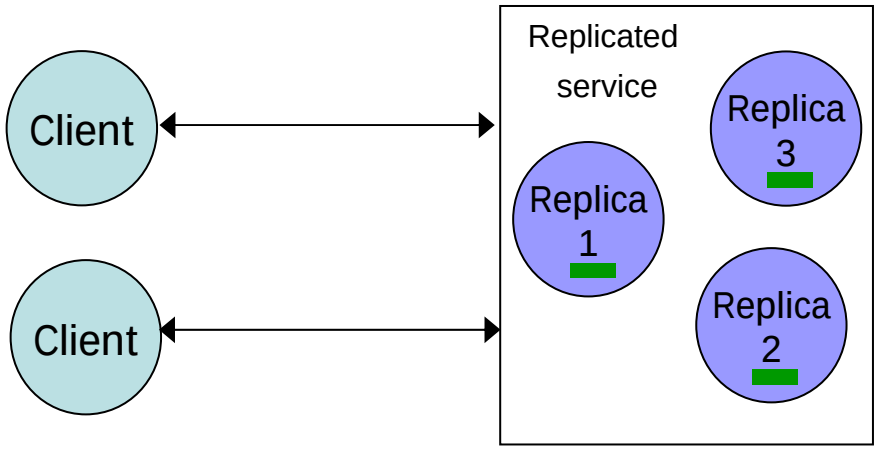
Replication

Brian Nielsen
bnielsen@cs.aau.dk

Service Improvements

- Replication is a key technology to enhance service



- Performance enhancement
  - Load-balance
  - Proximity-based response
  - Example
    - caches in DNS servers / file servers (NFS/AFS)
    - replicated web servers

Service Improvements

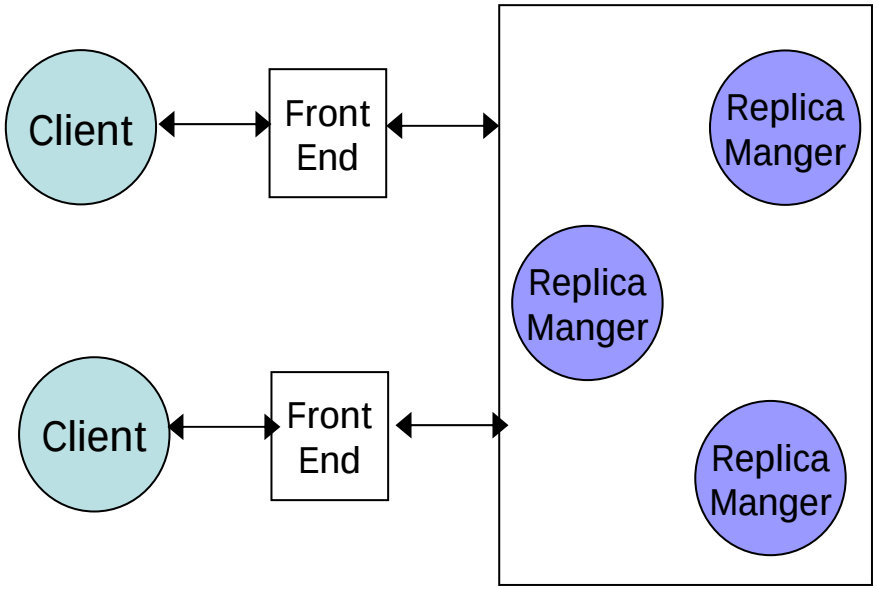
- Increase availability
  - Server failures, Network partitions
  - Availability (Uptime): $1 - p^{n:}$
  - The availability of the service that have *n* replicated servers each of which would crash in a probability of *p*

N (p=0.05)	Availability	Down time
1	95%	18 days / Y
2	99.75%	1 day / Y
3	99.99	1h / Y
4	99.999	3 min / Y

- Fault tolerance
  - Guarantee strictly correct behavior despite a certain number and type of faults
  - Strict data consistency between all replicated servers

Basic Architectural Model

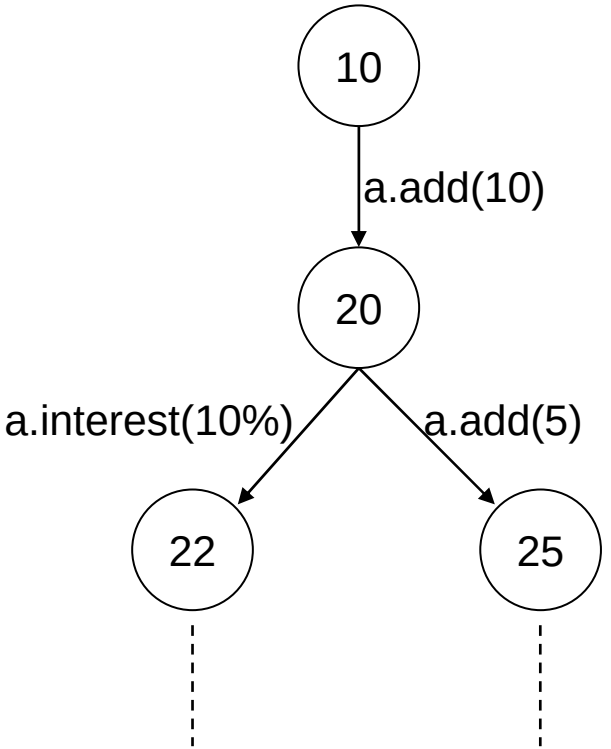
- **Requirements**
  - Transparency: no need to be aware of multiple replicas.
  - Consistency: data consistency among replicated files.



Operations

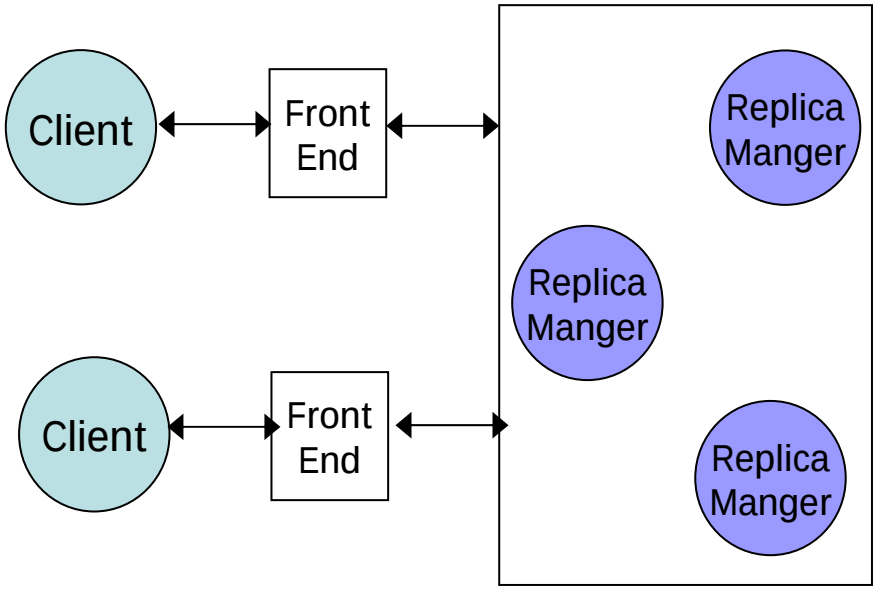
- Client performs operations on a replicated object **obj.m(…)**
  - Executed atomically
- ***"state machine objects":*** state depends only on initial state and sequence of operations (deterministic function)
  - Precludes that operations depend on external inputs such as system clock and sensor values
- Updates vs. queries (read-only)

- Single operations vs. sequence (transactions (15.5))



Basic Architectural Model

- **Requirements**
  - –Transparency: no need to be aware of multiple replicas.
  - –Consistency: data consistency among replicated files.



General Phases in an Replication alg.:

1. **Request:** send a client request to a manager.
2. **Coordination:** decide on delivery order of the request.
3. **Execution:** process a client request but not permanently commit it.
4. **Agreement:** agree on outcome and if the execution will be committed
5. **Response:** respond to the front end

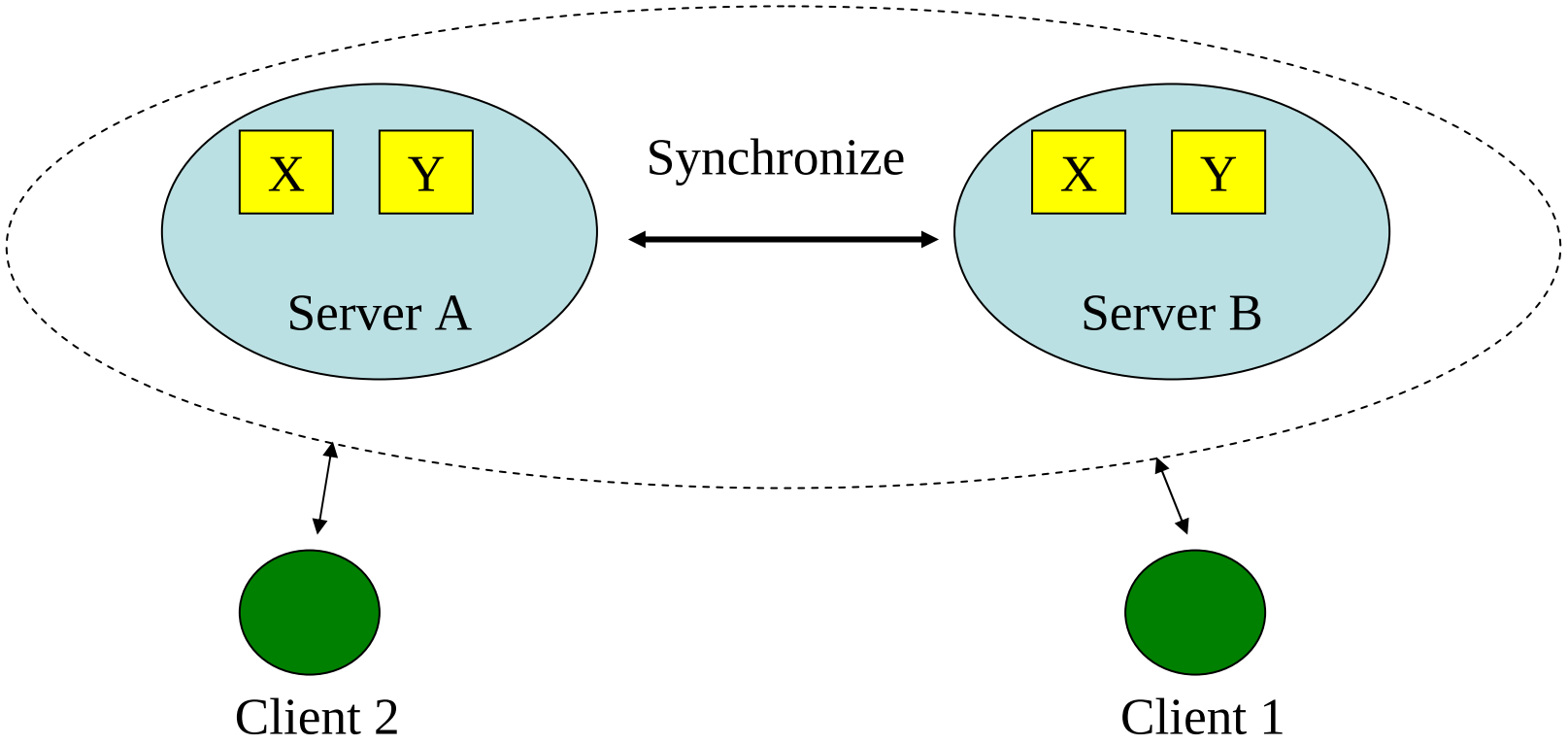
Fault Tolerance

Fault-tolerance

- ***Provide uninterrupted correct service even in the presence of server failures***
- A service based on replication is correct if it keeps responding despite failures,
- and if clients cannot tell the difference between the service they obtain form an implementation with replicated data and one provided by a single correct replica manager (***Consistency***).

An example of inconsistency between two replications

- Each of computer A and B maintains replicas of two bank accounts x and y
- Client accesses any one of the two computers, updates synchronized between the two computers



An example of inconsistency between two replications

Initially x=y=$0

Client1:

$setBalance_B(x,1)$

Server B failed…

$setBalance_A(y,2)$

Client2:

$getBalance_A(y)=2$

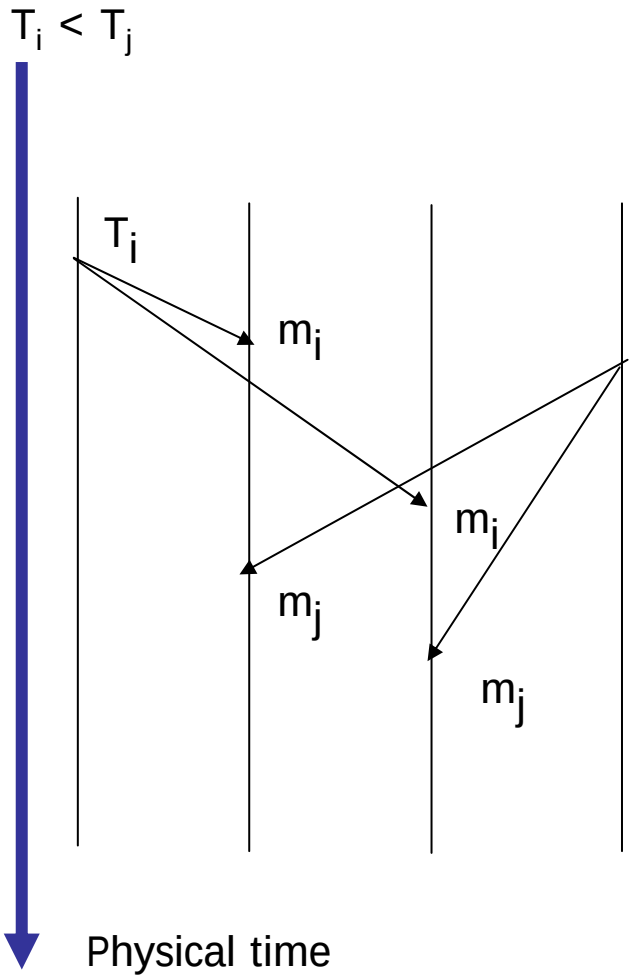
$getBalance_A(x)=0$

- ***Inconsistency happens since computer B fails before propagating new value to computer A***

Linearizability (Lamport)

- The interleaved sequence of operations
  - Assume client *i* performs operations: $o_{io}, o_{i1}, o_{i2}, \ldots$
  - Then a sequence of operations executed on one replica that issued by two clients may be: $o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, \ldots$
- **Linearizability criteria**
  - 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 times at which the operations occurred in the actual execution

Linearizability



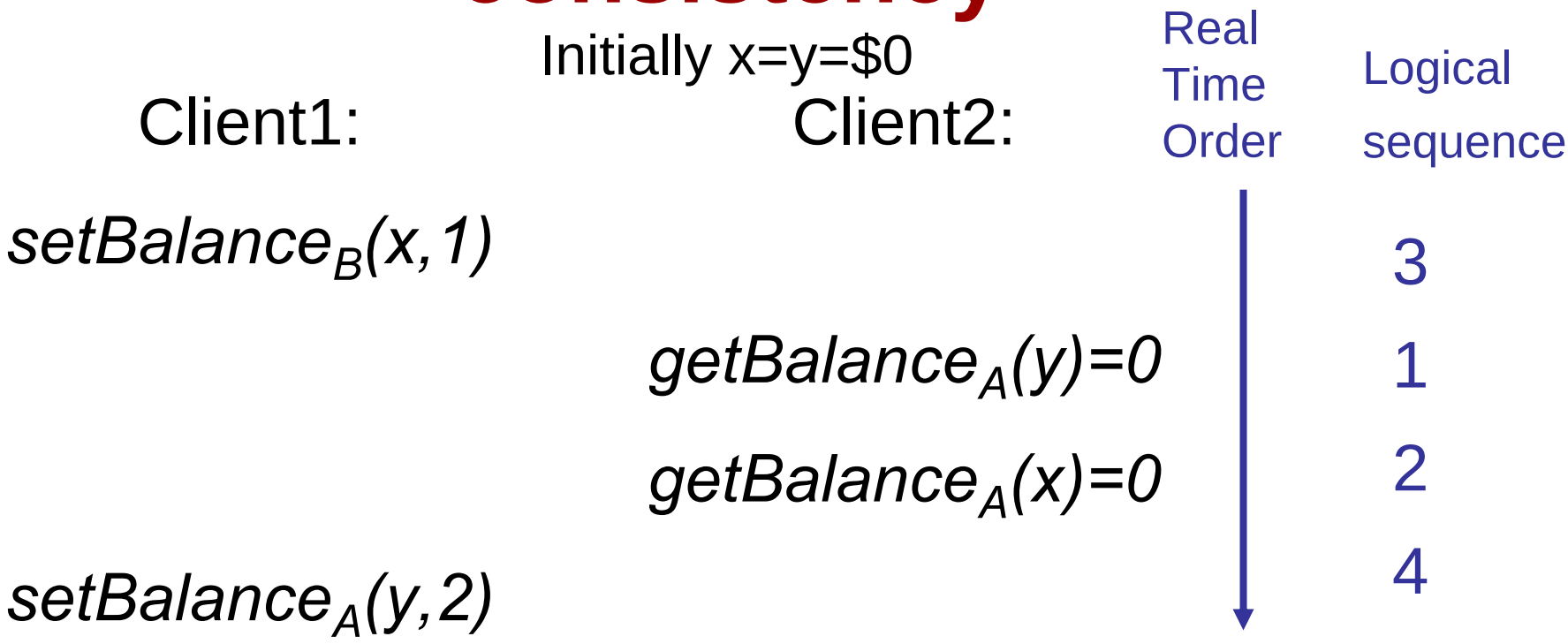
- Rule:
  - m_i must be delivered before m_j if $T_i < T_j$
- Implementation:
  - A clock synchronized among machines
  - A sliding time window used to commit message delivery whose timestamp is in this window.
- Drawback
  - Too strict constraint
  - No absolute synchronized clock
  - No guarantee to catch all tardy messages

Sequential consistency (Lamport)

- Sequential consistency criteria
  - 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 individual client executed them
    - *Client 1:* $o_{10}, o_{11}, \ldots$
    - *Client 2:* $o_{20}, o_{21}, o_{22}, \ldots$
    - *Consistent order* $o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, \ldots$

An example of sequential consistency

Initially x=y=$0



- An interleaving operations at server A:

  getBalance$_A$(y)=0;getBalance$_A$(x)=0;setBalance$_B$(x,1); setBalance$_A$(y,2)

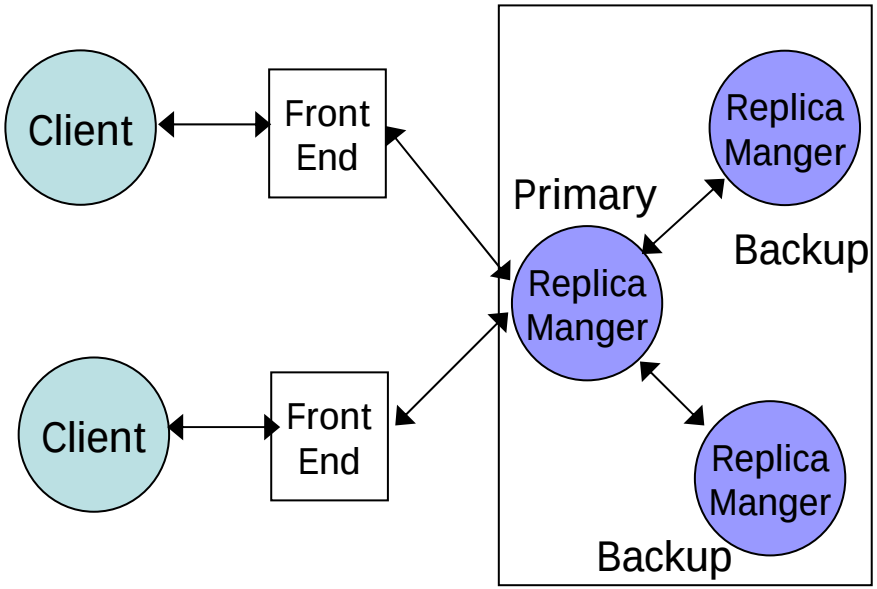
  – Does Not satisfy linearizability

  – Satisfy sequential consistency

Multi-copy Update Problem

- Read-only replication
  - Allow the replication of only immutable files.
- Primary backup replication
  - Designate one copy as the primary copy and all the others as secondary copies.
- Active backup replication
  - Access any or all of replicas
    - Read-any-write-all protocol
    - Available-copies protocol
    - Quorum-based consensus

Primary-Backup (Passive) Replication



1. **Request:** The front end sends a request to the primary replica.
2. **Coordination:**. The primary takes the request atomically.
3. **Execution:** The primary executes and stores the results.
4. **Agreement:** The primary sends the updates to all the backups and receives an ack from them.
5. **Response:** reply to the front end.

Advantage: an easy implementation, linearizable, coping with n-1 crashes.

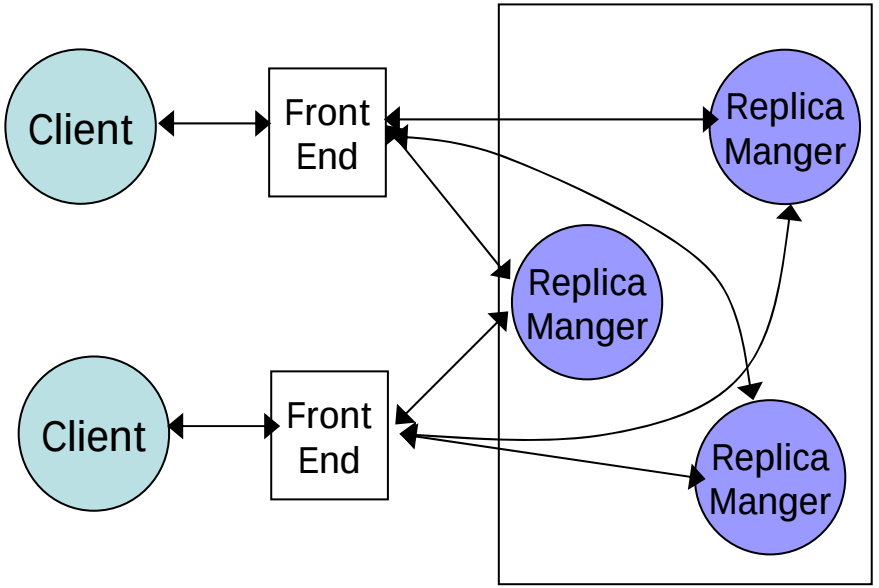
Disadvantage: large overhead especially if the failing primary must be replaced with a backup

Reading from backups => sequential consistency

Handover: agree on performed operations, and elect unique new primary!

(View-synchrounous group communication)

Active Replication

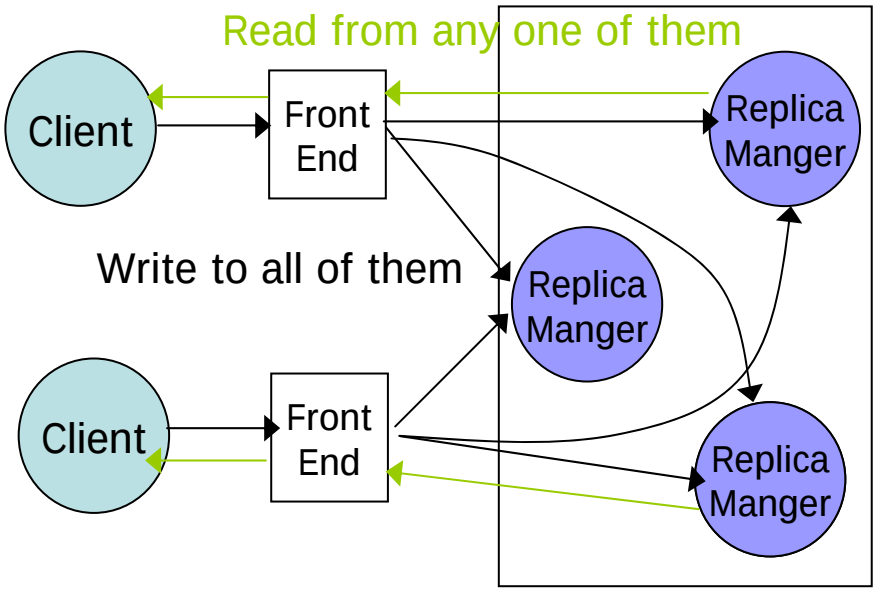


1. **Request:** The front end **RTO-multicasts** to all replicas.
2. **Coordination:**. All replica take the request in the sequential order.
3. **Execution:** Every replica executes the request.
4. **Agreement:** No agreement needed.
5. **Response:** Each replies to the front.

Advantage: achieve sequential consistency,
cope with (n/2 – 1) byzantine failures using majority + message signing

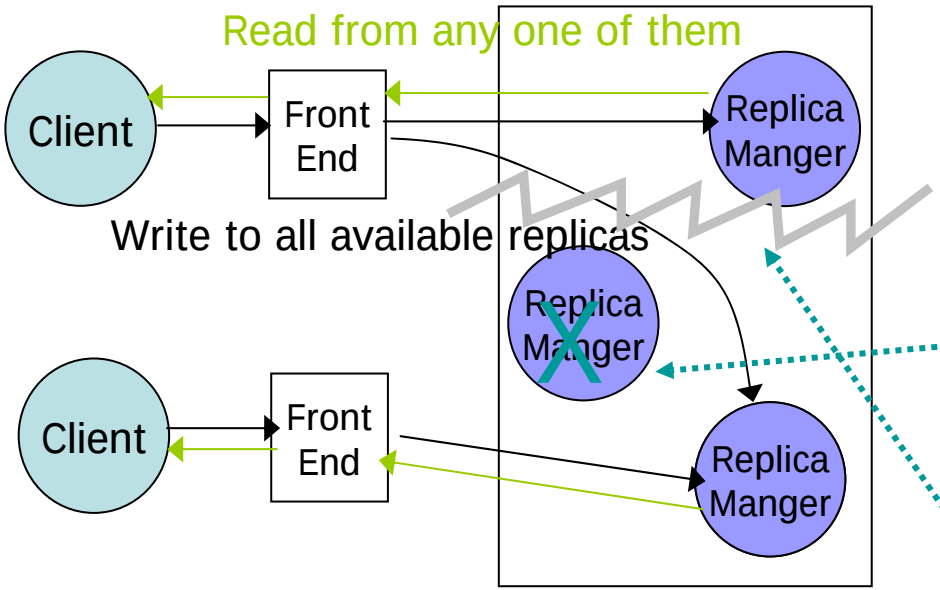
Disadvantage: no more linearizable, RMs are state machines

Read-Any-Write-All Protocol



- Read
  - Perform read at any one of the replicas
- Write
  - Perform on *all* of the replicas
- Sequential consistency
- Cannot cope with even a single crash (by definition)

Available-Copies Protocol

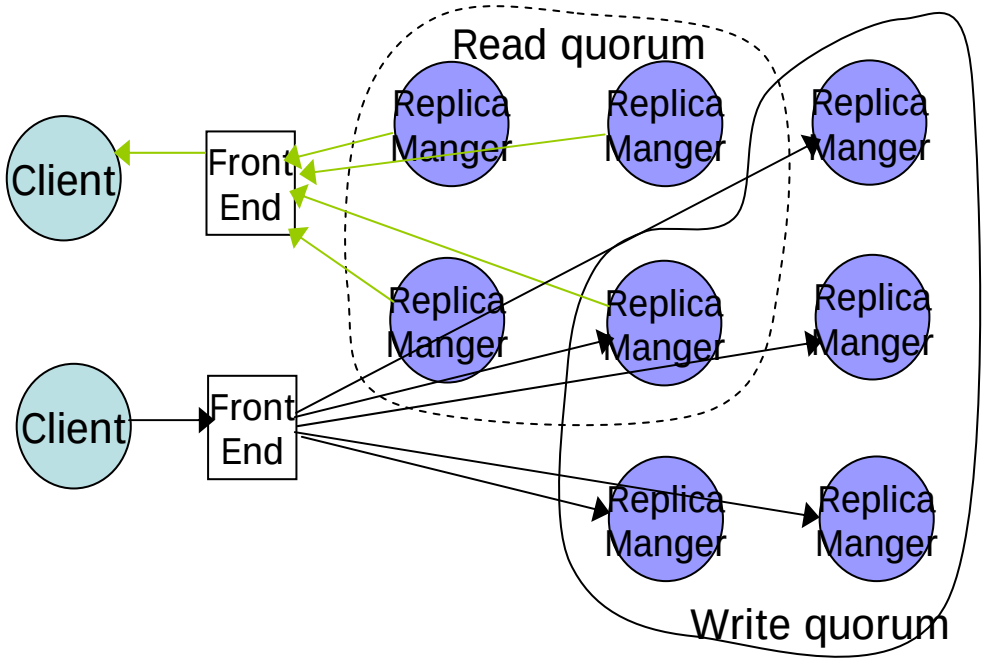


- Read
  - Perform on any one of the replicas
- Write
  - Perform on *all available* replicas
- Recovering replica
  - Bring itself up to date by coping from other servers before accepting any user request.
- Better availability
- Cannot cope with network partition. (Inconsistency in two sub-divided network groups)

Quorum-Based Protocols

Quorum Constriants

1. Intersecting R/W #replicas in read quorum + #replicas in write quorum > n
2. Write majority: #replicas in write quorum > n/2



Read-any-write-all: r = 1, w = n

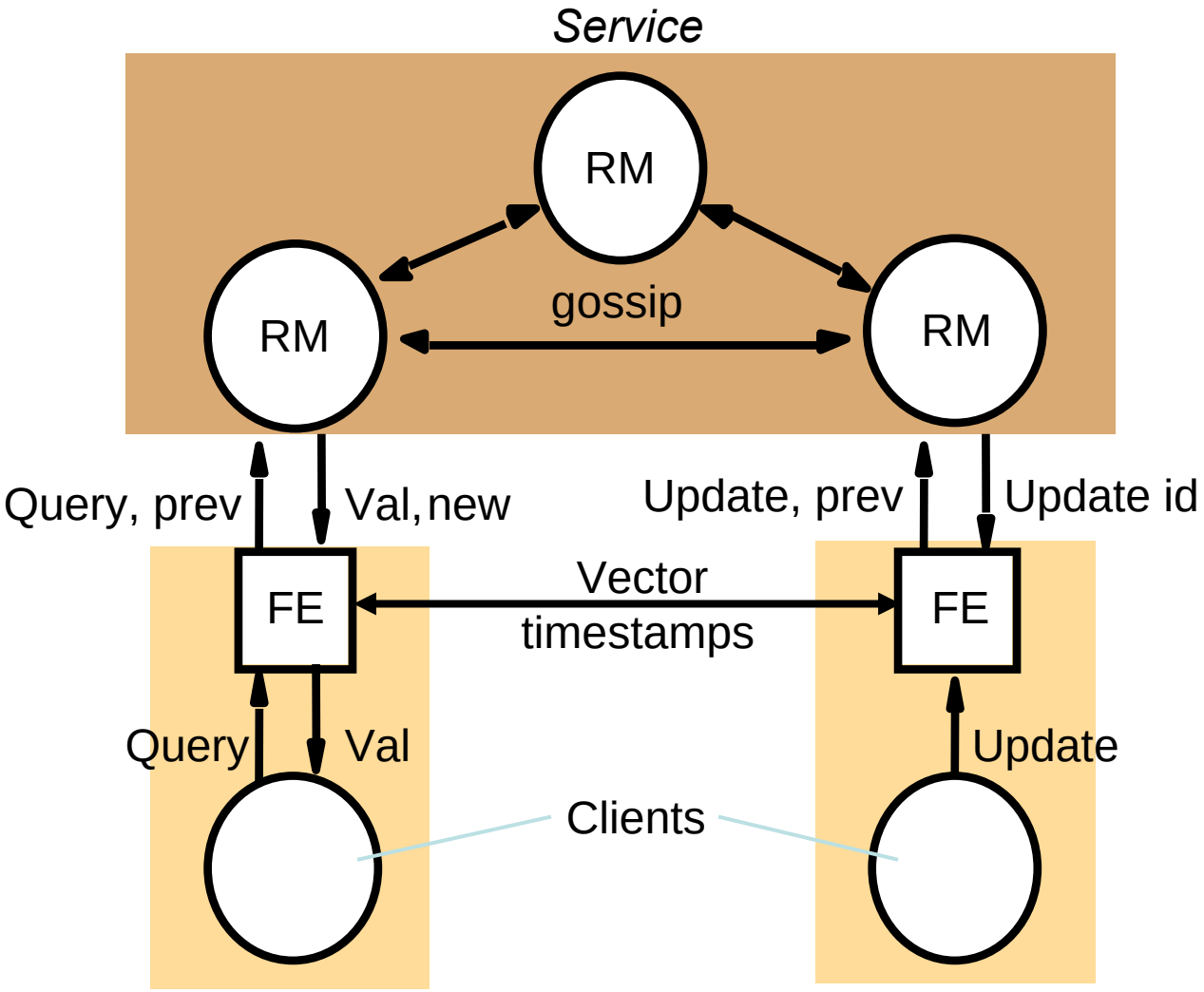
- Read
  - Retrieve the read quorum
  - Select the one with the latest version.
  - Perform a read on it
- Write
  - Retrieve the write quorum.
  - Find the latest version and increment it.
  - Perform a write on the entire write quorum.
- If a sufficient number of replicas from read/write quorum fails, the operation must be aborted.

High Availability

High availability vs. fault tolerance

- Fault tolerance
  - Strict (sequential) consistency
    - all replicas reach agreement before passing control to client
- **High availability**
  - Obtain access to a service for as much time as possible
  - Reasonable Response time
  - Relaxed consistency (lazy update)
    - Reach consistency until next access
    - Reach agreement after passing control to client
  - Eg: Gossip, Bayou, Coda

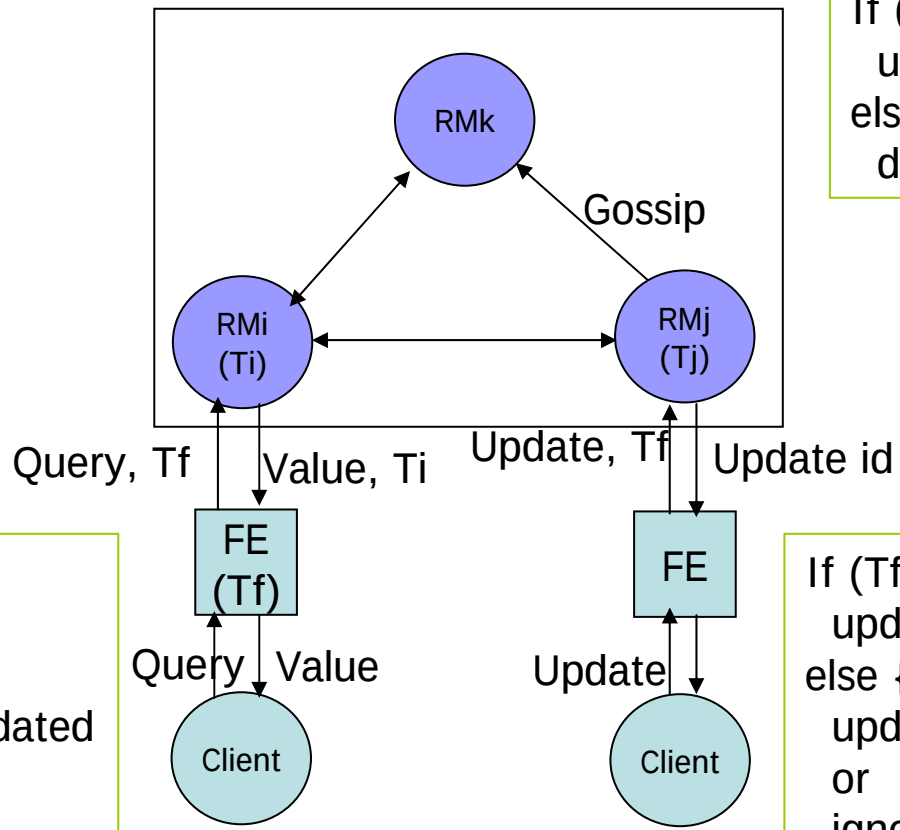
Operations in a gossip service



Basic Gossip Operation

Perform operations in causal order

Ti are vector time-stamps



If (Tj > Tk)
  update RMk
else
  discard the gossip message

If (Tf < Ti)
  return value
else {
  waits for RMi to be updated
  or
  query RMj/RMk}

If (Tf > Tj)
  update RMj
else {
  update Client
  or
  ignore and update RMj}

Phases in Gossip

- Request
  - The *front end* sends the request to a *replica manager*
    - Query: client may be blocked
    - Update: unblocked
- Coordination
  - Suspend the request until it can be apply
    - May receive gossip messages that sent from other replica managers
- Execution
  - The replica manager executes the request
- Agreement
  - exchange gossip messages which contain the most recent updates applied on the replica
    - Exchange occasionally
    - Ask the particular replica manager to send when some replica manager finds it has missed one
- Response
  - Query: Reply after coordination
  - Update: Replica manager replies immediately

(Recall) Vector Clocks

- Lamport: e → f implies C(e) < C(f)
- Vector clocks: e → f **iff** C(e) < C(f)
- **vector timestamps**: Each node maintains an array of N counters
- $\mathbf{V}_i[i]$ is the local clock for process p_i
- In general, $\mathbf{V}_i[j]$ is the latest info the node has on what p_j's local clock is.

Implementation Rules

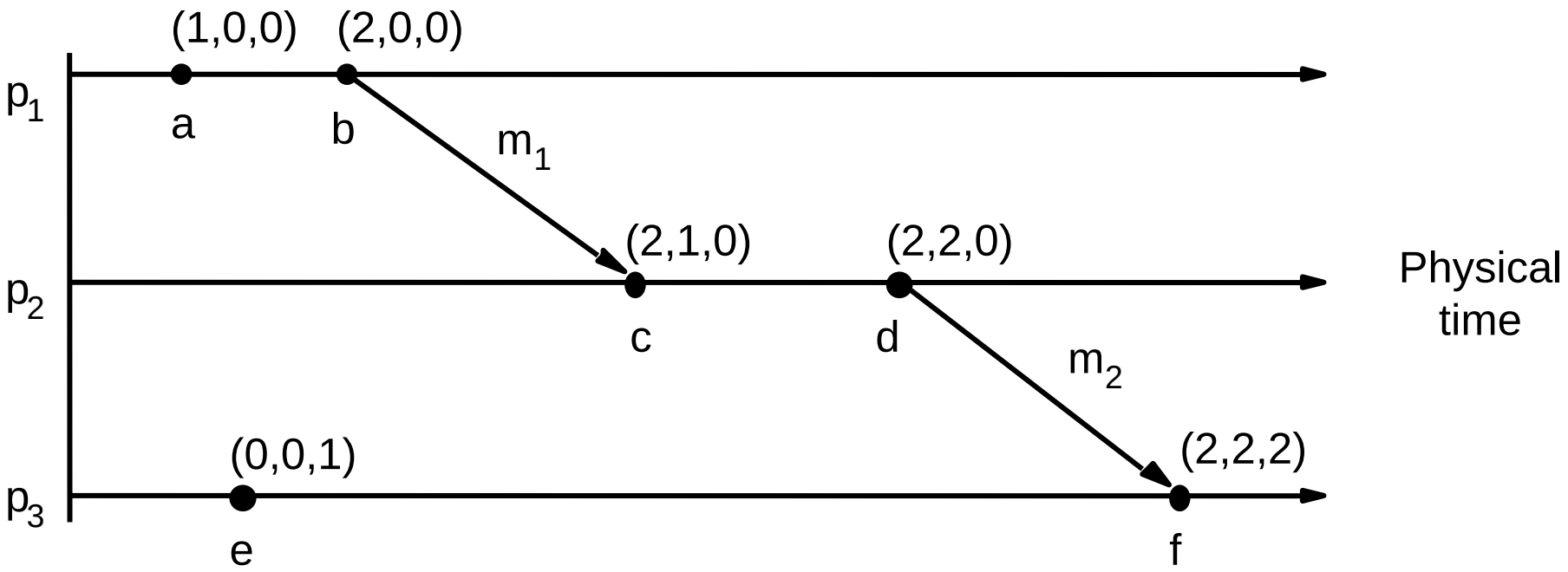
- **[VC1]** Initially $\mathbf{V}_i[j]=0$ for i,j = 1…N
- **[VC2]** Before P_i timestamps an event:
  $\mathbf{V}_i[i] := \mathbf{V}_i[i] + 1$
- **[VC3]** P_i sends m: piggy-back timestamp $\mathbf{t}=\mathbf{V}_i$:
  m'=<m, **t**>
- **[VC4](Merge)** P_j receives m'=<m, **t**>
  $\mathbf{V}_i[i] := \max(\mathbf{V}_i[i], \mathbf{t}_i[i])$

Comparison of Vector Clocks

Comparing vector clocks

- $V = V'$ iff $V[j] = V'[j]$ for all $j=1,2,\ldots,N$.
- $V \leq V'$ iff $V[j] \leq V'[j]$ for all $j=1,2,\ldots,N$.
- $V < V'$ iff $V \leq V'$ and $V \neq V'$.

Vector clocks illustrated

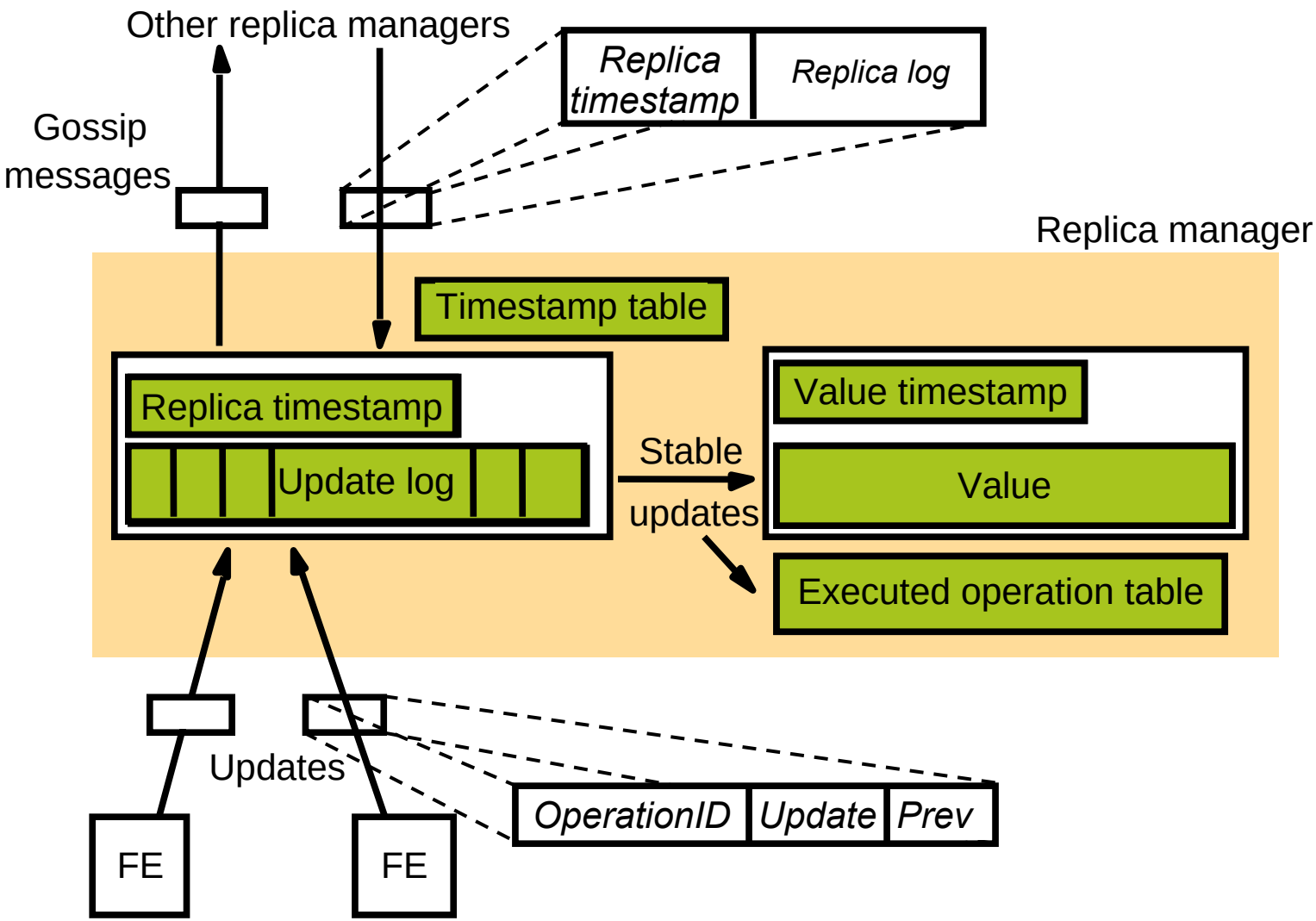


NOTE e and b are not related

The front end's version timestamp

- **Client Communication**
  - Access the gossip service
    - Update any set of RMs
    - Read from any RM
  - Communicate with other clients directly
- **Causal Updates**
  - A vector timestamp at each front end contains an entry for each replica manager
  - Attached to every message sent to the gossip service or other *front ends*
  - When *front end* receives a message
    - Merge the local vector timestamp with the timestamp in the message
- **Front end Vector timestamp:**
  - Reflect the version of the latest data values accessed by the front end

Gossip Manager State



Replica Manager State

- **Value**
- **Value timestamp**
  - Represent the updates that are reflected in the value
  - E.g., (2,3,5): the replica has received 2 updates from 1st FE, 3 updates from 2nd FE, and 5 updates from 3rd FE
- **Update log**
  - Record all received updates; *stable* update; gossip propagated
- **Replica timestamp**
  - Represents the updates that have been accepted by the replica manager
- **Executed operation table**
  - Filter duplicated updates that could be received from *front end* and other *replica managers*
- **Timestamp table**
  - Contain a vector timestamp for each other replica manager to identify what updates have been applied at these replica managers

Queries

- When the query q reach the RM
  - **If *q.prev <= valueTS***
    - Return immediately
    - The timestamp in the returned message is *valueTS*
  - Otherwise
    - Pend the query in a hold-back queue until the condition is satisfied
    - E.g. *valueTS* = (2,5,5), *q.prev*=(2,4,6): one update from *replica manager* 2 is missing
- When query return
  - *frontEndTS:= merge(frontEndTS, valueTS)*

Causal Update 1

- **A *front end* sends the update as**
  - ***<u.op(par-list), u.prev, u.id>***
    - *u.prev*: the timestamp of the *front end*
- **When *replica manager i* receives the update**
  - Discard
    - If the update has been in the *executed operation table* or is in the *log*
  - Otherwise, save it in the log
    - Replica timestamp[i]++
    - ***logRecord= <i, ts, u.op, u.prev, u.id>***
      - Where *ts* =*u.prev*, *ts[i]=replica timestamp[i]*
  - Pass *ts* back to the front end
    - *frontEndTS=merge(frontEndTS, ts)*

Causal Update 2

- Check if the update becomes stable
  - ***u.prev <= valueTS***
  - Example: a stable update at RM 0
    - *ts*=(3,3,4), *u.prev*=(2,3,4), *valueTS*=(2,4,6)
- Apply the stable update
  - *Value* = apply(*value*, *r.u.op*)
  - *valueTS* = merge(*valueTS*, *r.ts*) (3,4,6)
  - *executed* = *executed* $\cup${*r.u.id*}

Sending Gossip

- Exchange gossip message
  - Estimate the missed messages of one replica manager by its timestamp table
  - Exchange gossip messages periodically or when some other replica manager ask
- The format or a gossip message
  - ***<m.log,m.ts>***
  - *m.log*: one or more updates in the source *replica manager*'s log
  - *m.ts*: the *replica timestamp* of the source *replica manager*

Receiving Gossip 1

1. Check the record *r* in *m.log*

- Discard if *r.ts* <= *replicaTS*
  - The record *r* has been already in the local log or has been applied to the value
- Otherwise, insert *r* in the local log
  - *replicaTS* = merge (*replicaTS*, *m.ts*)

2. Find out the stable updates

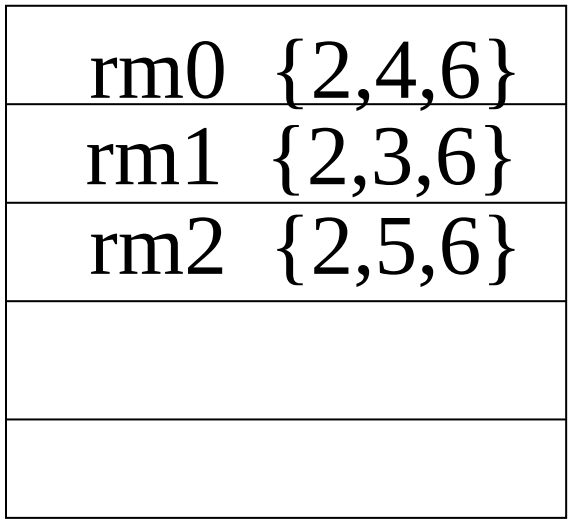
- Sort the updates log to find out stable ones, and apply to the value according to the "≤" (thus happens-before) order

3. Update the timestamp table

- If the gossip message is from replica manager *j*, then *tableTS[j]=m.ts*

Receiving Gossip 2

- Discard useless (have been received everywhere) update *r* in the log
  - if *tableTS[i][c]* >= *r.ts[c]*, then discard *r*
    - *c* is the *replica manager* that created *r*
    - For all *i*



logRecord {i,ts,u.op,u.prev,u.id}

Gossiping

- How often to exchange gossip messages?
  - Minutes, hours or days
    - Depend on the requirement of application
- How to choose partners to exchange?
  - Random
  - Deterministic
    - Utilize a simple function of the replica manager's state to make the choice of partner
  - Topological
    - Mesh, circle, tree

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

Optimistic approaches

- Provides a high availability by relaxing the consistency guarantees
- When conflicts are rare
- Detect conflicts
  - Relies domain specific conflict detection and resolution
  - Inform user
- Eg
  - Bayou data replication service
  - CODE file system

END