Computer Science 425
Distributed Systems

CS 425 / CSE 424 / ECE 428

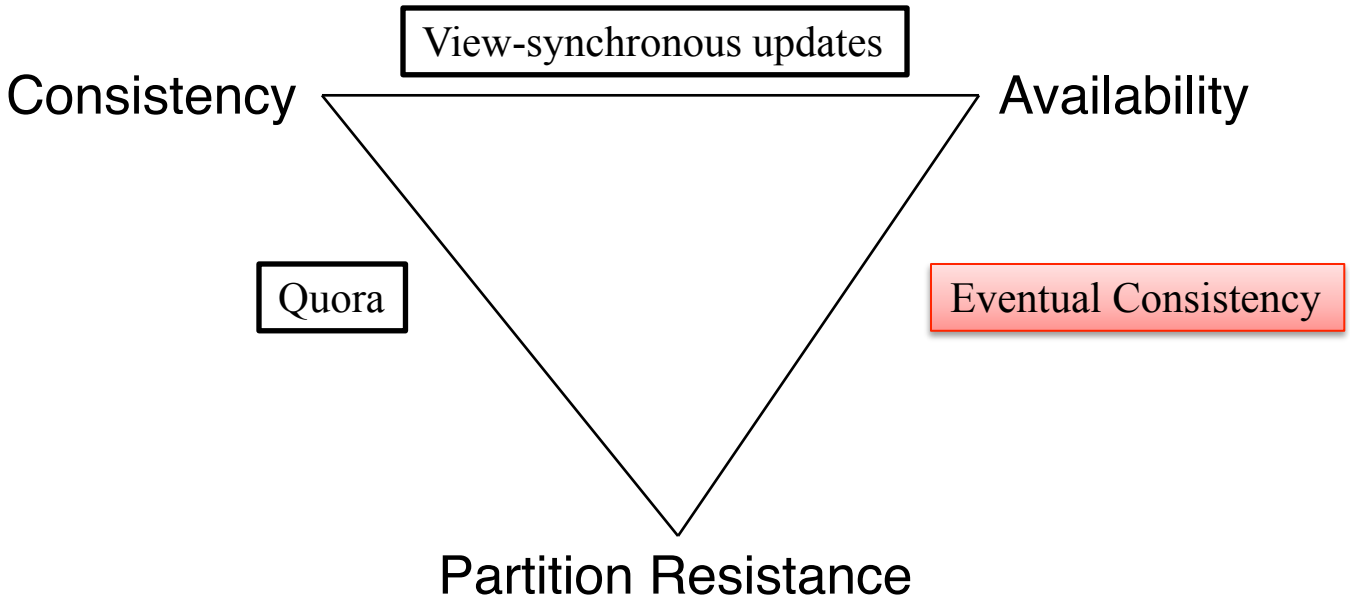
Fall 2011

Gossiping

Reading: Section 15.4 / 18.4

CAP Theorem

Can't have all 3



BASE

- Counterpart to ACID
  - **B**asically **A**vailable
  - **S**oft-state
  - **E**ventually Consistent
- Eventual consistency: *After a long enough period of no updates, all replicas will have the same view*
- Optional properties
  - Causal consistency
  - Eventual agreement even with constant updates, failures, etc.

Conflict Resolution

- Concurrent updates during partitions will cause *conflicts*

E.g., scheduling a meeting under the same time

E.g., concurrent modifications of same file

- Conflicts must be resolved, either automatically or manually

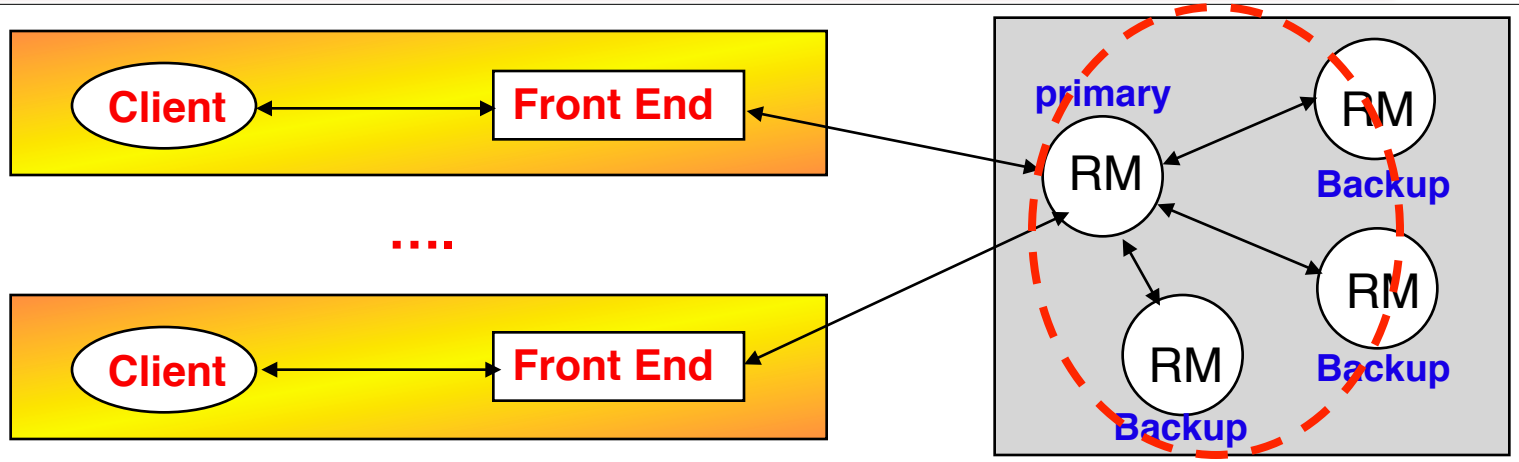
E.g., file merge

E.g., priorities

E.g., kick it back to human

- System must decide: what kind of conflicts are OK & how to minimize them

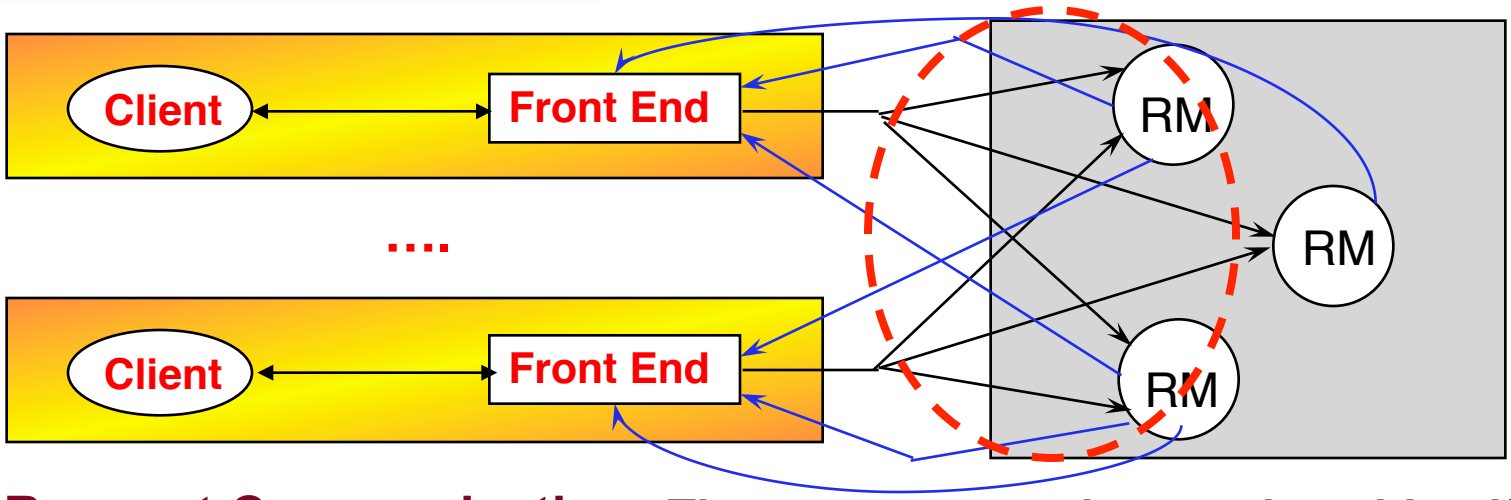
Passive (Primary-Backup) Replication ?



- **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 result to the front end

Active Replication

?

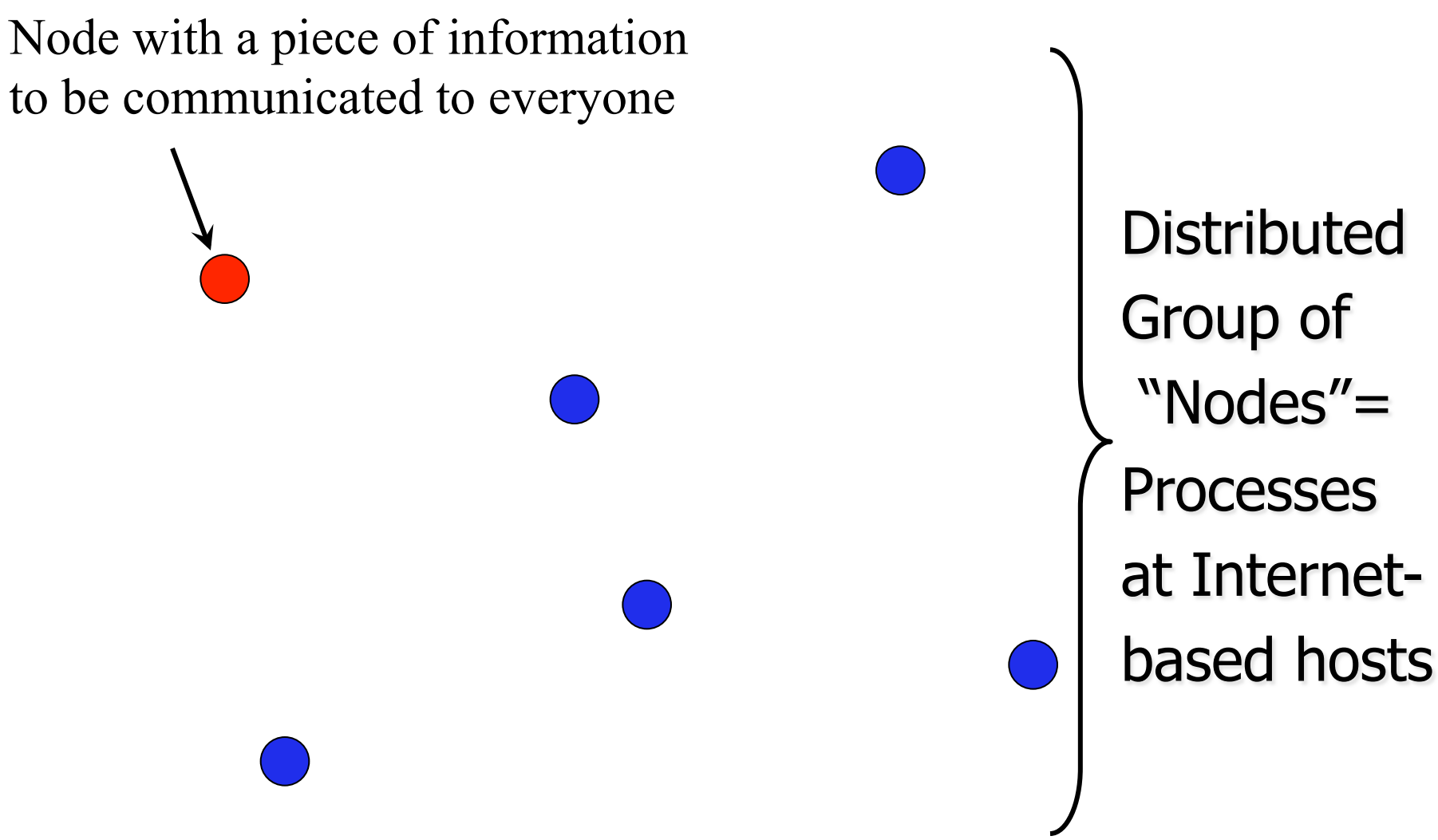


- **Request Communication:** The request contains a unique identifier and is multicast to all by a reliable totally-ordered multicast.
- **Coordination:** Group communication ensures that requests are delivered to each RM in the same order (but may be at different physical times!).
- **Execution:** Each 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

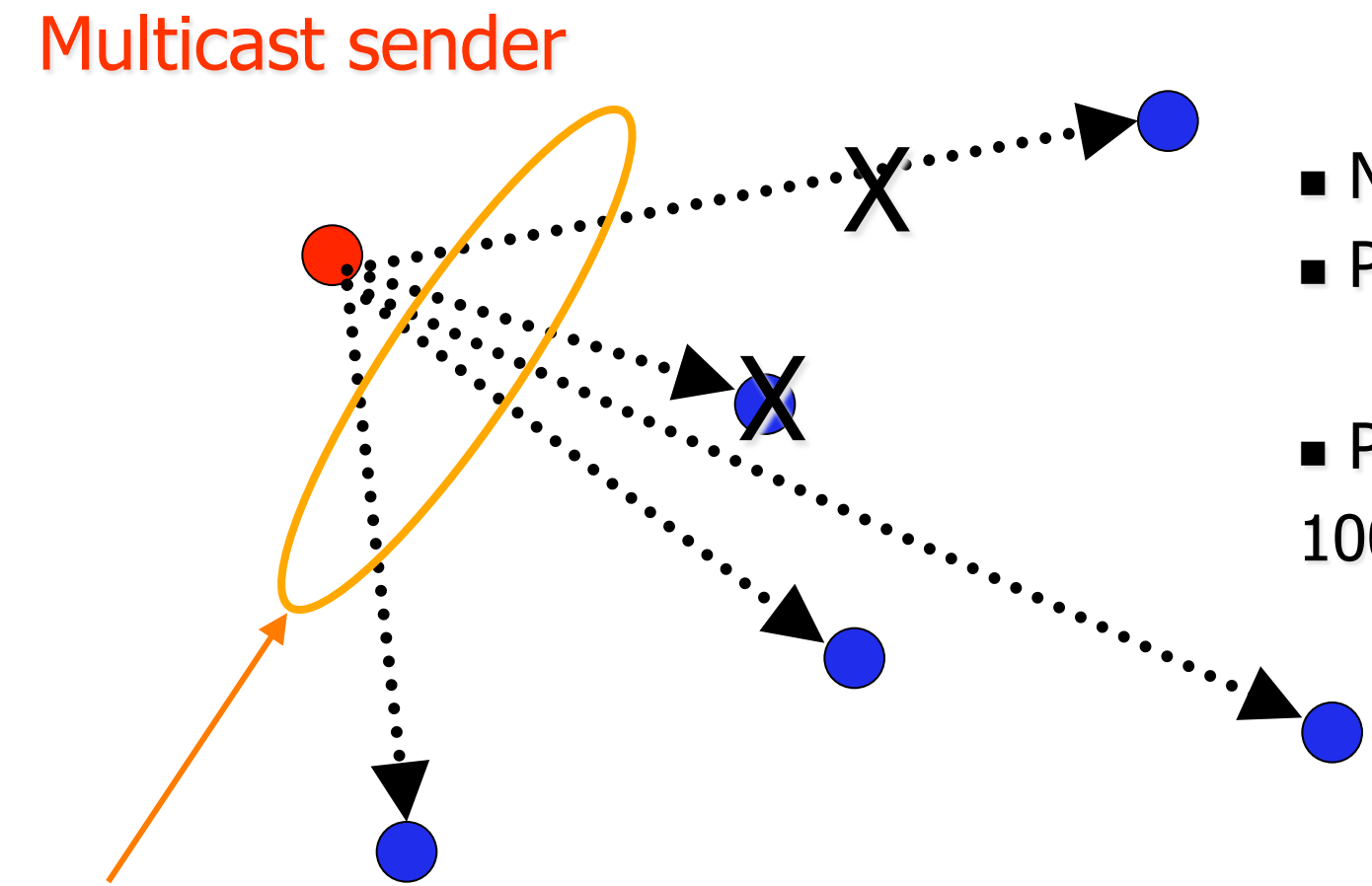
Eager versus Lazy

- **Eager replication, e.g., B-multicast, R-multicast, etc. (previously in the course)**
  - **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**
  - **Propagate updates and queries lazily, e.g., when network bandwidth available**
  - **FEs need to wait for reply from only one RM**
  - **Allow other RMs to be disconnected/unavailable**
  - **May provide weaker consistency than sequential consistency, but improves performance**
- **Lazy replication can be provided by using the gossiping**

Multicast

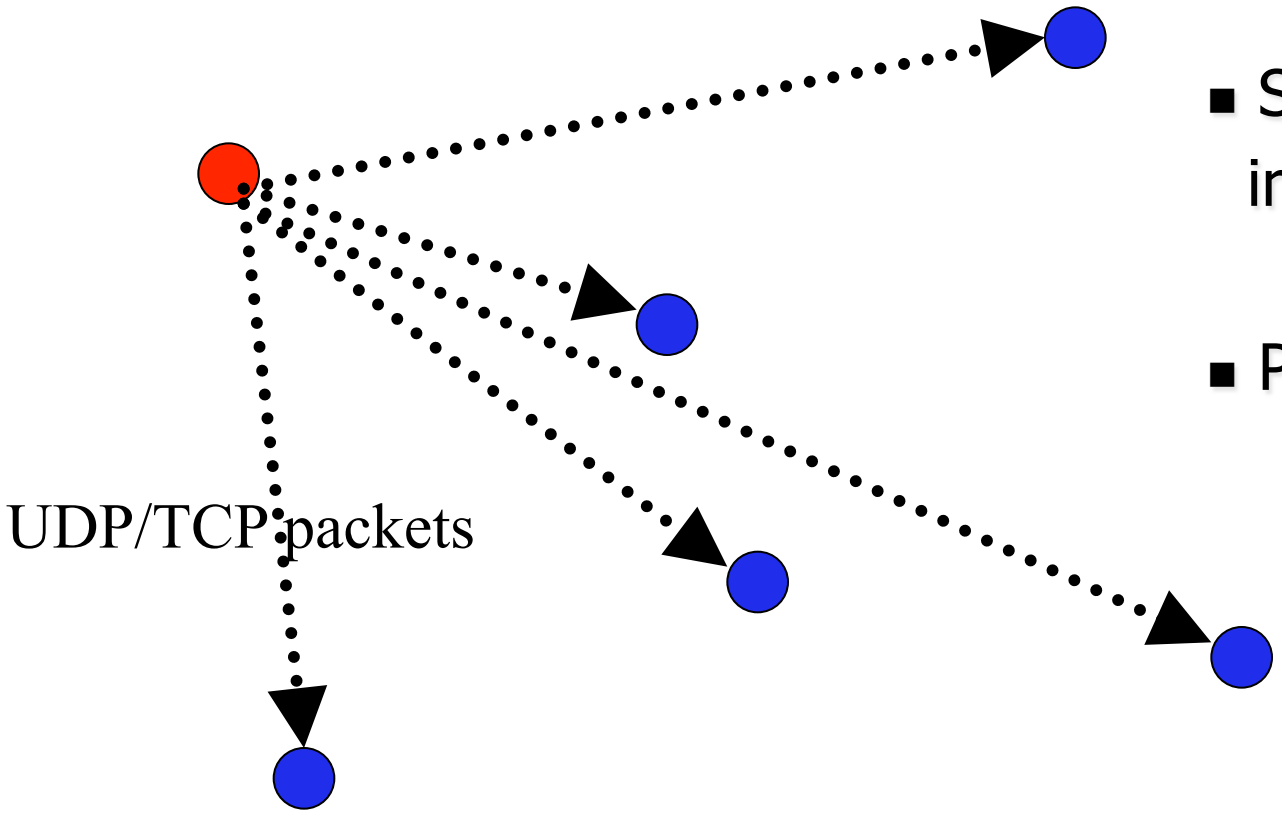

Node with a piece of information
to be communicated to everyone
Distributed
Group of
"Nodes"=
Processes
at Internet-
based hosts

Fault-tolerance and Scalability



- Nodes may crash
- Packets may be dropped
- Possibly 1000's of nodes

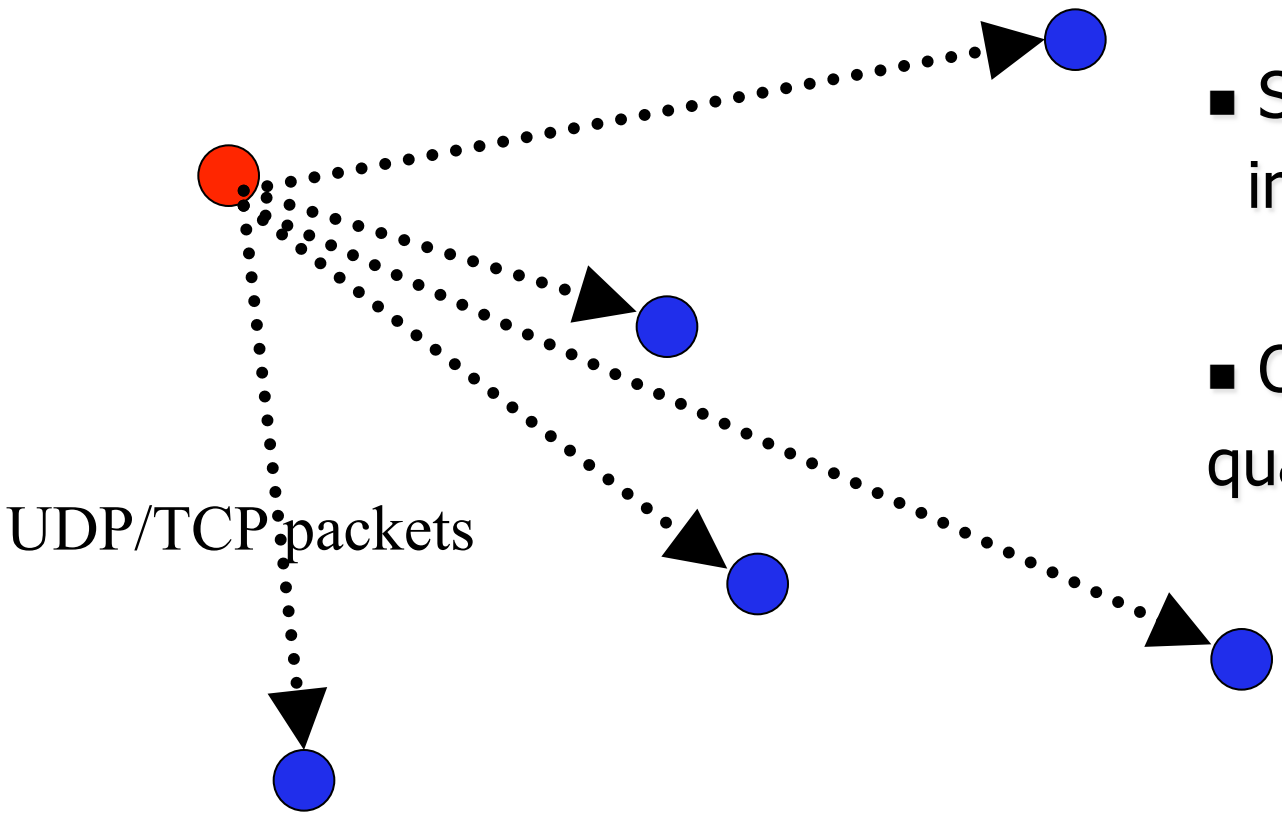
Centralized (B-multicast)



- Simplest implementation
- Problems?

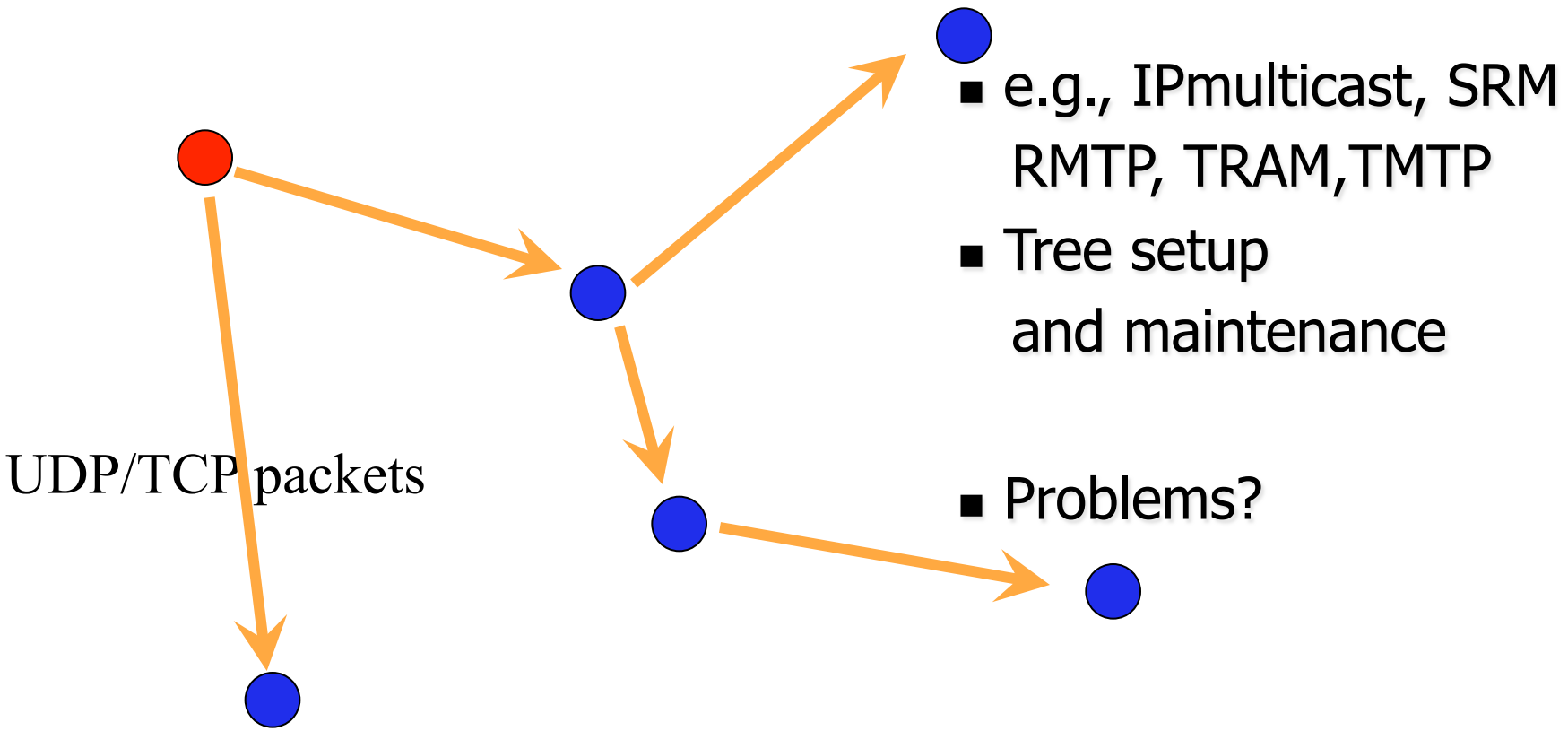
R-multicast

+ Every node B-multicasts the message



- Simpler implementation
- Overhead is quadratic in N

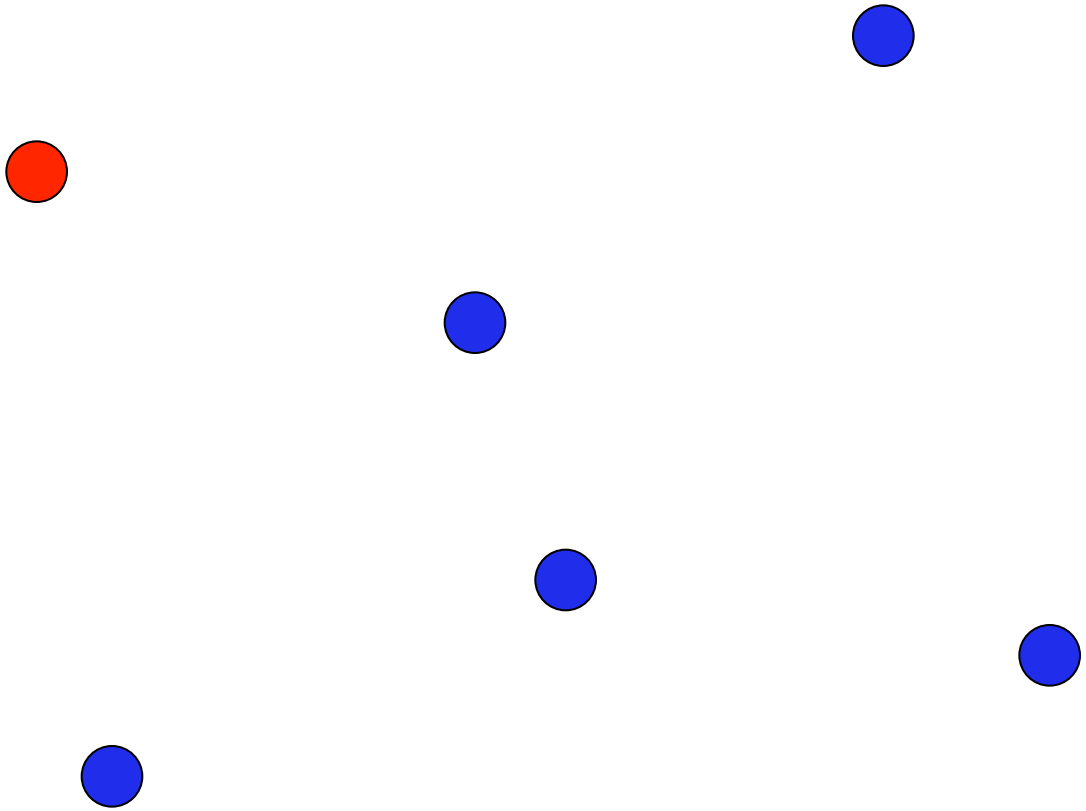
Tree-Based



- e.g., IPmulticast, SRM RMTP, TRAM,TMTP
- Tree setup and maintenance
- Problems?

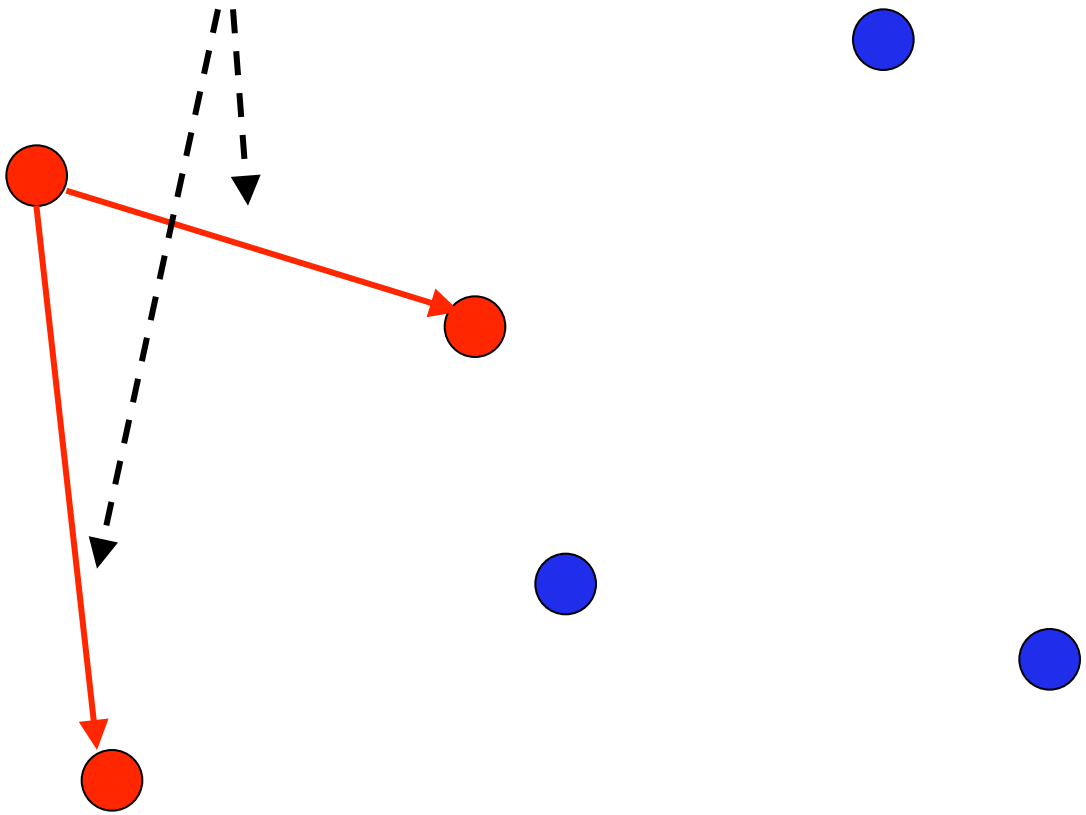
A Third Approach

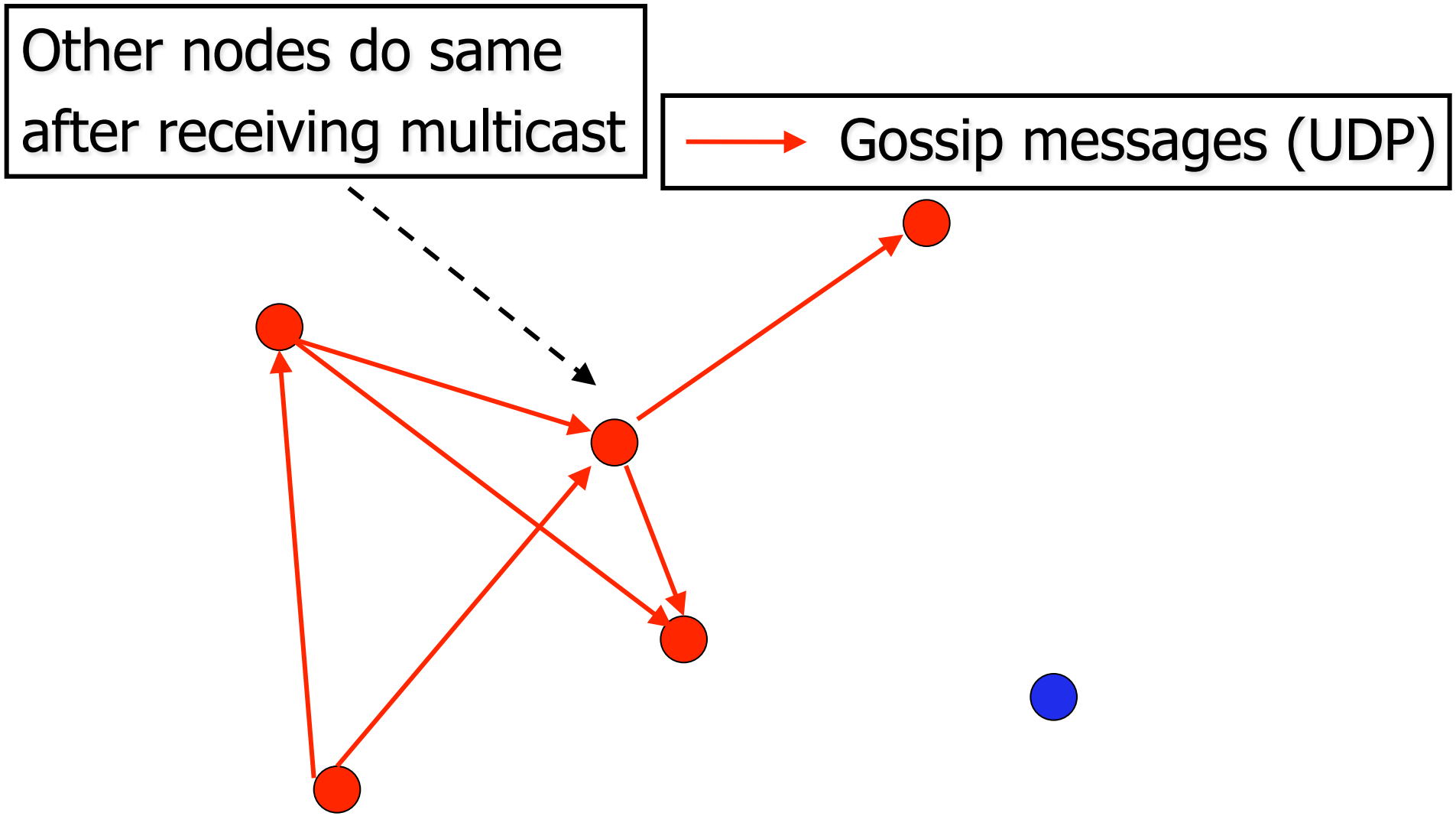
Multicast sender

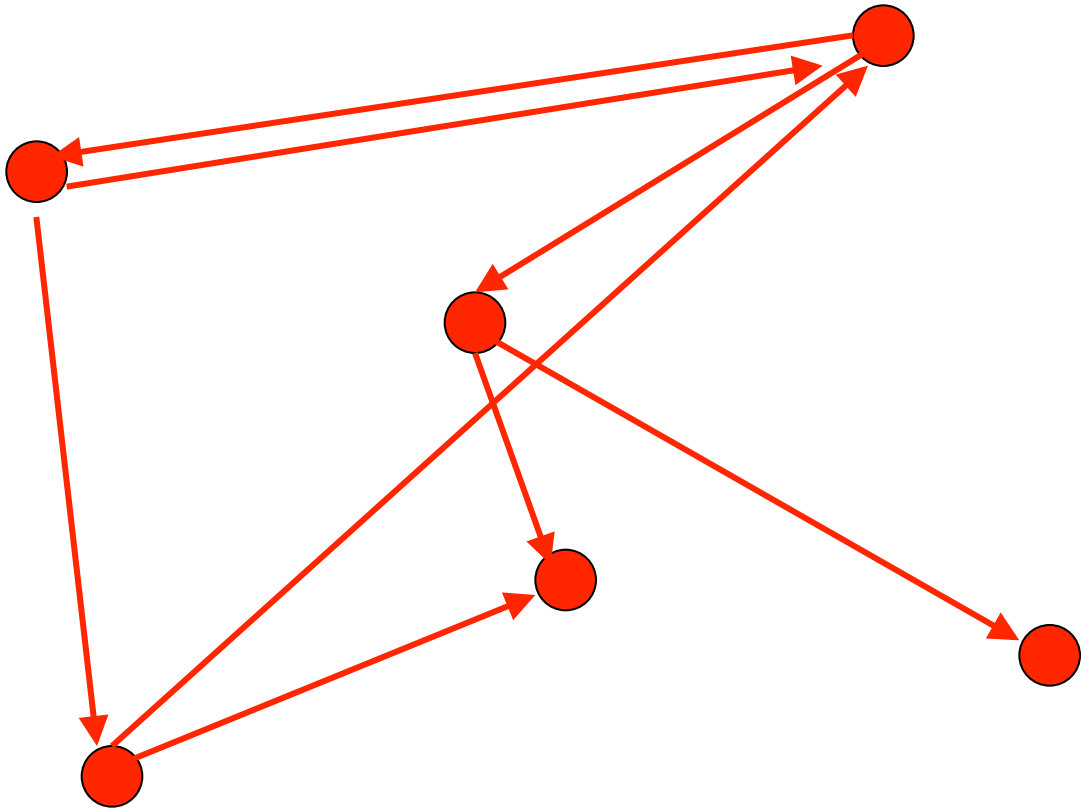


Periodically, transmit to *b* random targets

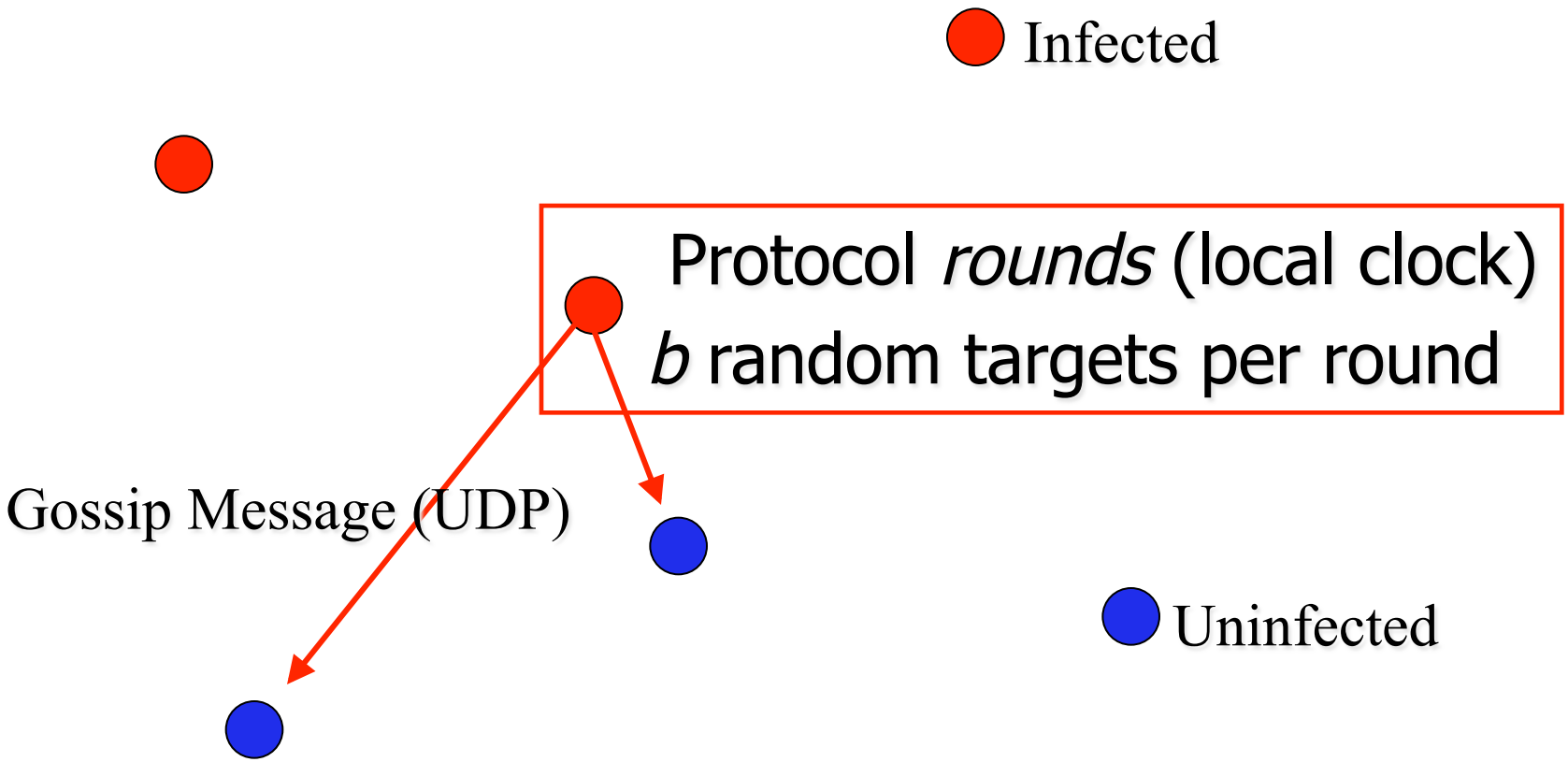
Gossip messages (UDP)




Other nodes do same after receiving multicast
Gossip messages (UDP)



"Epidemic" Multicast (or "Gossip")



Properties

Claim that this simple protocol

- **Is lightweight in large groups**
- **Spreads a multicast quickly**
- **Is highly fault-tolerant**

Analysis

From old mathematical branch of *Epidemiology* [Bailey 75]

- **Population of (*n+1*) individuals mixing homogeneously**
- **Contact rate between any individual pair is** β
- **At any time, each individual is either uninfected (numbering *x*) or infected (numbering *y*)**
- **Then,** $x_0 = n, y_0 = 1$

  and at all times

$$x + y = n + 1$$

- **Infected–uninfected contact turns latter infected**

Analysis (contd.)

- **Continuous time process**
- **Then**

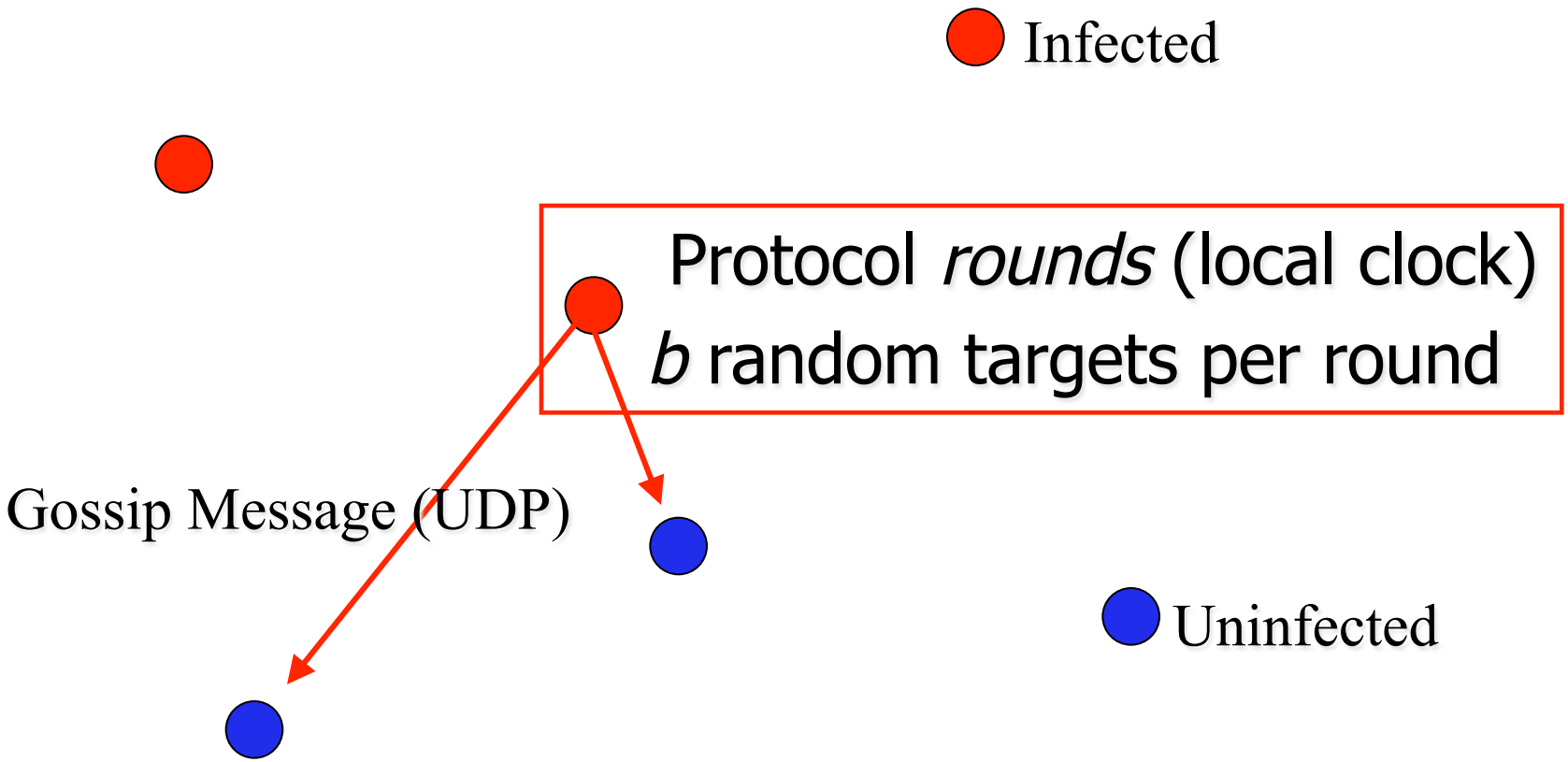
$$\frac{dx}{dt} = -\beta xy$$ **(why?)**

with solution

$$x = \frac{n(n+1)}{n + e^{\beta(n+1)t}}, y = \frac{(n+1)}{1 + ne^{-\beta(n+1)t}}$$

(what do these become when t very large?)

Epidemic Multicast



Epidemic Multicast Analysis

$$\beta = \frac{b}{n}$$ **(why?)**

Substituting, at time *t=clog(n)*, num. infected is

$$y \approx (n+1) - \frac{1}{n^{cb-2}}$$

Analysis (contd.)

- **Set *c,b* to be small numbers independent of *n***
  - ***E.g., c=2; b=2;***
- **Within *clog(n)* rounds, [**low latency**]**
  - **all but** $\frac{1}{n^{cb-2}}$ **of nodes receive the multicast** **[**reliability**]**
  - **each node has transmitted no more than *cblog(n)* gossip messages [**lightweight**]**

Fault-tolerance

- **Packet loss**
  - **50% packet loss: analyze with *b* replaced with *b/2***
  - **To achieve same reliability as 0% packet loss, takes twice as many rounds**
  - **Work it out!**
- **Node failure**
  - **50% of nodes fail: analyze with *n* replaced with *n/2* and *b* replaced with *b/2***
  - **Same as above**
  - **Work it out!**

Fault-tolerance

- **With failures, is it possible that the epidemic might die out quickly?**
- **Possible, but improbable:**
  - **Once a few nodes are infected, with high probability, the epidemic will not die out**
  - **So the analysis we saw in the previous slides is actually behavior *with high probability***

  [Galey and Dani 98]
- **Think: why do rumors spread so fast? why do infectious diseases cascade quickly into epidemics? why does a worm like Blaster spread rapidly?**

So,…

- **Is this all theory and a bunch of equations?**
- **Or are there implementations yet?**

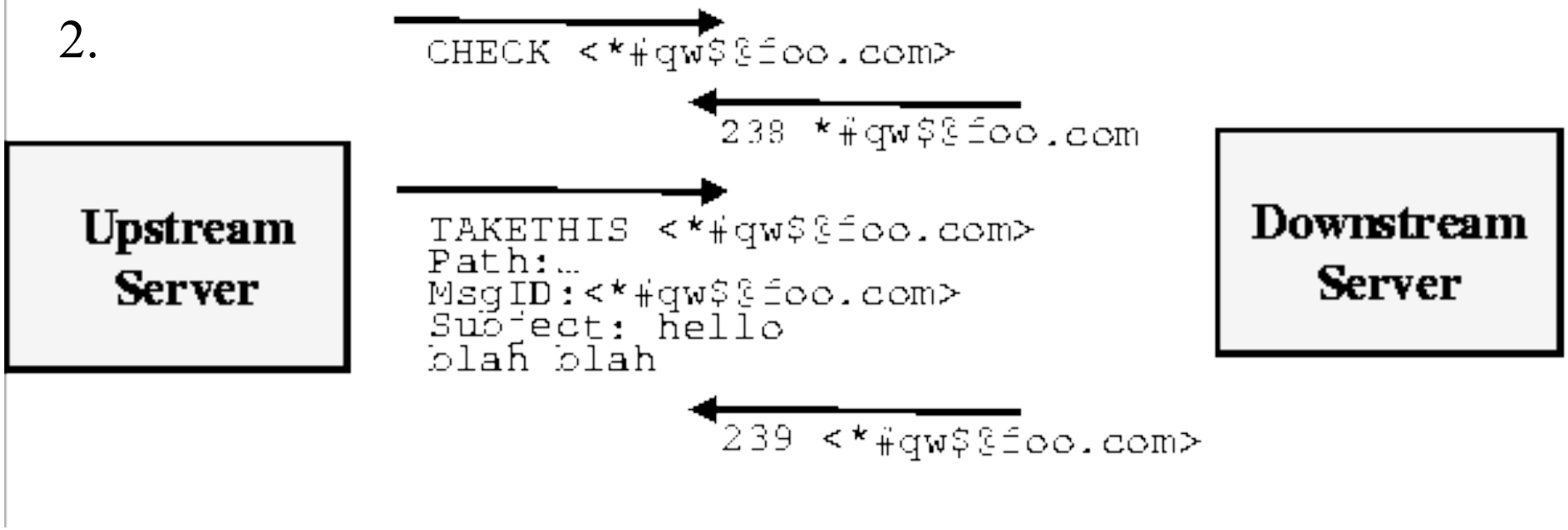
Some implementations

- **Amazon Web Services EC2/S3 (rumored)**
- **Clearinghouse project: email and database transactions [PODC ‘87]**
- **refDBMS system [Usenix ‘94]**
- **Bimodal Multicast [ACM TOCS ‘99]**
- **Ad-hoc networks [Li Li et al, Infocom ‘02]**
- **Delay-Tolerant Networks [Y. Li et al ‘09]**
- **Usenet NNTP (Network News Transport Protocol) ! [‘79]**

NNTP Inter-server Protocol

1. Each client uploads and downloads news posts from a news server

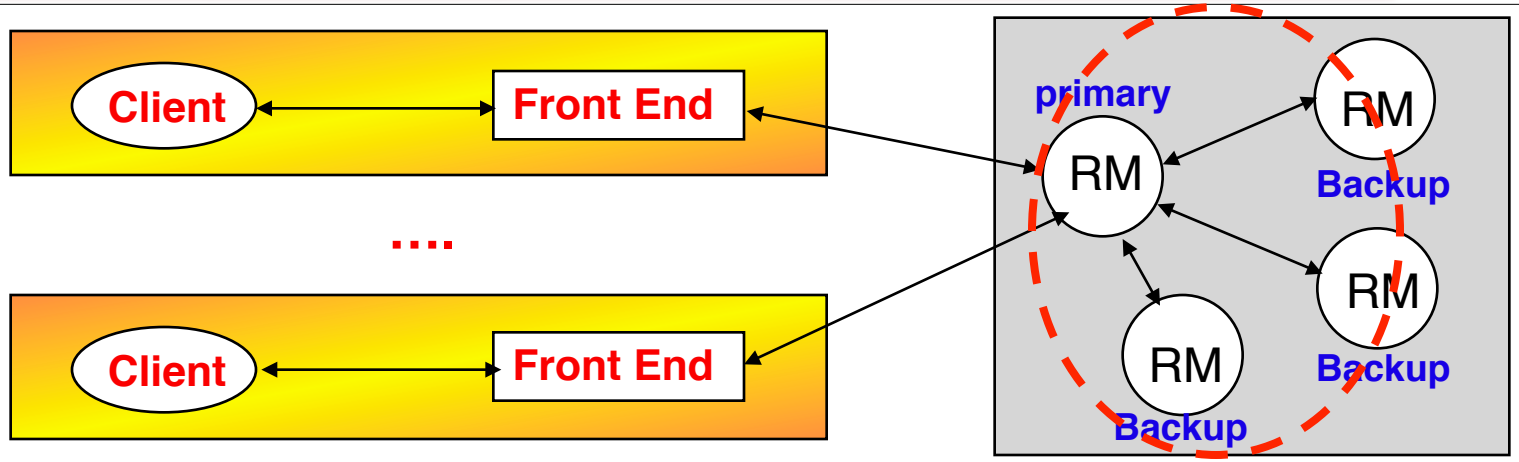
2.



Server retains news posts for a while,
transmits them lazily, deletes them after a while

Gossiping + Replicated Objects for Transactions

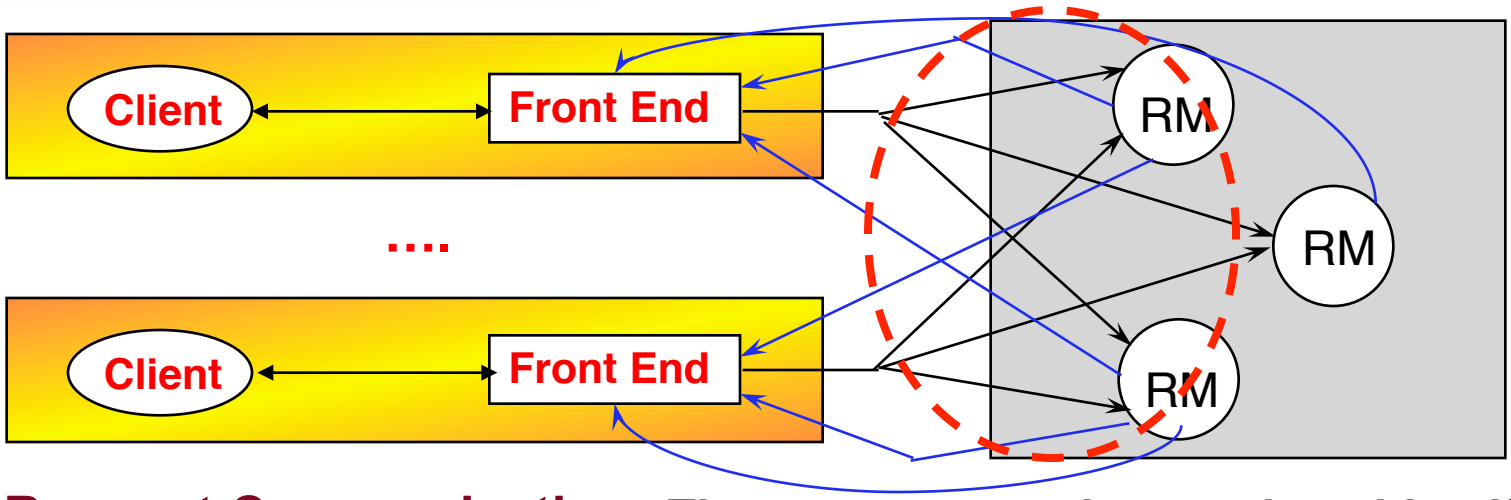
Passive (Primary-Backup) Replication ?



- **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 result to the front end

Active Replication

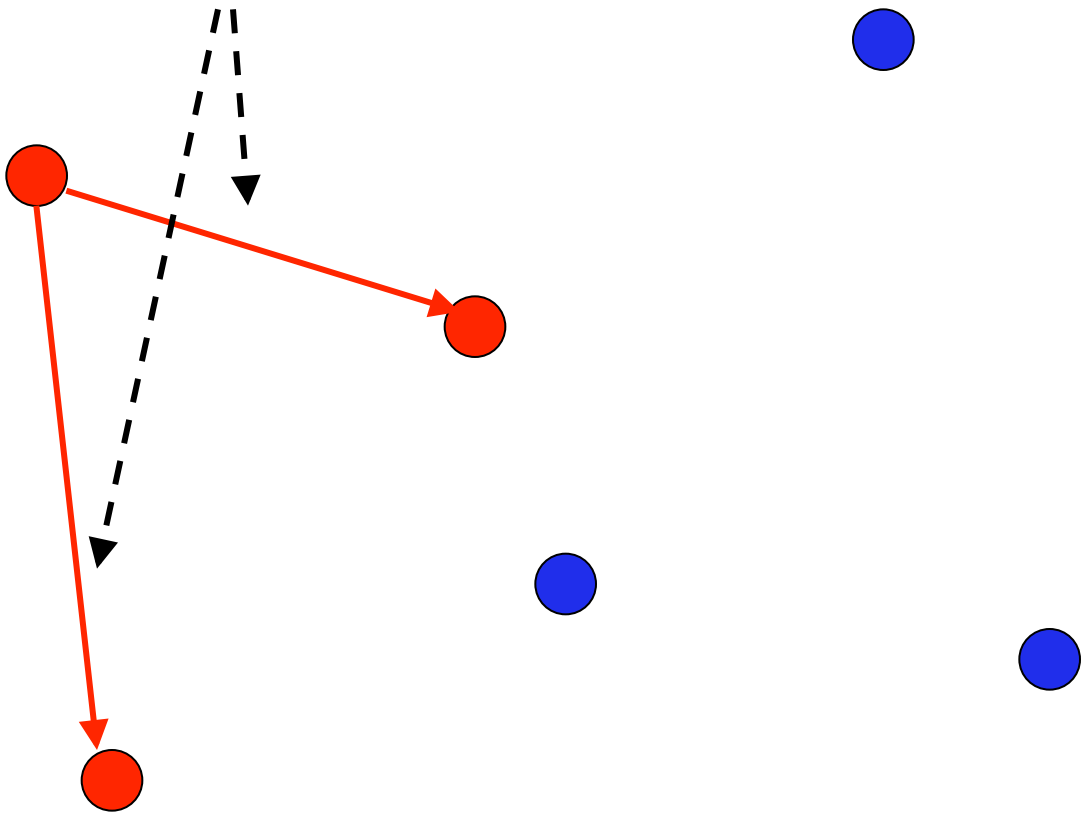
?



- **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

Periodically, transmit to *b* random targets

Gossip messages (UDP)

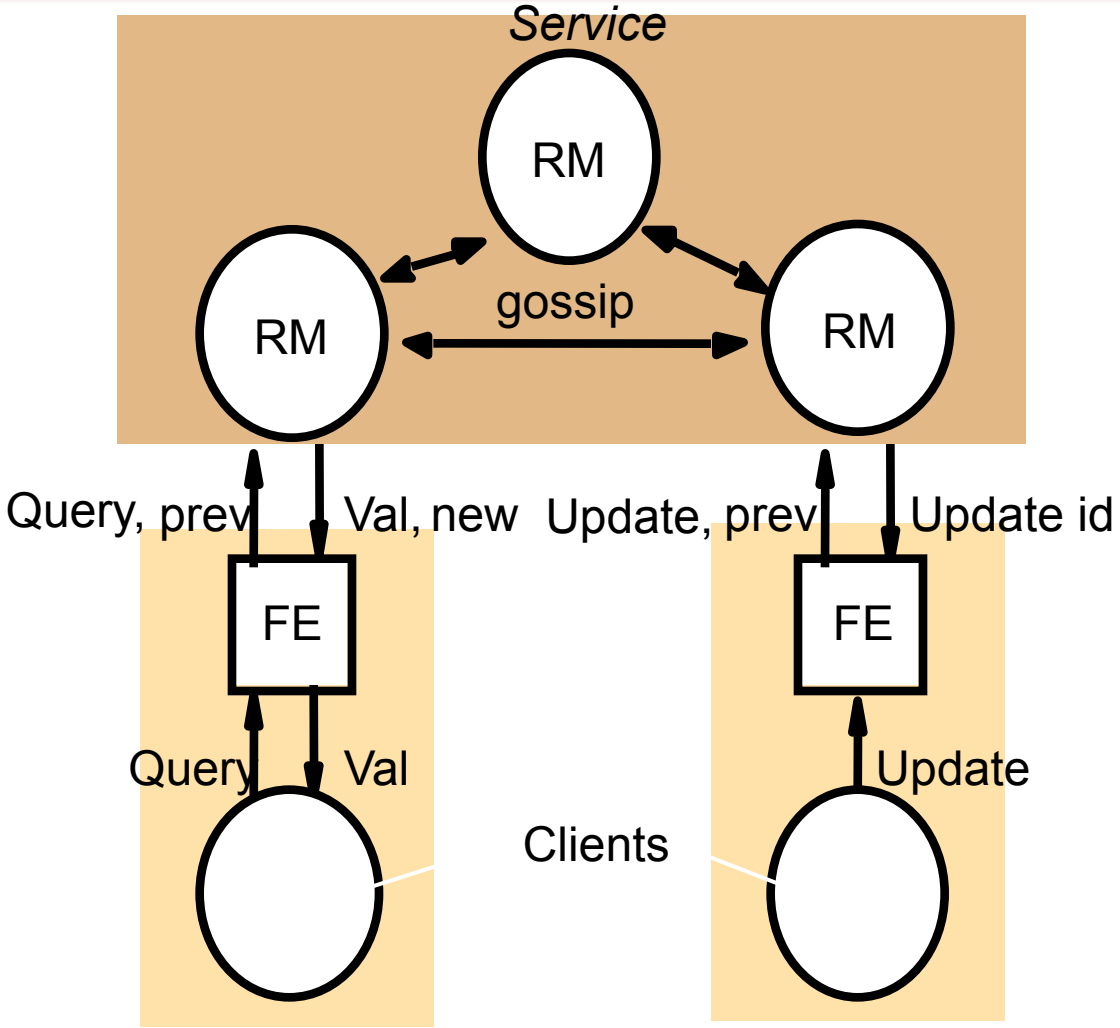


Gossiping Architecture

- **The RMs exchange "gossip" messages**

  (1) periodically and (2) amongst 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.**
- **How to provide this?**

Query and Update Operations in a Gossip Service

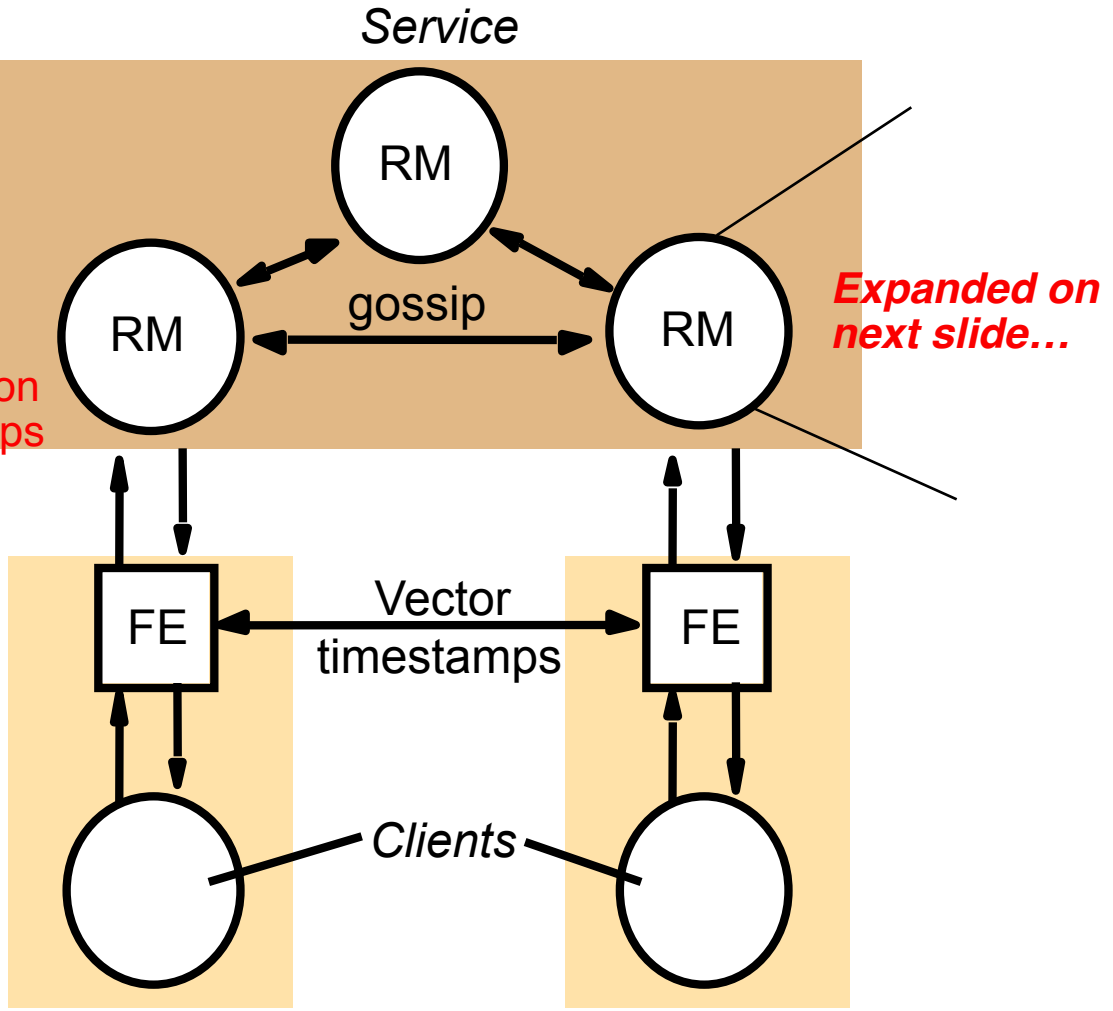


Various Timestamps

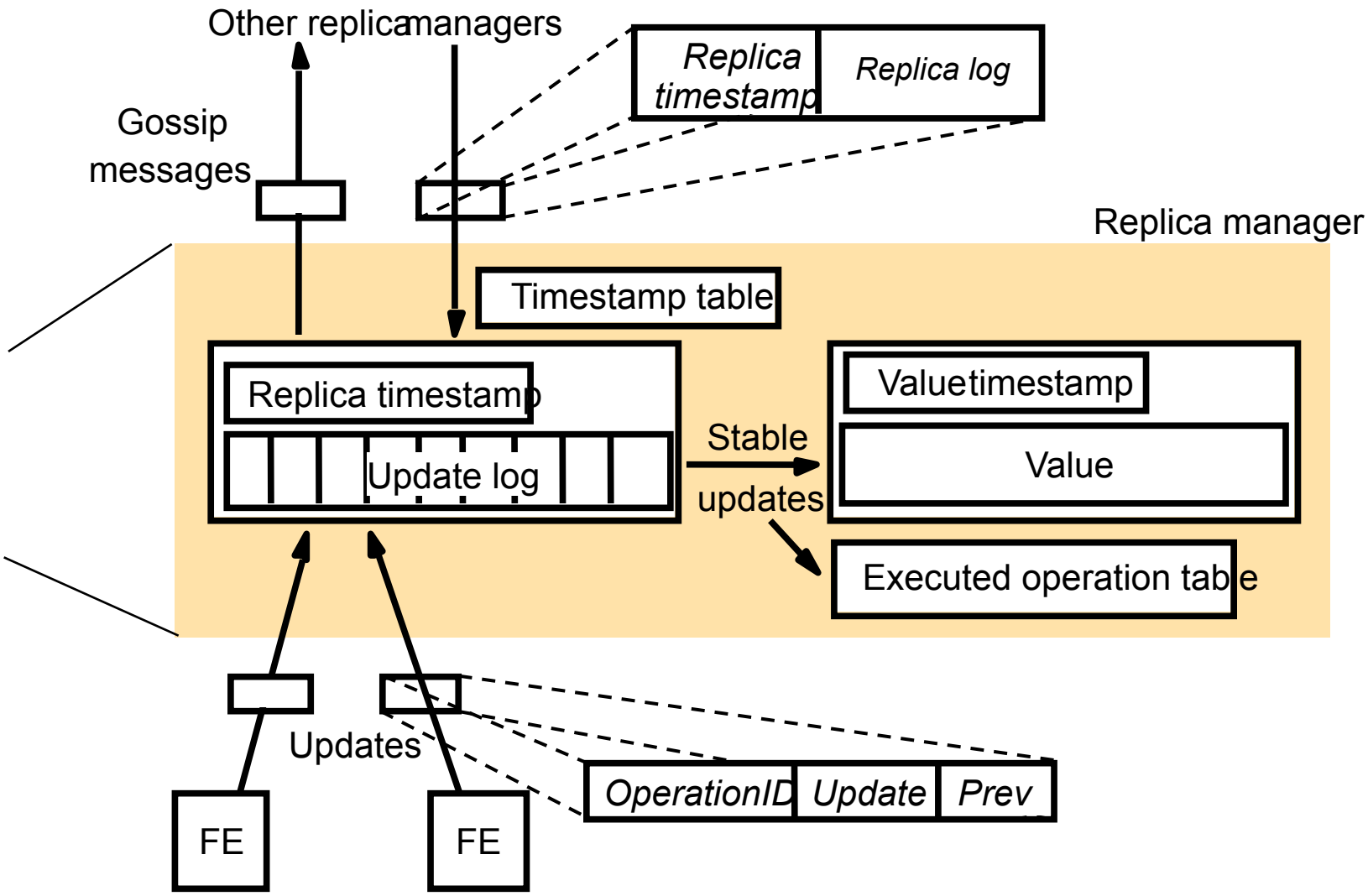
- **Virtual timestamps are used to control the order of operation processing. The timestamp contains an entry for each RM (i.e., it is a vector timestamp).**
- **Each front end keeps a vector timestamp, *prev*, that reflects the latest data values accessed by that front end. The FE sends this along with every request it sends to any RM.**
- **Replies to FE:**
  - **When an RM returns a value as a result of a query operation, it supplies a new timestamp, *new*.**
  - **An update operation returns 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.**
  - **Merging is a pairwise max operation applied to each element i (from 1 to N)**

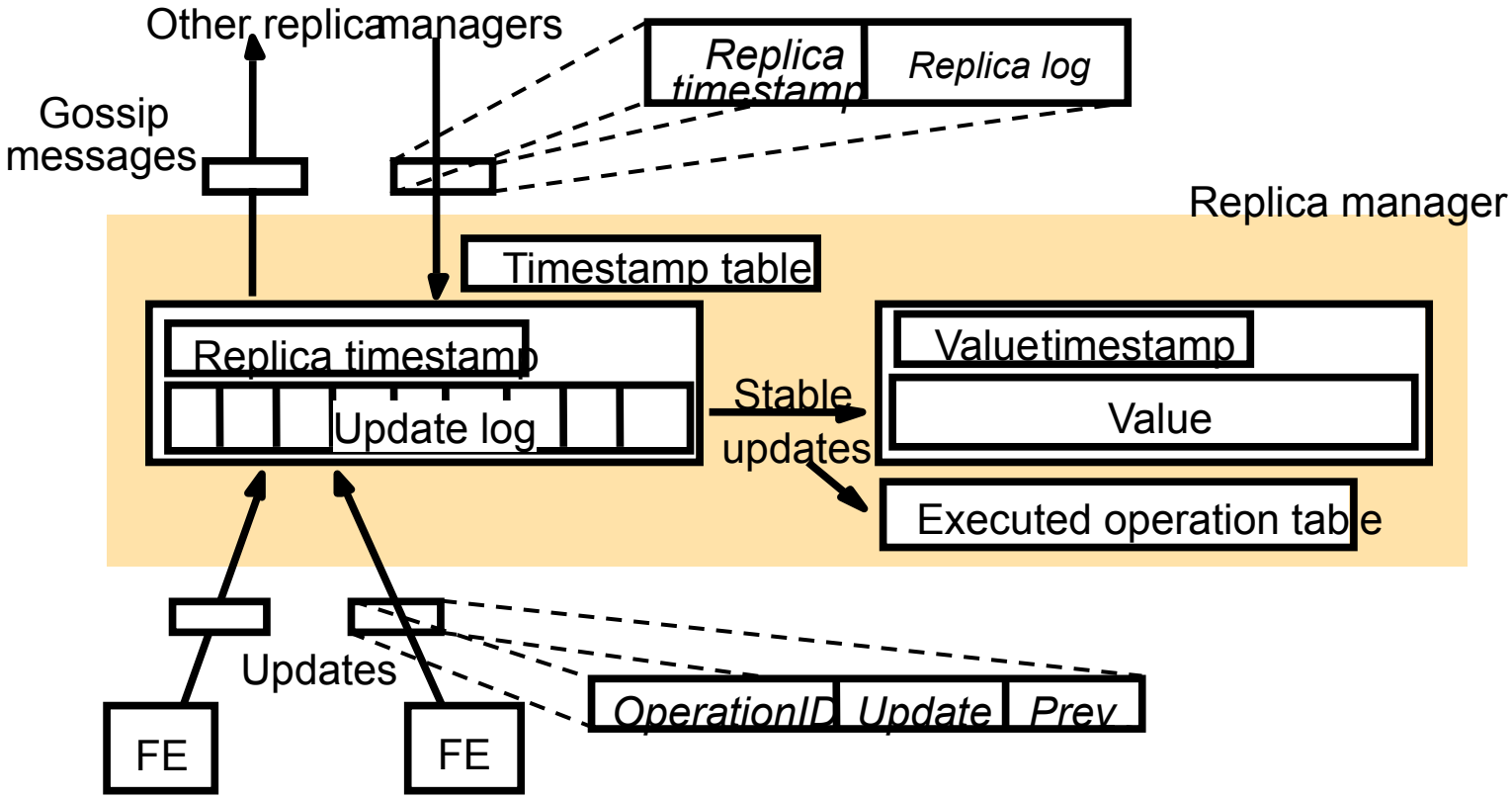
Front ends Propagate Their Timestamps

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.

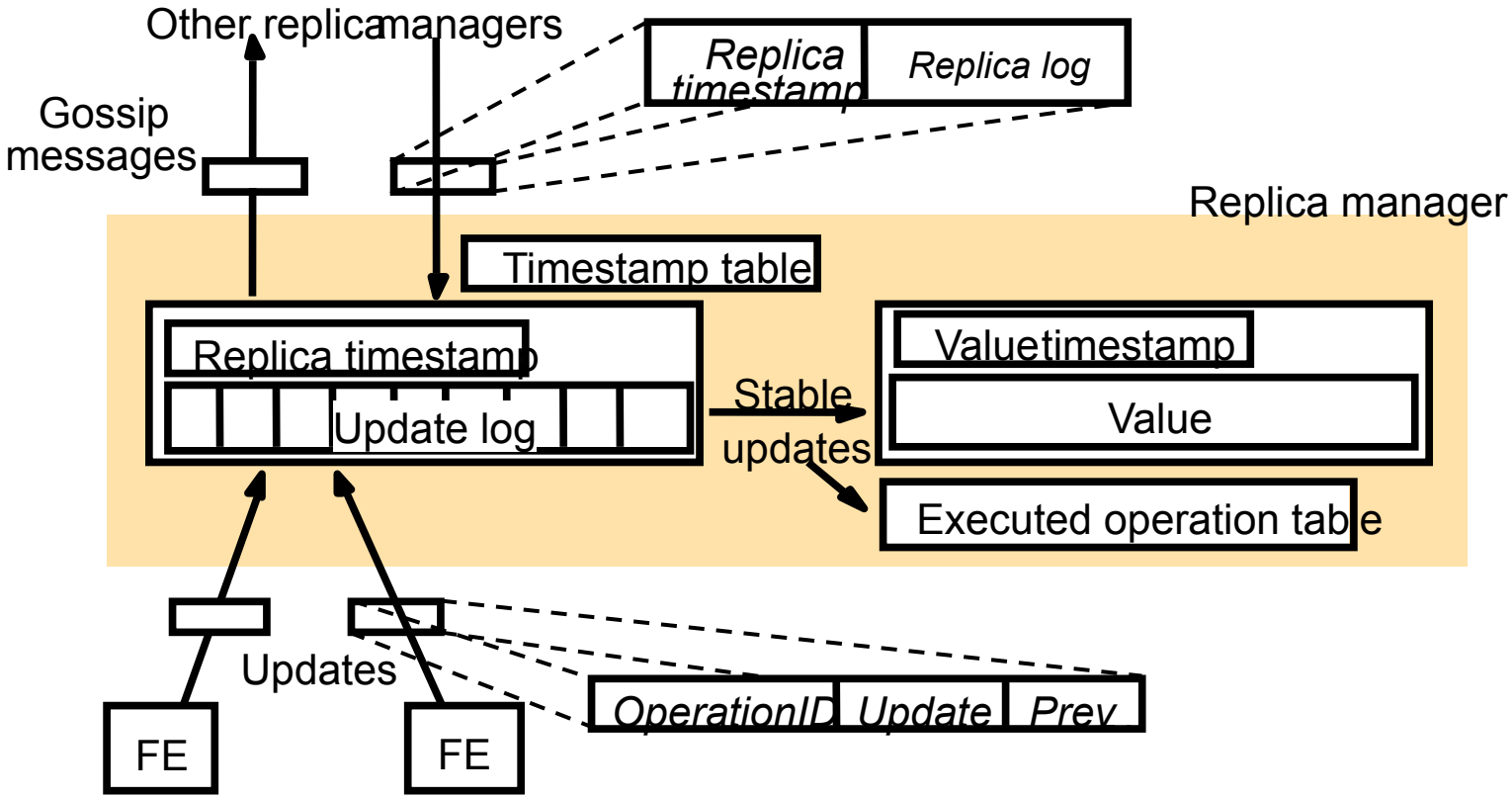


A Gossip Replica Manager

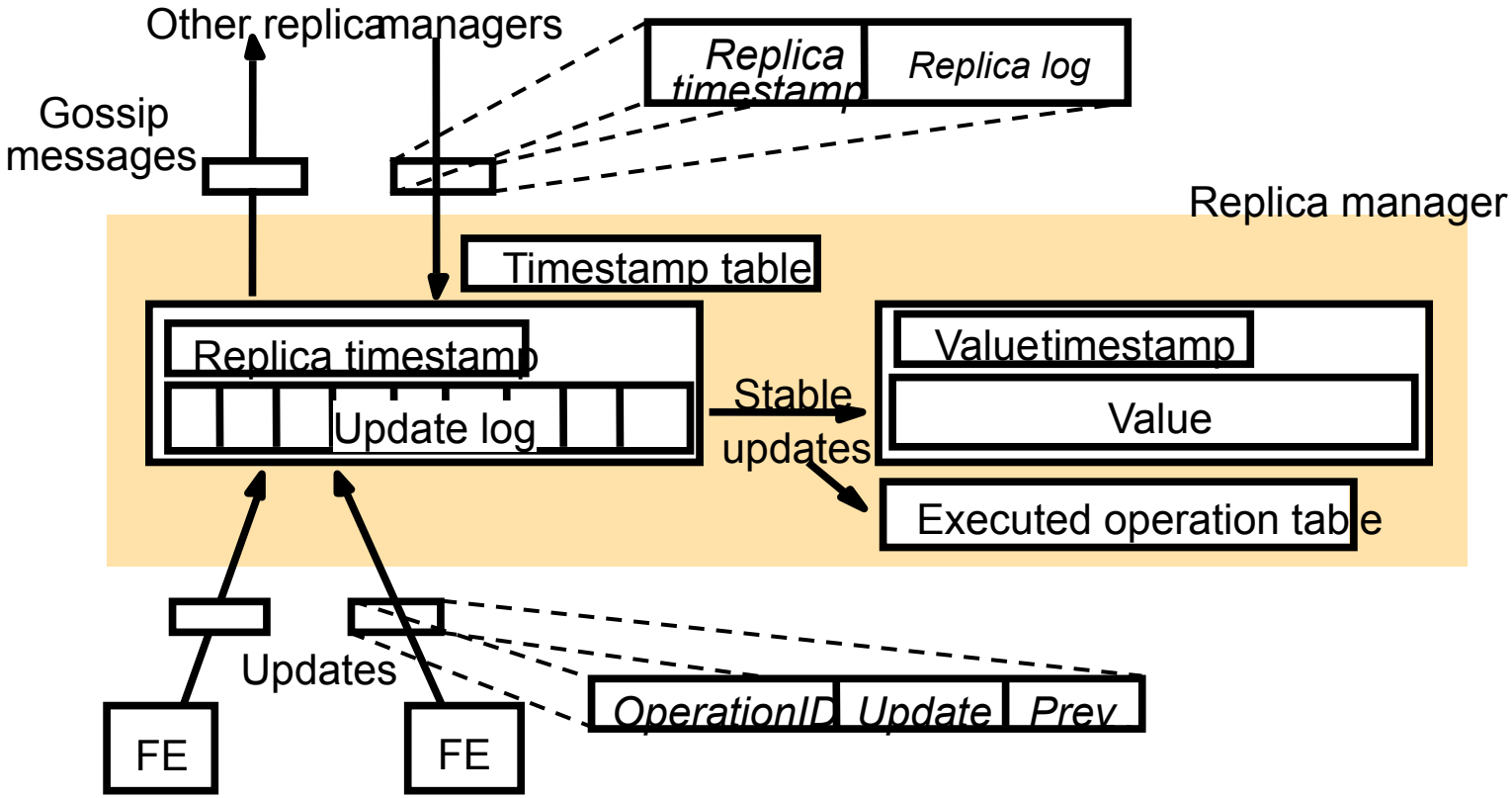




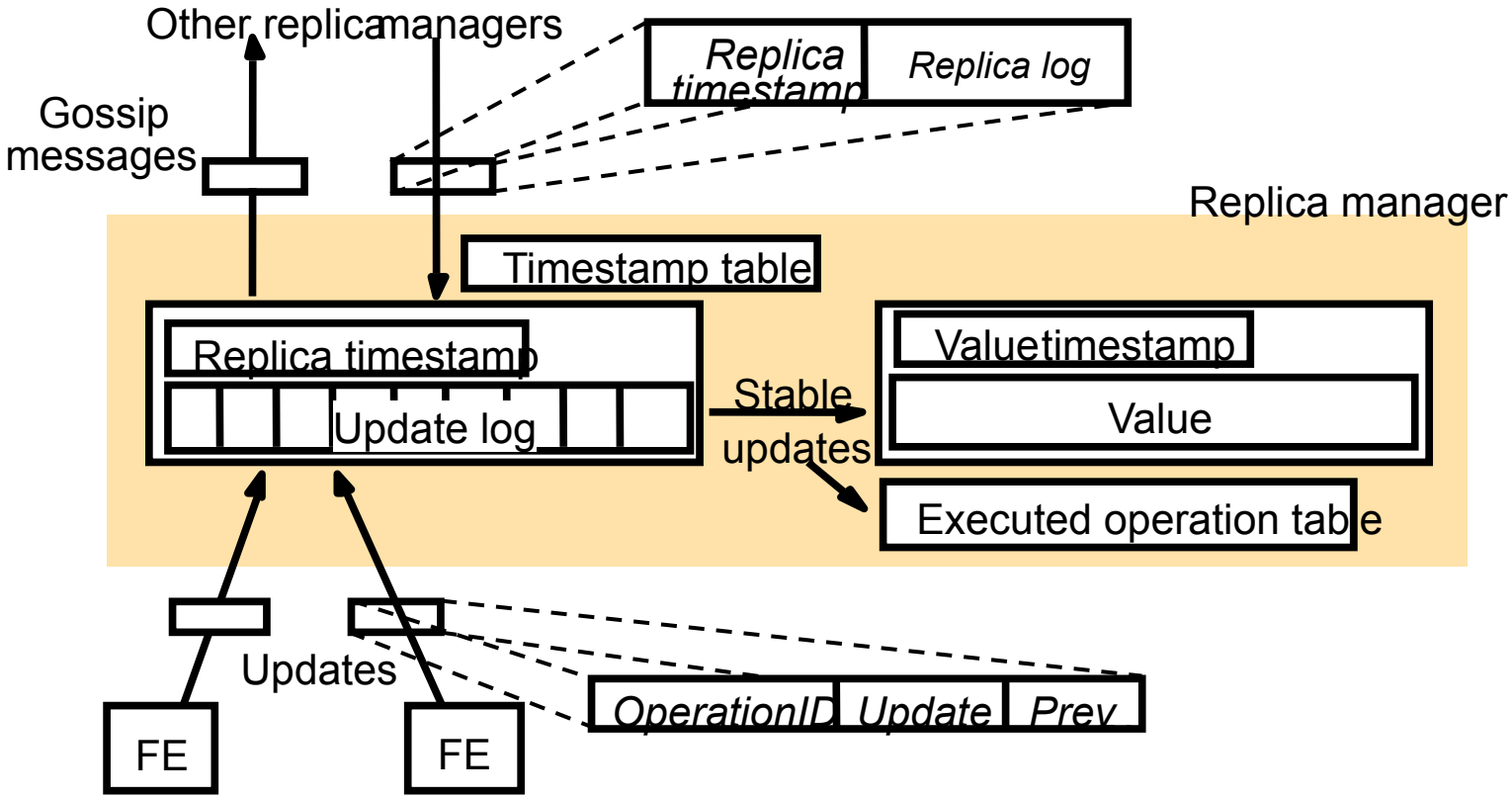
- **Value: value of the object maintained by the RM.**
- **Value timestamp: the timestamp that represents the updates reflected in the value. Updated whenever an update operation is applied.**



- **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 total 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 that have been forwarded to RM_i *in gossip messages.***

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 in 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 *m* 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 to 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.**

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.

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), or a 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.**
- ***Example: Network News Transport Protocol (NNTP) uses gossip communication. Your updates to class.cs425 are spread among News servers using the gossip protocol!***
- **Gives probabilistically reliable and fast dissemination of data with very low background bandwidth**
  - **Analogous to the spread of gossip in society.**

More Examples

- **Bayou**
  - **Replicated database with weaker guarantees than sequential consistency**
  - **Uses gossip, timestamps and concept of *anti-entropy***
  - **Section 15.4.2 / 18.4.2**
- **Coda**
  - **Provides high availability in spite of disconnected operation, e.g., roving and transiently-disconnected laptops**
  - **Based on AFS**
  - **Aims to provide *Constant data availability***
  - **Section 15.4.3 / 18.4.3**