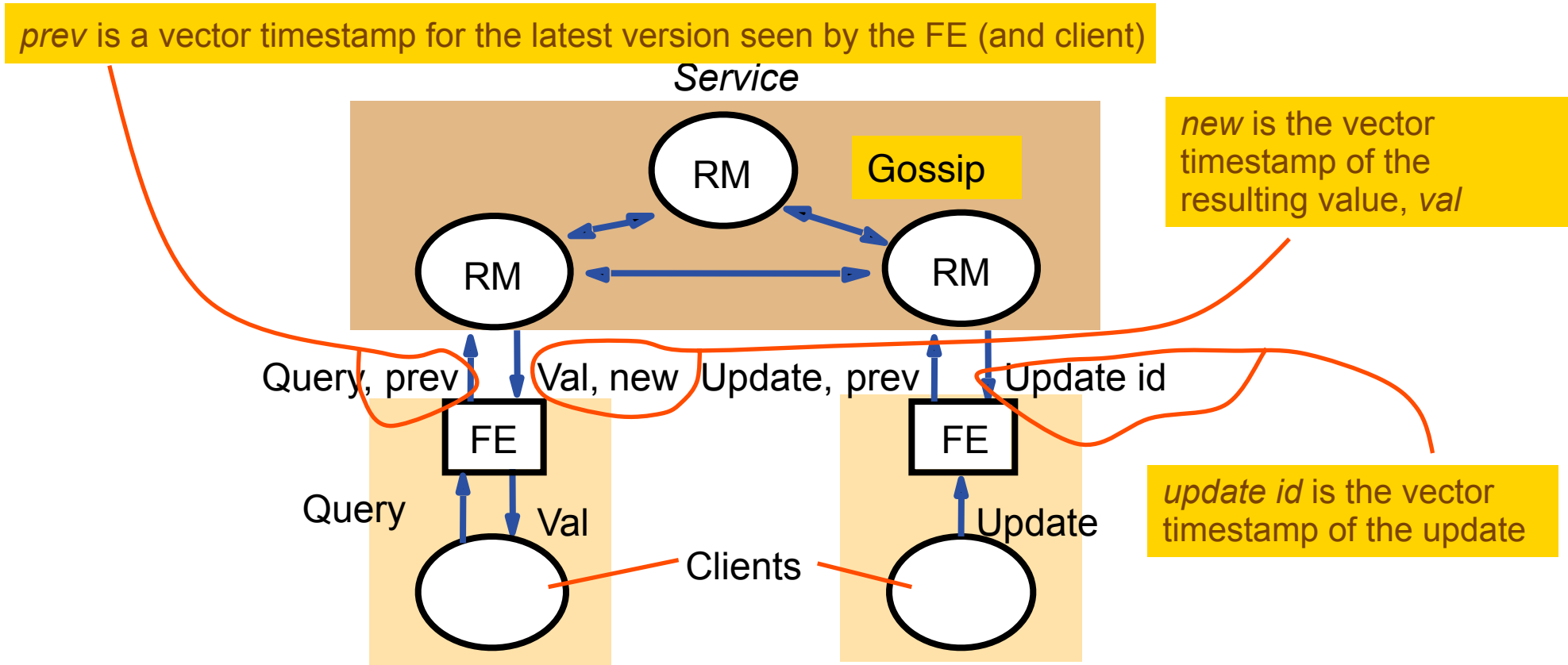
- As RMs are state machines we have sequential consistency
  - due to reliable totally ordered multicast, the RMs collectively do the same as a single copy would do
  - it works in a synchronous system
  - in an asynchronous system reliable totally ordered multicast is impossible – but failure detectors can be used to work around this problem. How to do that is beyond the scope of this course.
- this replication scheme is not linearizable
  - because total order is not necessarily the same as real-time order
- To improve performance,
  - FEs send read-only requests to just one RM

The gossip architecture

- the gossip architecture is a framework for implementing highly available services
  - data is replicated close to the location of clients
  - 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 sends queries and updates to any chosen RM
  - one that is available and gives reasonable response times
- Two guarantees (even if RMs are temporarily unable to communicate)
  - *each client gets a consistent service over time* (i.e. data reflects the updates seen by client, even if it uses different RMs). Vector timestamps are used – with one entry per RM.
  - *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.

Query and update operations in a gossip service

- 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



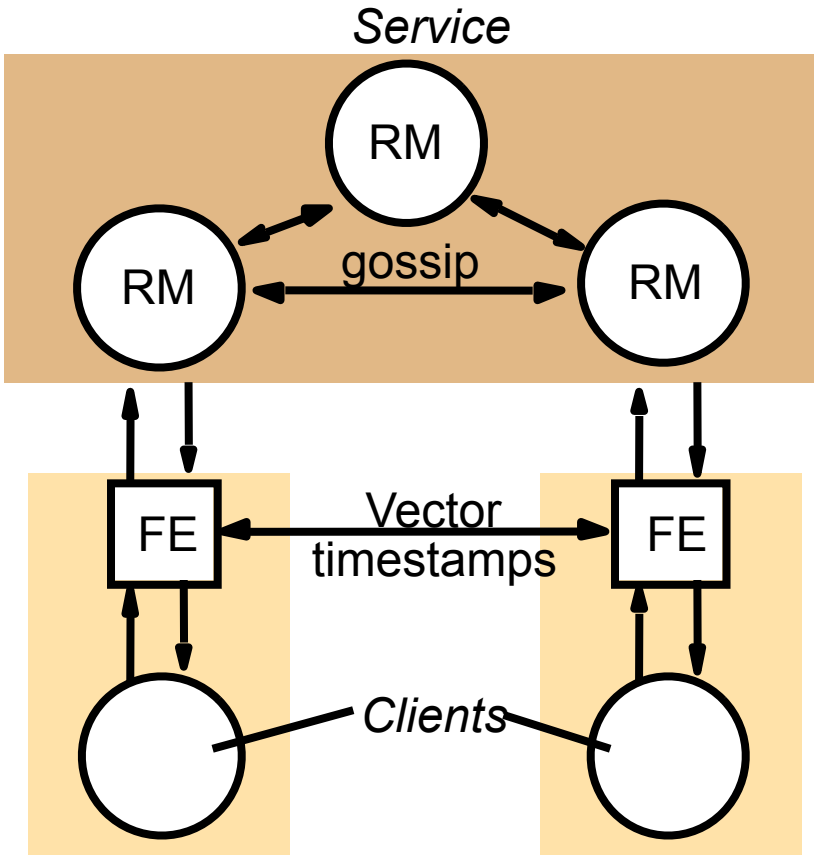
Gossip processing of queries and updates

- The five 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 seen 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

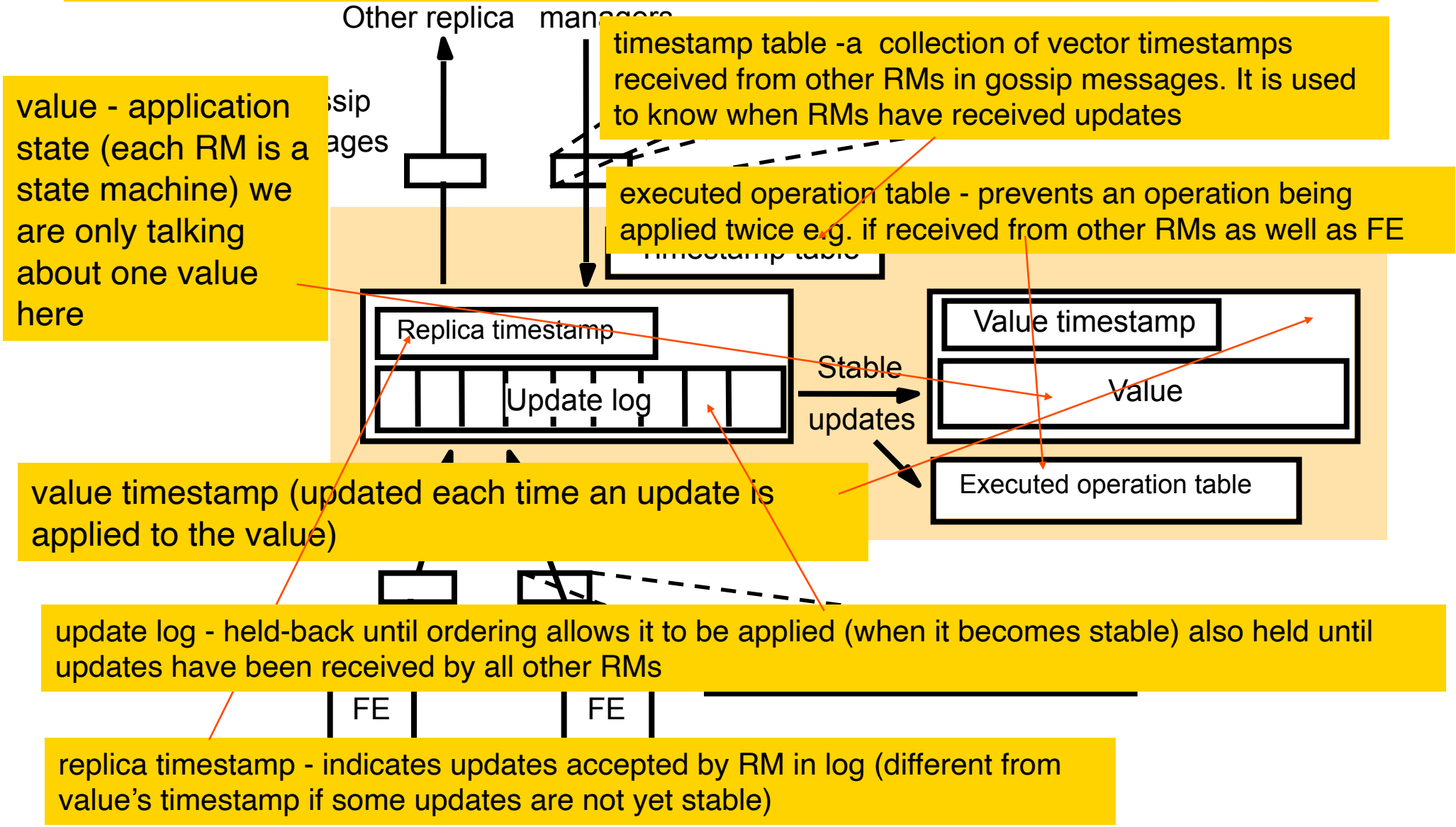
Front ends propagate their timestamps whenever clients communicate directly

- 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.



A gossip replica manager, showing its main state components



Processing of query and update operations

- Vector timestamp held by RM *i* consists of:
  - *i*th element holds updates received from FEs by that RM
  - *j*th element holds updates received by RM *j* and propagated to RM *i*
- Query operations contain *q.prev*
  - they can be applied if *q.prev* ≤ *valueTS* (value timestamp)
  - failing this, the RM can wait for gossip message or initiate them
    - e.g. if *valueTS* = (2,5,5) and *q.prev* = (2,4,6) - RM 0 has missed an update from RM 2
  - Once the query can be applied, the RM returns *valueTS* (*new*) to the FE. The FE merges *new* with its vector timestamp

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.

Gossip update operations

- Update operations are processed in causal order
  - 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 ops table and its log
  - if it is new, the RM
    - increments by 1 the *i*th 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 *ts* is calculated from *u.prev* by replacing its *i*th element by the *i*th element of the replica timestamp.
  - The RM returns *ts* to the FE,which merges it with its vector timestamp
  - For stability *u.prev* ≤ *valueTS*
  - That is, the valueTS reflects all updates seen by the FE.
  - When stable, the RM applies the operation *u.op* to the *value*,updates *valueTS* and adds *u.id* to the executed operation table.

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 log *m.log* and replica timestamp *m.ts*
- an RM receiving gossip message *m* has the following main tasks
  - merge the arriving log with its own (omit those with *ts* ≤ replicaTS)
  - apply in causal order updates that are new and have become stable
  - remove redundant entries from the log and executed operation table when it is known that they have been applied by all RMs
  - merge its replica timestamp with *m.ts*, so that it corresponds to the additions in the log

Discussion of Gossip architecture

- the gossip architecture is designed to provide a highly available service
- 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
  - it is inappropriate for updating replicas in real time (e.g. a conference)
- 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