

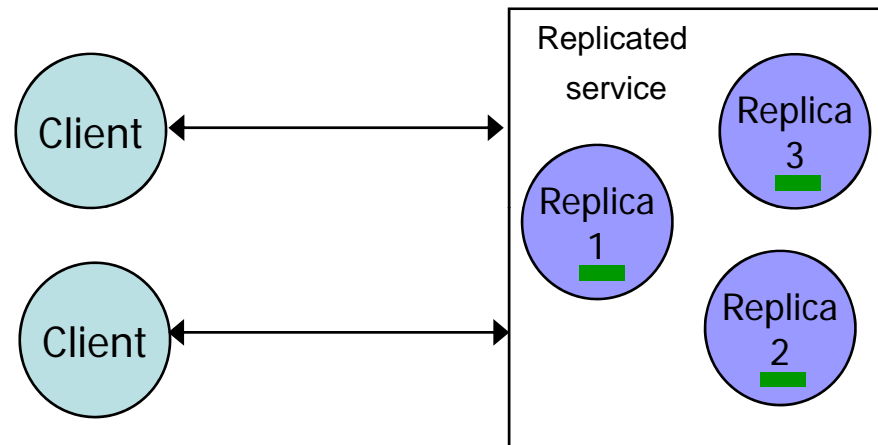
Replication

Brian Nielsen

`bnielsen@cs.aau.dk`

Service Improvements

- Replication is a key technology to enhance service



- Performance enhancement
- Fault tolerance
- Availability

Service Improvements

- Performance enhancement
 - Load-balance
 - Proximity-based response
 - Read queries can be executed in parallel
 - Example
 - caches in DNS servers / file servers (NFS/AFS)
 - replicated web servers
 - Load pattern may determine *if* performance improves
 - E.g R/W ratio
 - Beware updates may be costly (consistency)

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

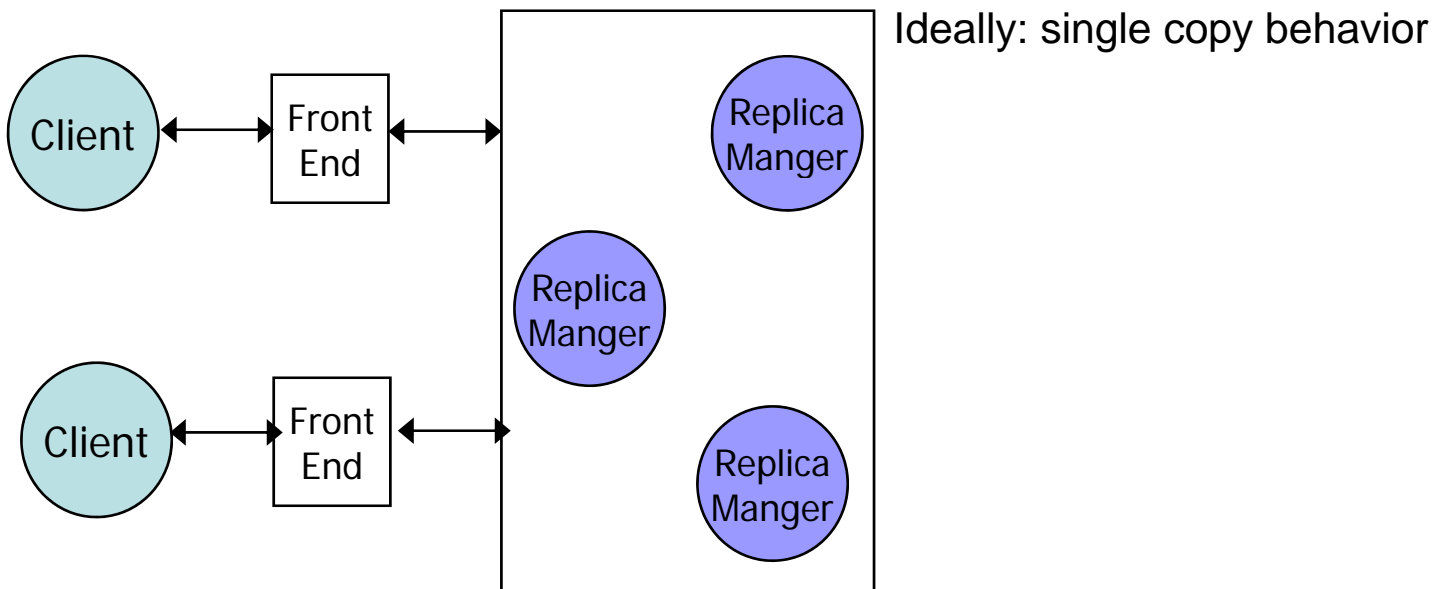
- Fault tolerance

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

Basic Architectural Model

- Requirements

- Transparency: clients need not be aware of multiple replicas.
- Consistency: data consistency among replicated files.

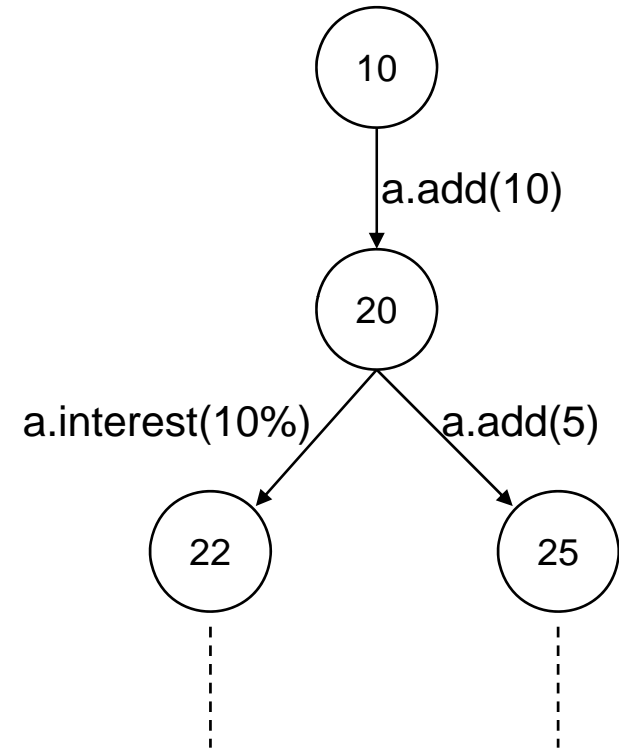


Replication

- **Difference between replication and caching**
 - A replica is associated with a server, whereas a cache with client.
 - A replicate focuses on availability, while a cache on locality
 - A replicate is more persistent than a cache is
 - A cache is contingent upon a replica
- **Advantages**
 - Increased availability/reliability
 - Performance enhancement (response time and network traffic)
 - Scalability and autonomous operation
- **Requirements**
 - Transparency: no need to be aware of multiple replicas.
 - Consistency: data consistency among replicated files.

Operations

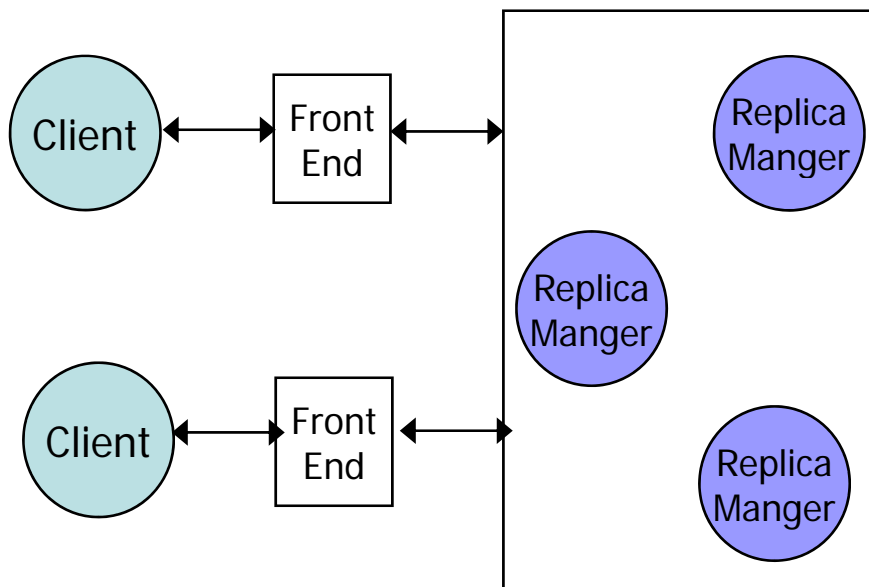
- Client performs operations on a replicated object **obj.m(...)**
 - Executed atomically
- “**state machine objects**”: state depends only on initial state and sequence of operations (deterministic function)
 - Precludes that operations depend on external inputs such as system clock and sensor values
- Updates vs. queries (read-only)
- Single operations vs. sequence (transactions (15.5))



Basic Architectural Model

•Requirements

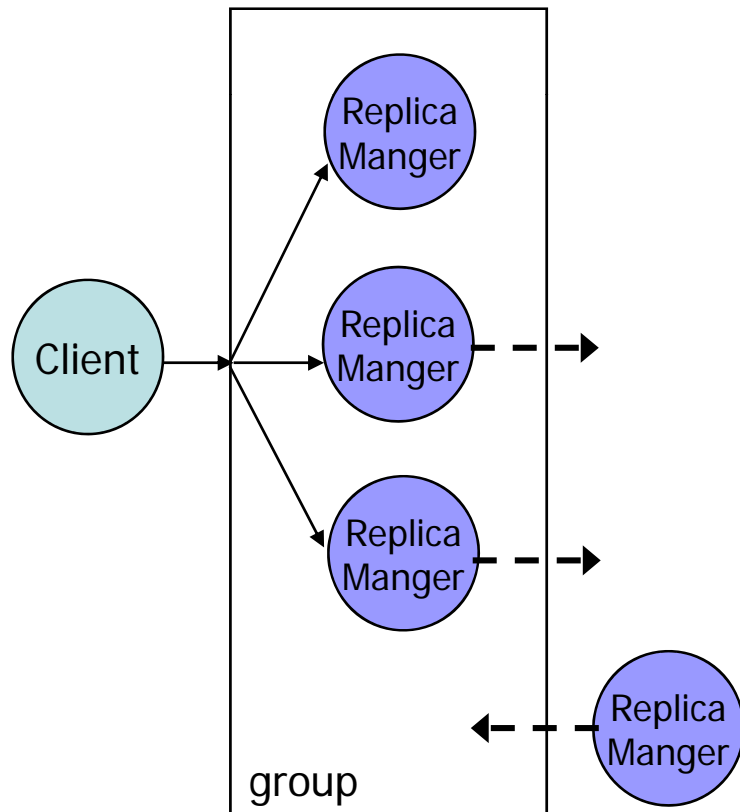
- Transparency: no need to be aware of multiple replicas.
- Consistency: data consistency among replicated files.



General Phases in an Replication alg.:

1. **Request:** client sends request to a manager (via front-end).
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 client (via front end)

Group Communication



- Group membership service
 - Create and destroy a group.
 - Add or withdraw a replica manager to/from a group.
 - Detect a failure.
 - Notify members of group membership changes.
 - Provide clients with a group address.
- Message delivery
 - Absolute ordering
 - Consistent ordering

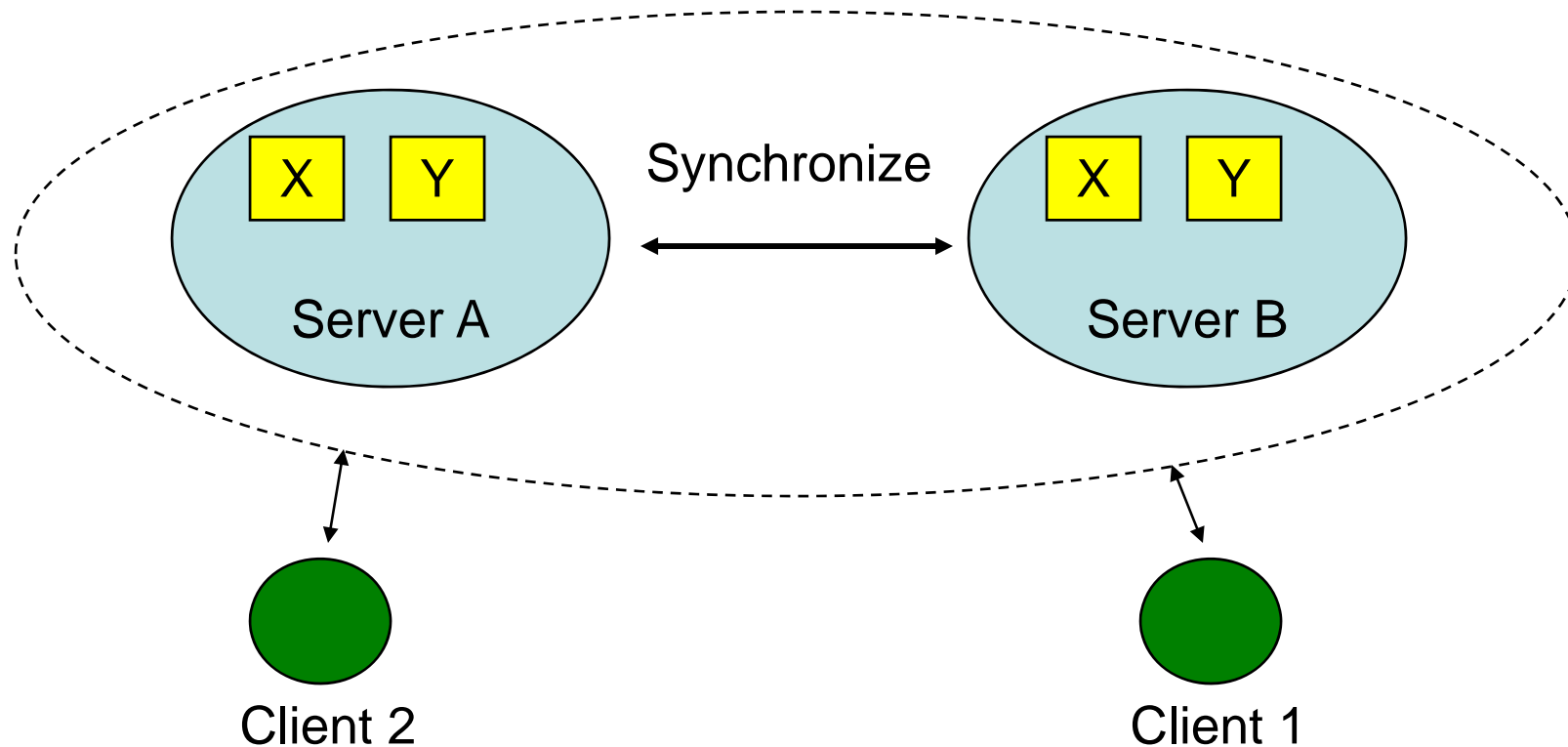
Fault Tolerance

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 from 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

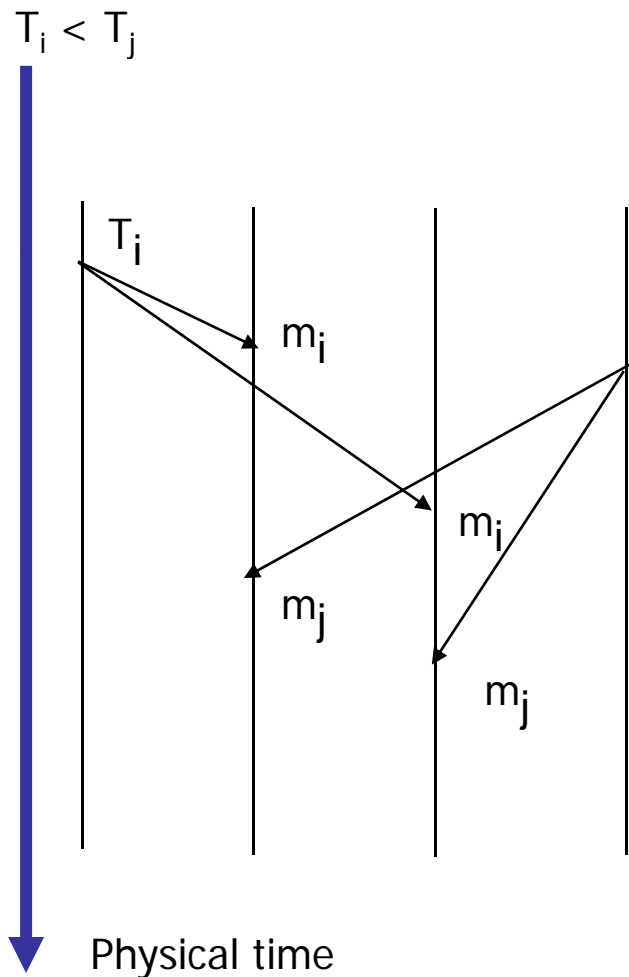
time

- ***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: $O_{i0}, O_{i1}, O_{i2}, \dots$
 - Then a sequence of operations executed on one replica that issued by two clients may be: $O_{20}, O_{21}, O_{10}, O_{22}, O_{11}, \dots$
- **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

Linearizability ... *continued*

- Example of “a single correct copy of the objects”
 - A correct bank account
 - For **auditing** purposes, if one account update occurred after another, then the first update should be observed if the second has been observed
- Linearizability is not for transactions
 - Concern only the interleaving of individual operations
- The most strict consistency between replicas
 - Linearizability is hard to achieve


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}, \dots*
 - *Client 2: $o_{20}, o_{21}, o_{22}, \dots$*
 - *Consistent order $o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, \dots$*

An example of sequential consistency

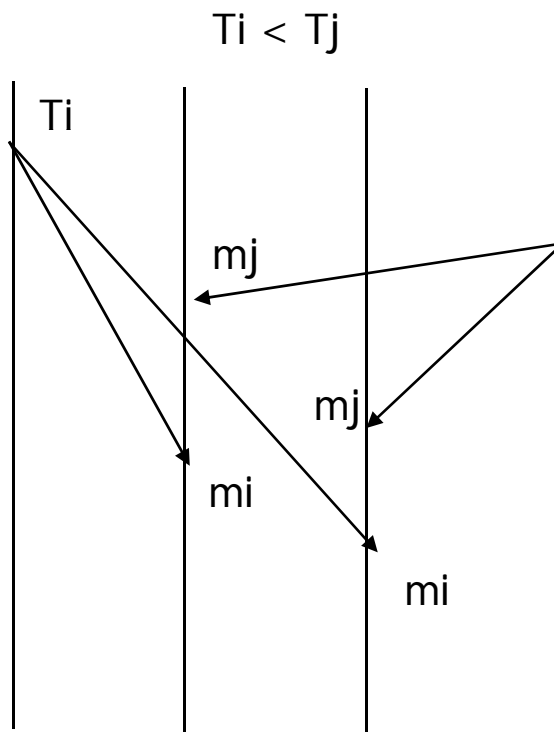
Initially $x=y=\$0$

Client1:	Client2:	Real Time Order	Logical sequence
$setBalance_B(x, 1)$			3
	$getBalance_A(y)=0$		1
	$getBalance_A(x)=0$		2
$setBalance_A(y, 2)$			4



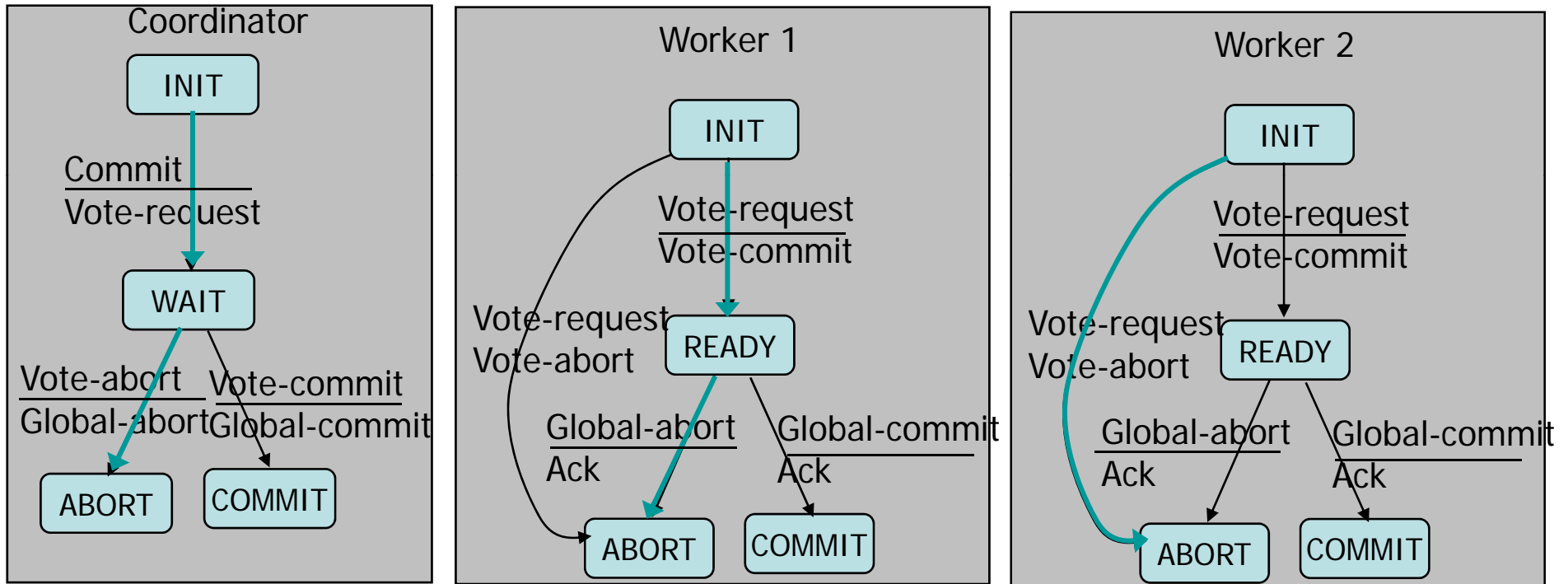
- An interleaving operations at server A:
 $getBalance_A(y)=0; getBalance_A(x)=0; setBalance_B(x, 1); setBalance_A(y, 2)$
 - Does Not satisfy linearizability
 - Satisfy sequential consistency

Sequential Consistency



- Rule:
 - Messages received in the same order (regardless of their timestamp).
- Implementation:
 - T_j – A message sent to a sequencer, assigned a sequence number, and finally multicast to receivers
 - A message retrieved in incremental order at a receiver
- Drawback:
 - Still strong ordering requirement
 - A centralized algorithm

Two-Phase Commit Protocol



Another possible cases:

The coordinator didn't receive all vote-commits.

A worker didn't receive a vote-request.

A worker didn't receive a global-commit.

→ Time out and send a global-abort.

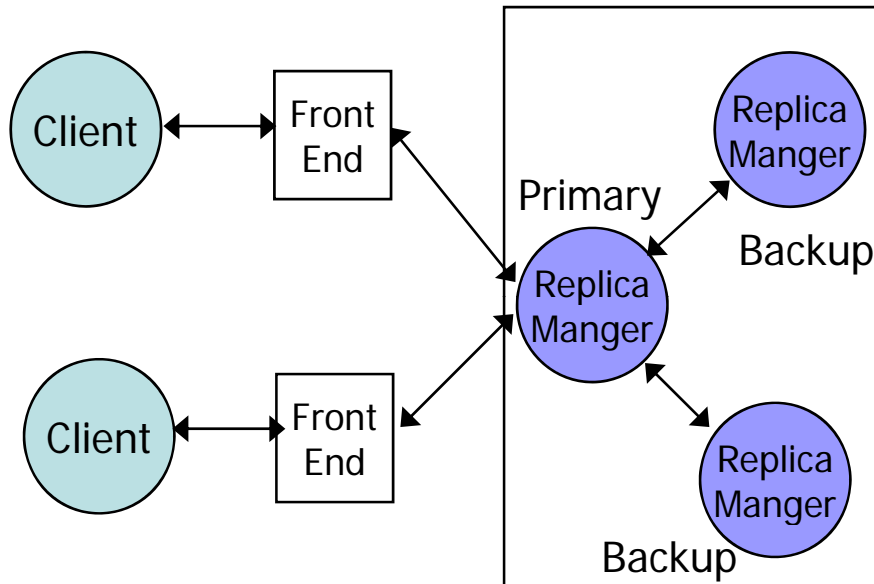
→ All workers eventually receive a global-abort.

→ Time out and check the other work's status.

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.

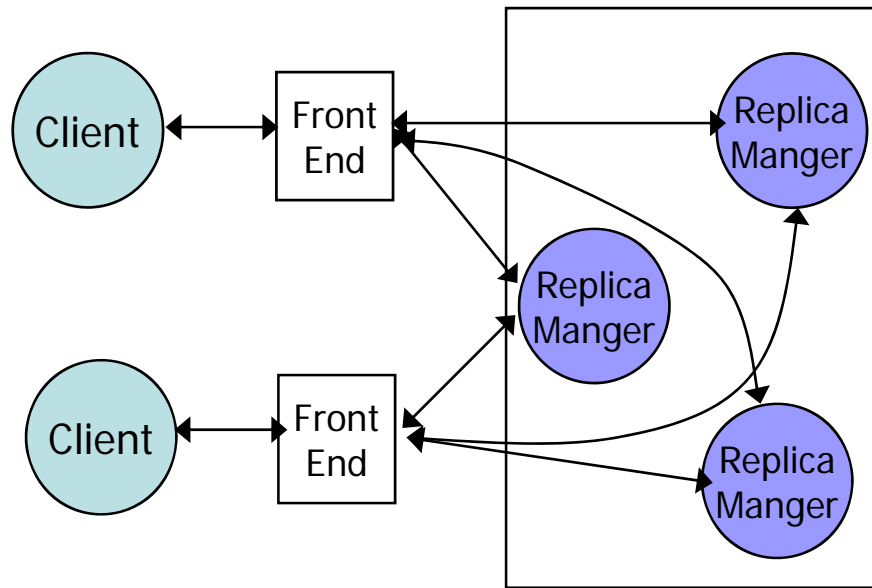
Advantage: an easy implementation, linearizable, coping with n-1 crashes.

Disadvantage: large overhead/delay when primary

Allowing reading from backups => sequential consistency

Handover: 1) detect failure, 2) agree on performed operations, 3) and elect unique new primary 4) inform clients to switch!

Active Replication



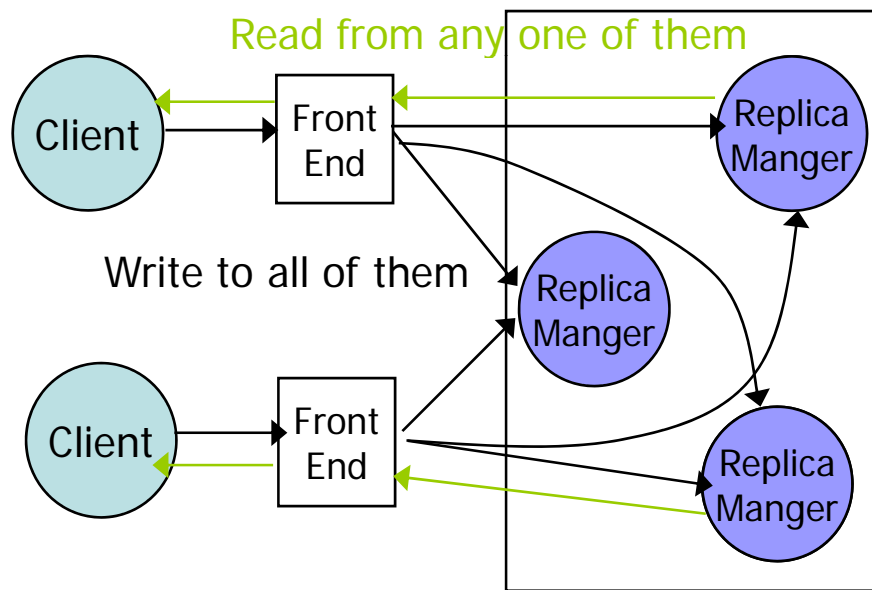
1. **Request:** The front end **RTO-multicasts** to all replicas.
2. **Coordination:** All replica take the request in the sequential order.
3. **Execution:** Every replica executes the request.
4. **Agreement:** No agreement needed.
5. **Response:** Each replies to the front.

Advantage:

- achieve sequential consistency,
- cope with $(n/2 - 1)$ byzantine failures using majority + message signing
- Hot-standby

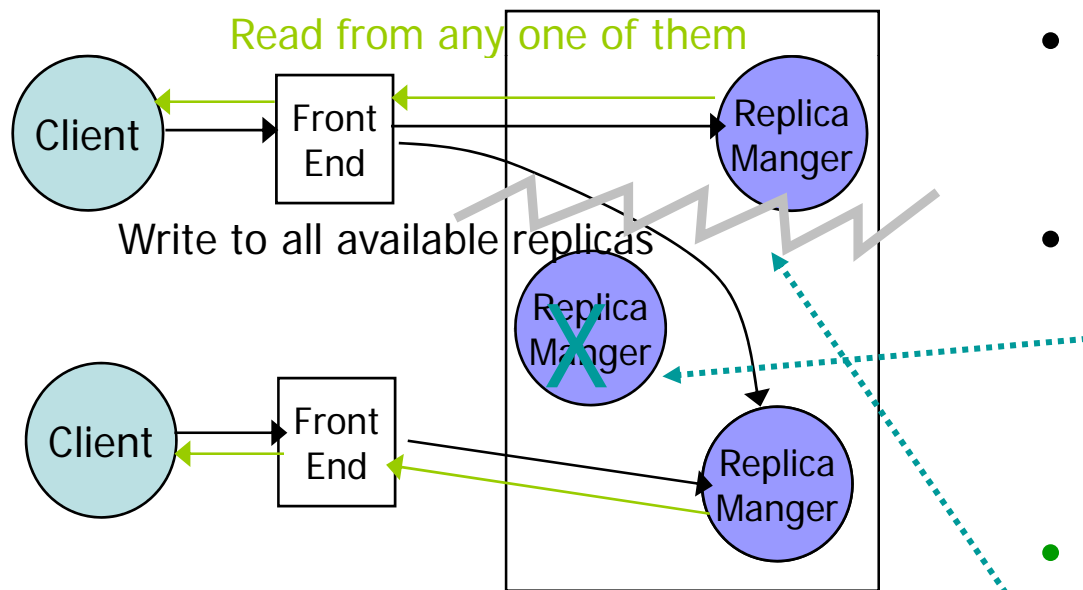
Disadvantage: no more linearizable, RMs are state machines

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

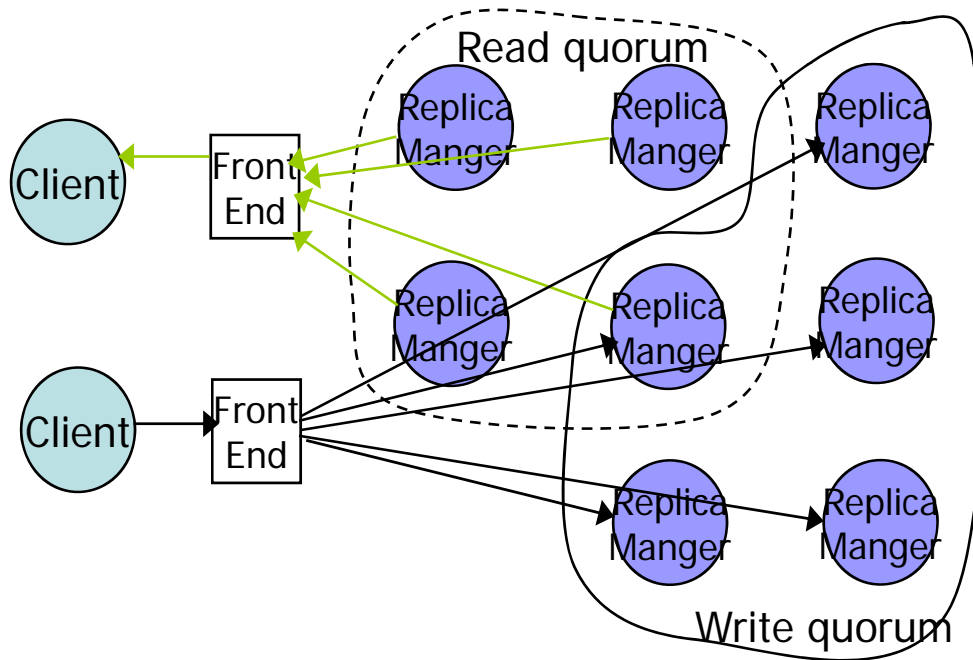


- Read
 - Perform on any one of the replicas
- Write
 - Perform on *all available* replicas
- Recovering replica
 - Bring itself up to date by coping from other servers before accepting any user request.
- Better availability
- Cannot cope with network partition. (Inconsistency in two sub-divided network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

Quorum-Based Protocols

Quorum Constraints

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

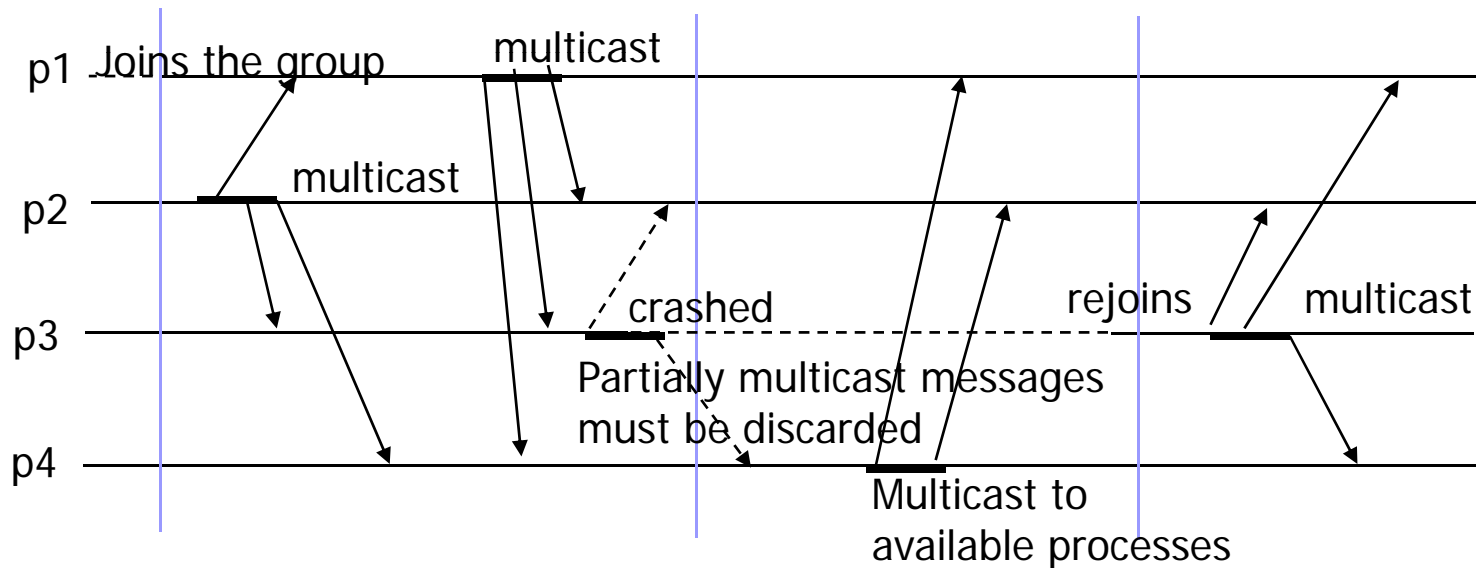


Read-any-write-all: $r = 1, w = n$

- 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.

ISIS System

- Process group: see page 4 of this ppt file
- Group view



- Reliable multicast
 - Causal multicast: see pages 5 & 6 of MPI ppt file
 - Atomic broadcast: see page 7 of this ppt file

High Availability

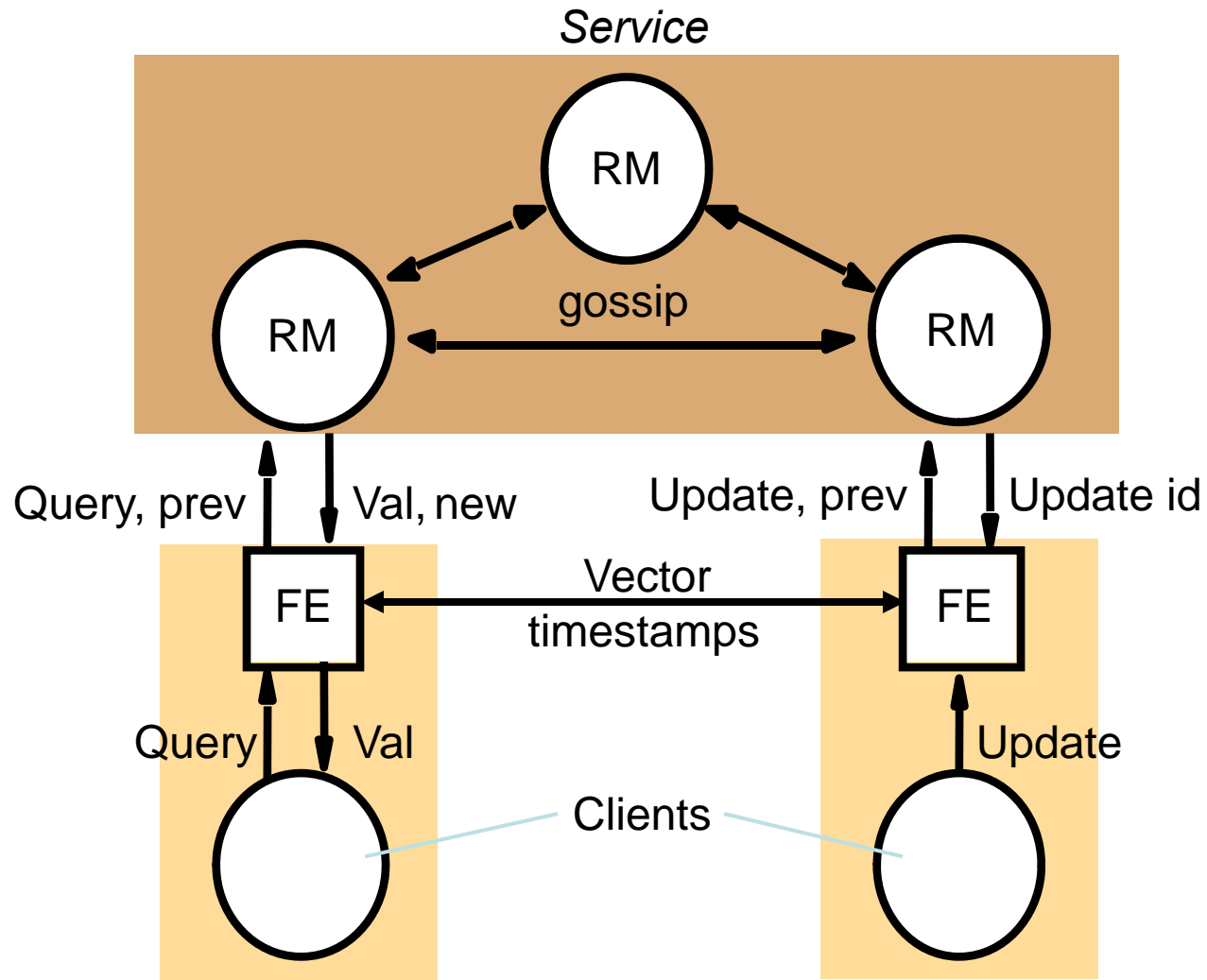
High availability vs. fault tolerance

- 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

High Availability Services

- Obtain access to a service for as much time as possible
- Provide reasonable response times
- Possibly relax the consistency requirements to replicas

Operations in a gossip service



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

Implementation Rules

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

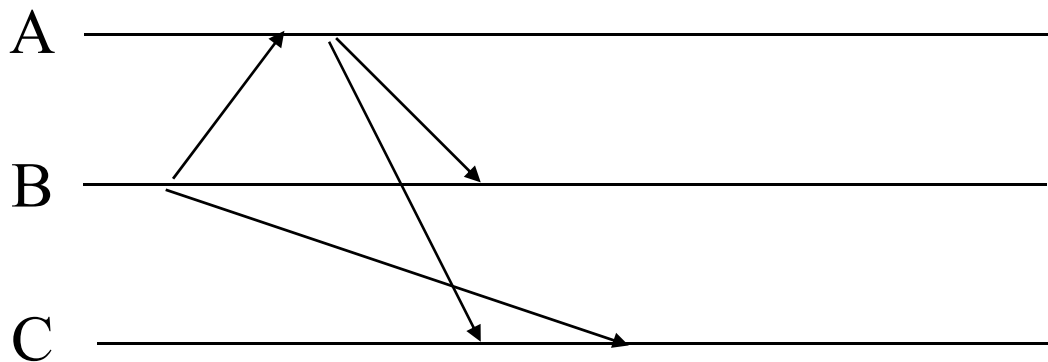
Comparison of Vector Clocks

Comparing vector clocks

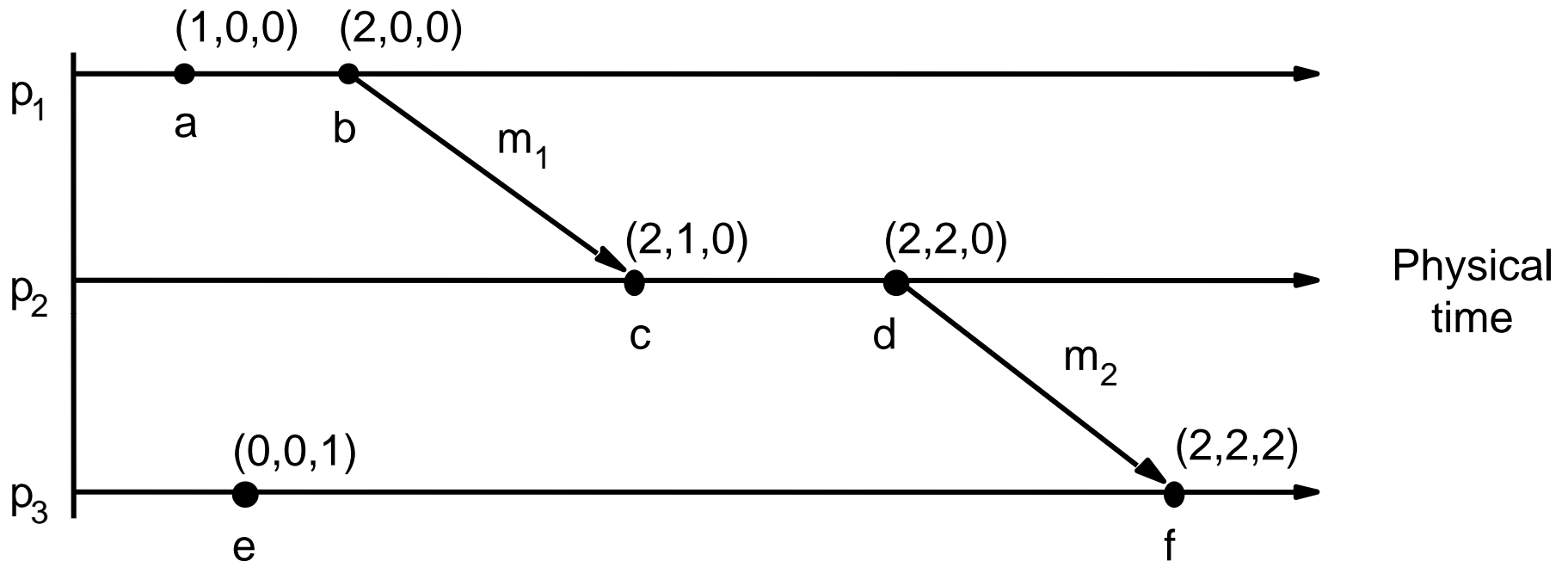
- $V = V'$ iff $V[j] = V'[j]$ for all $j=1,2,\dots,N$.
- $V \leq V'$ iff $V[j] \leq V'[j]$ for all $j=1,2,\dots,N$.
- $V < V'$ iff $V \leq V'$ and $V \neq V'$.

Vector Timestamps and Causal Violations

- C receives message (2,1,0) then (0,1,0)
- The later message causally precedes the first message if we define how to compare timestamps right



(Recall) Vector Clocks



- Vector clocks: $e \rightarrow f$ **iff** $C(e) < C(f)$
- $V_i[i]$ is the local clock for process p_i
- In general, $V_i[j]$ is the latest info the node has on what p_j 's local clock is.

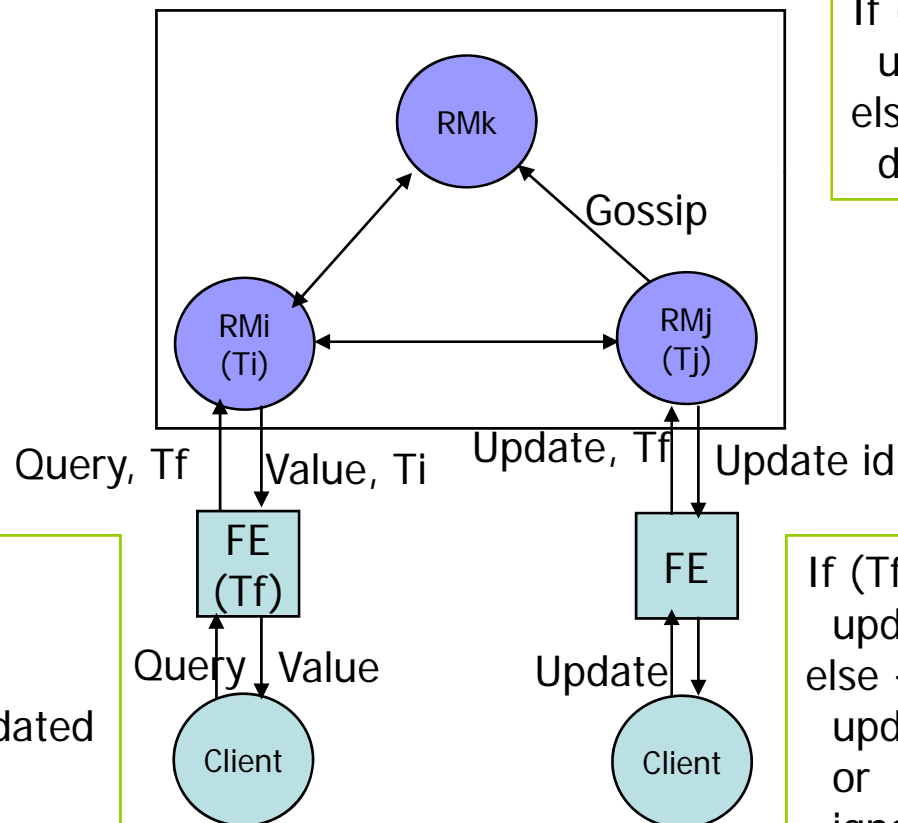
The front end's version timestamp

- **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

Basic Gossip Operation

Perform operations in causal order

T_i are vector time-stamps

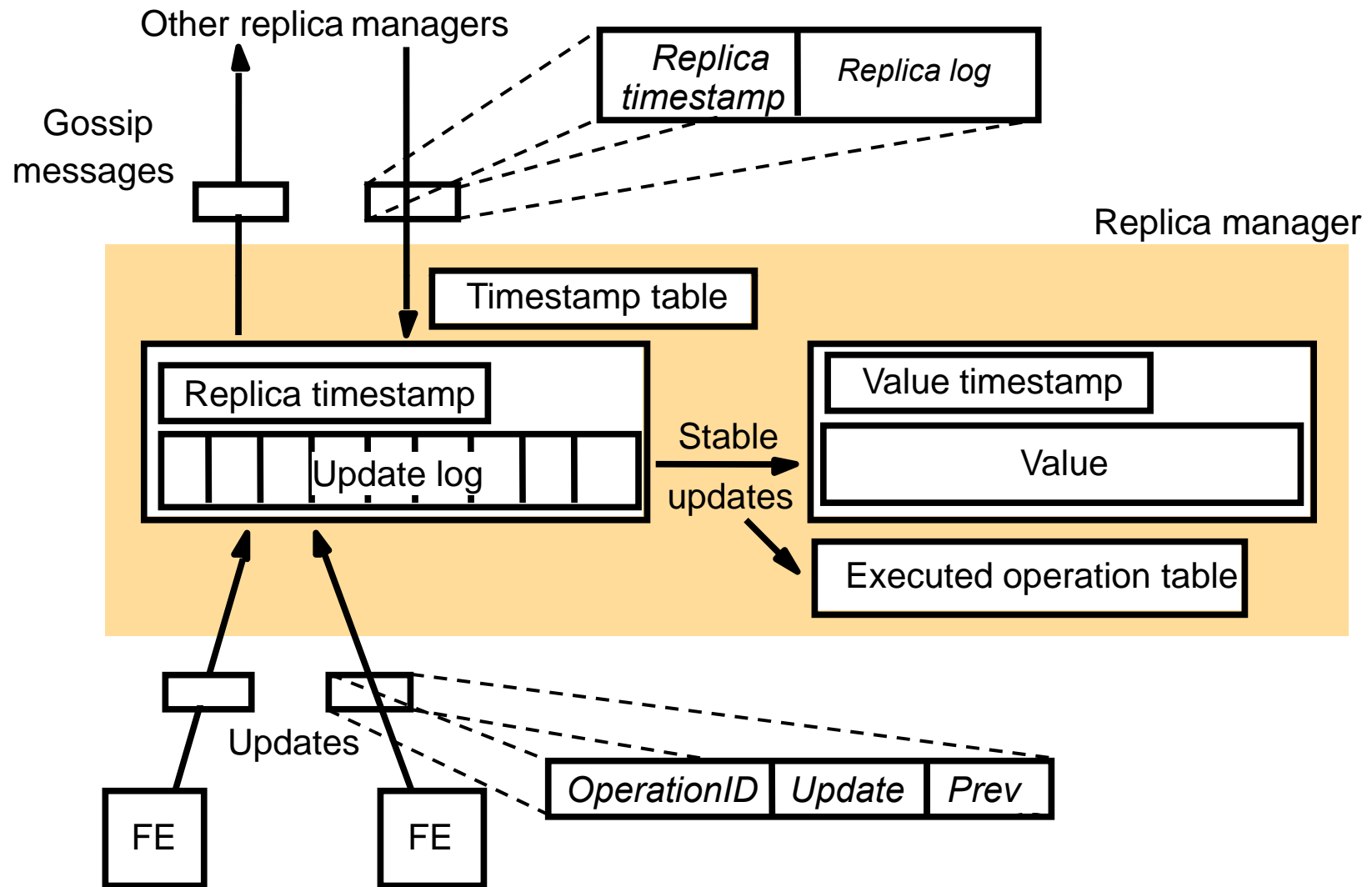


If ($T_j > T_k$)
update RMk
else
discard the gossip message

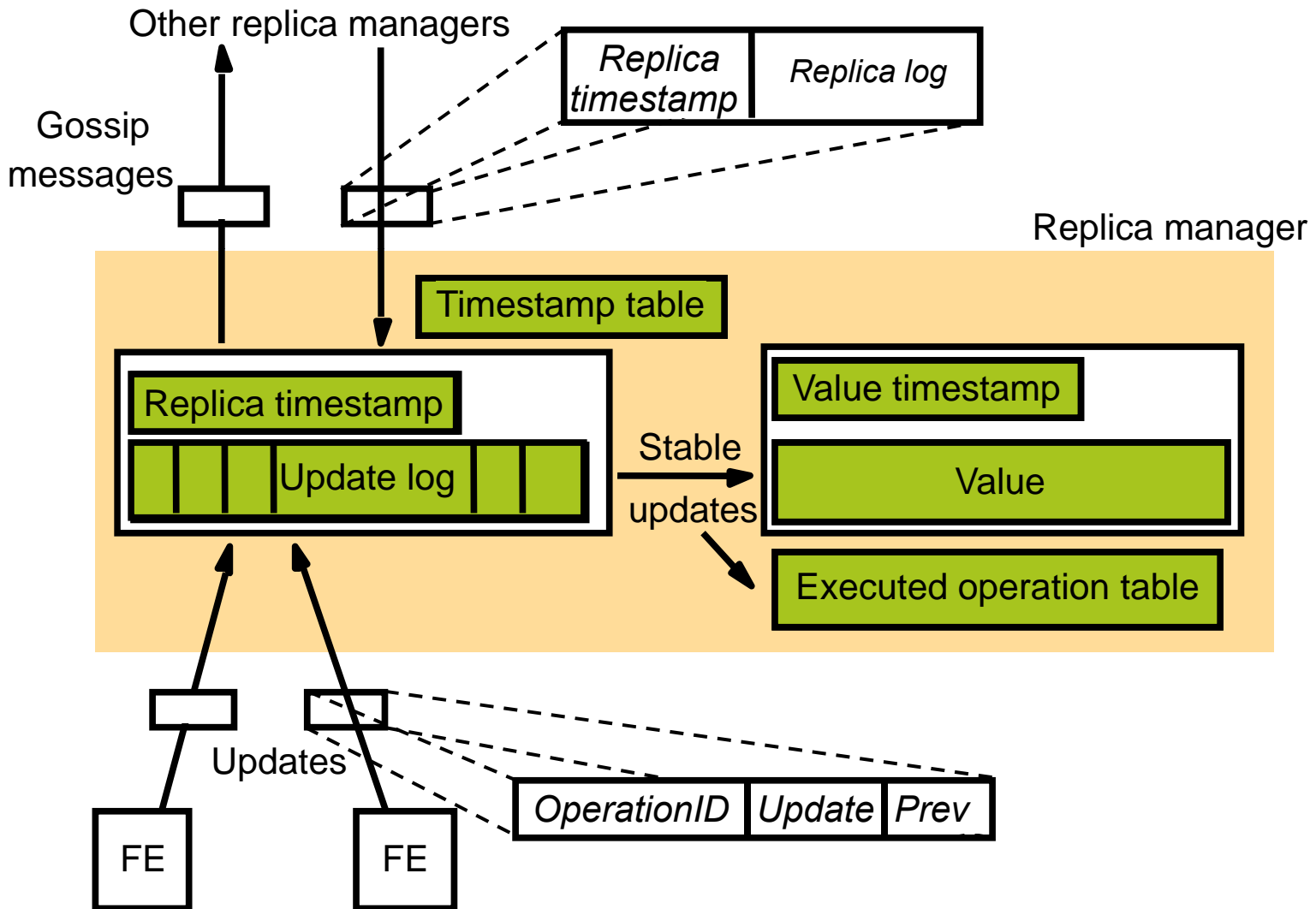
If ($T_f < T_i$)
return value
else {
waits for RMi to be updated
or
query RMj/RMk}

If ($T_f > T_j$)
update RMj
else {
update Client
or
ignore and update RMj}

A gossip replica manager, showing its main state components



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 (Reads)

- When the query q reach the RM
 - If $q.prev \leq valueTS$
 - 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
 - $\langle u.op(par-list), u.prev, u.id \rangle$
 - $u.prev$: 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]++
 - $logRecord = \langle i, ts, u.op, u.prev, u.id \rangle$
 - 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
 - $u.prev \leq valueTS$
 - 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 $r.ts \leq replicaTS$
 - The record r has been already in the local log or has been applied to the value
- Otherwise, insert r in the local log
 - $replicaTS = \text{merge}(replicaTS, m.ts)$

2. Find out the stable updates

- Sort the updates log to find out stable ones, and apply to the value according to the “ \leq ” (thus happens-before) order

3. Update the timestamp table

- If the gossip message is from replica manager j , then $\text{merge}(tableTS[j], m.ts)$

Receiving Gossip 2

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

rm0	{2,4,6}
rm1	{2,3,6}
rm2	{2,5,6}

C=1
r.ts={1,3,5}

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
 - Geographical

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

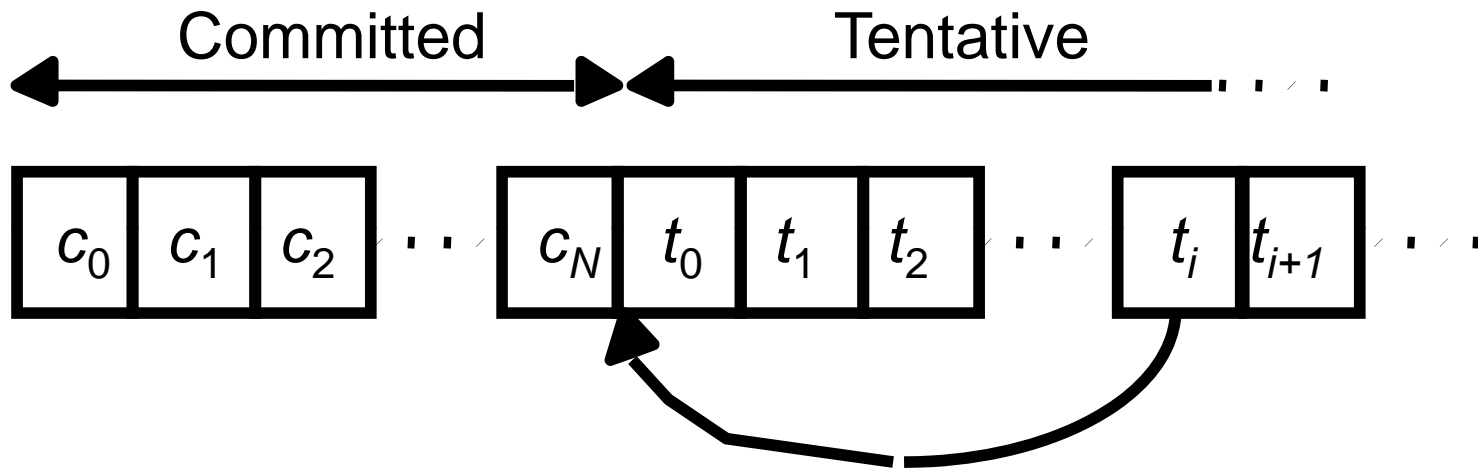
- Provides a high availability by relaxing the consistency guarantees
- When conflicts are rare
- Detect conflicts
 - Relies domain specific conflict detection and resolution
 - Inform user
- Eg
 - Bayou data replication service
 - CODA file system
 - CVS

END

Data replication in Bayou

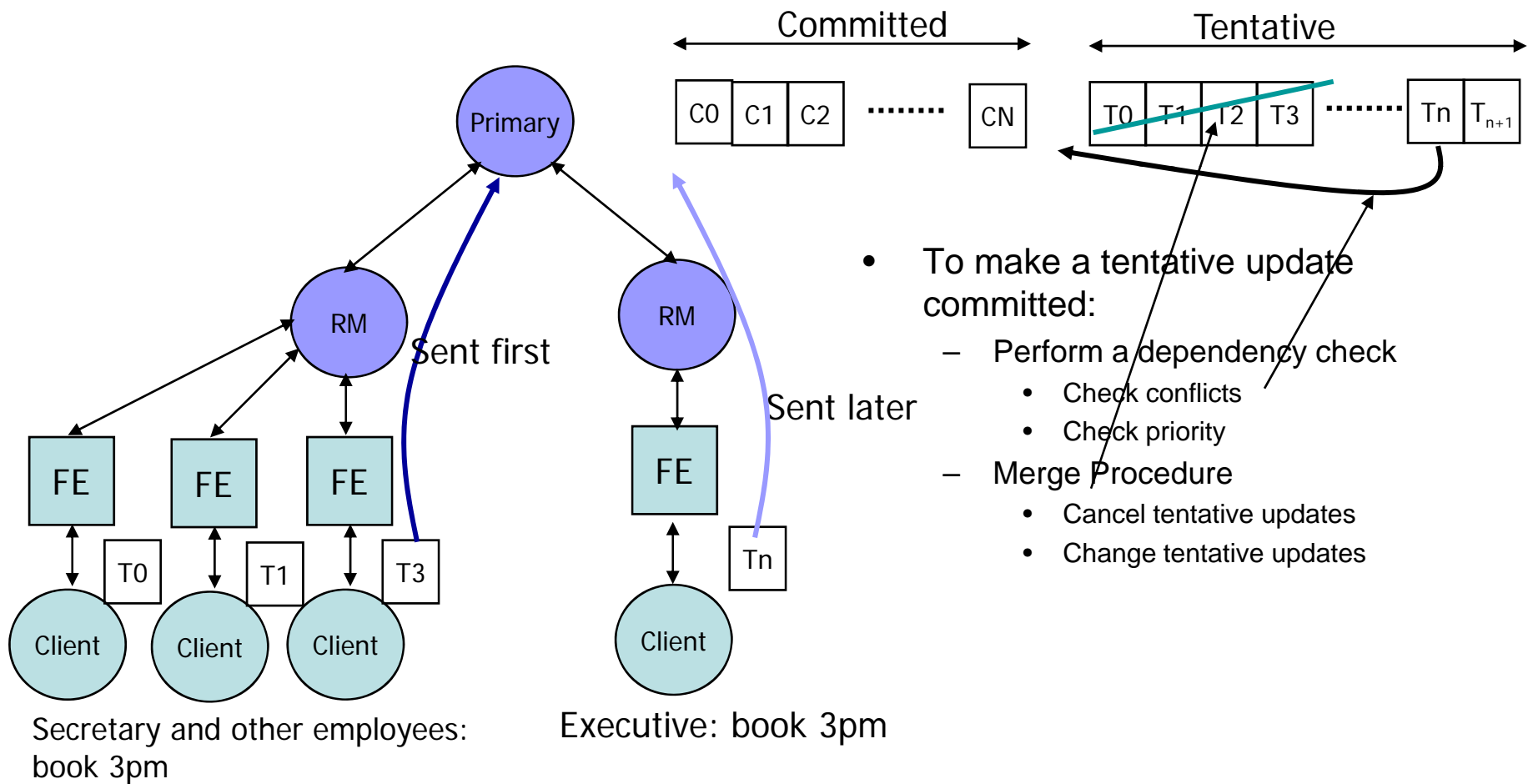
- Provides a high availability by relaxing the consistency guarantees
- Relies domain specific conflict detection and resolution
- Bayou Guarantees
 - *Eventually, every replica manager receives the same set of updates and eventually applies those updates in such a way that the replica manager's databases are identical.*
- Approach:
 - Any user can make updates, and all the updates are applied and recorded at whatever RM they reach.
 - When updates received at any two RM's are merged, the RM's detect and resolve conflicts, using any domain specific dependency check and merge procedure.

Committed and tentative updates in Bayou



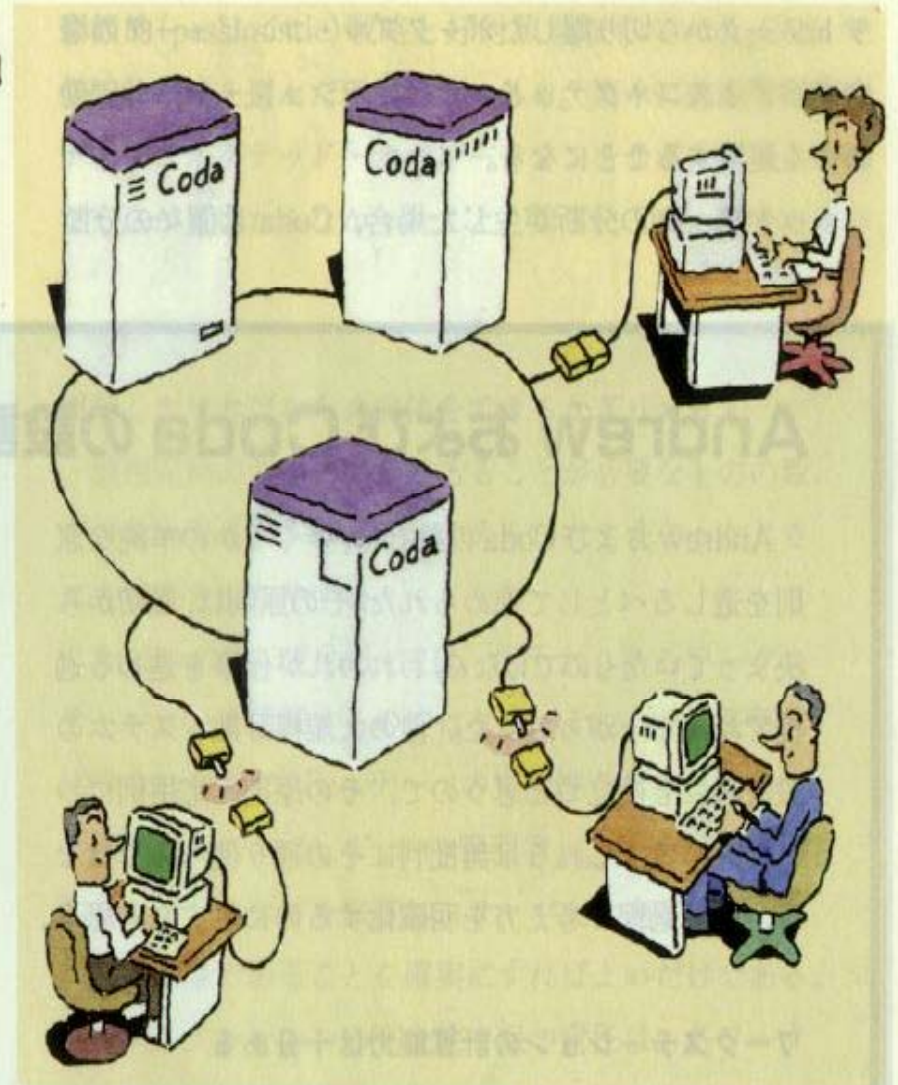
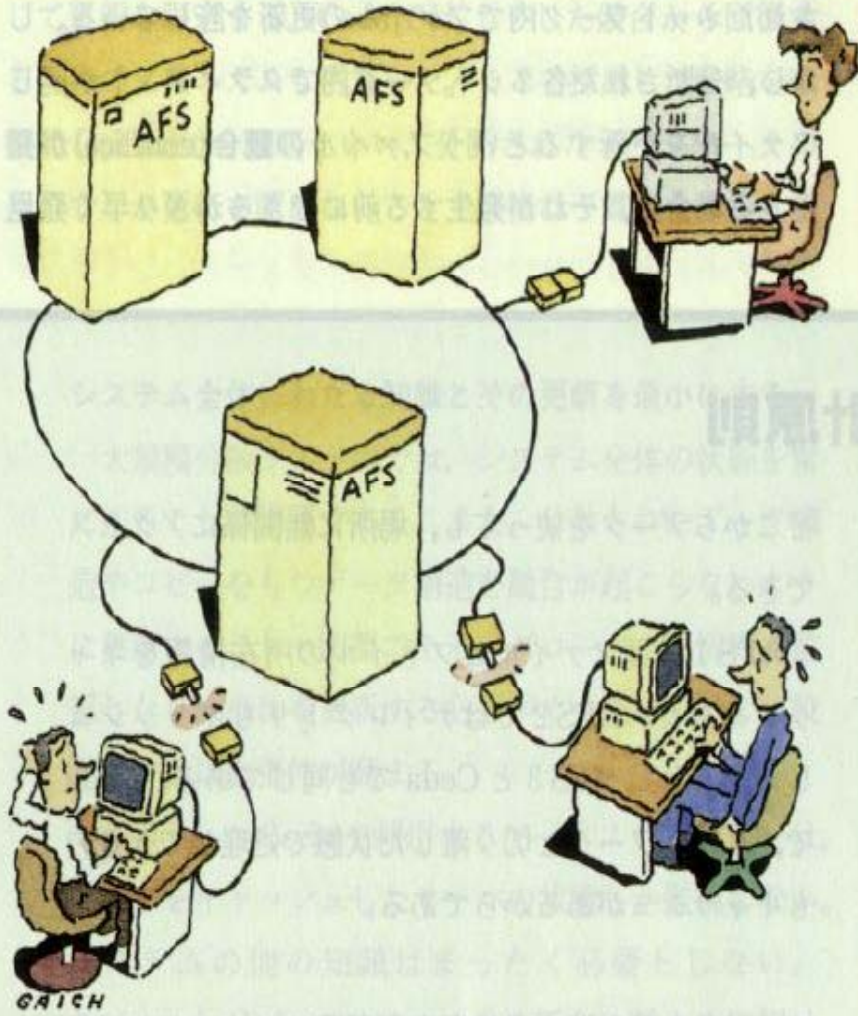
Tentative update t_i becomes the next committed update and is inserted after the last committed update c_N .

Bayou System



- To make a tentative update committed:
 - Perform a dependency check
 - Check conflicts
 - Check priority
 - Merge Procedure
 - Cancel tentative updates
 - Change tentative updates

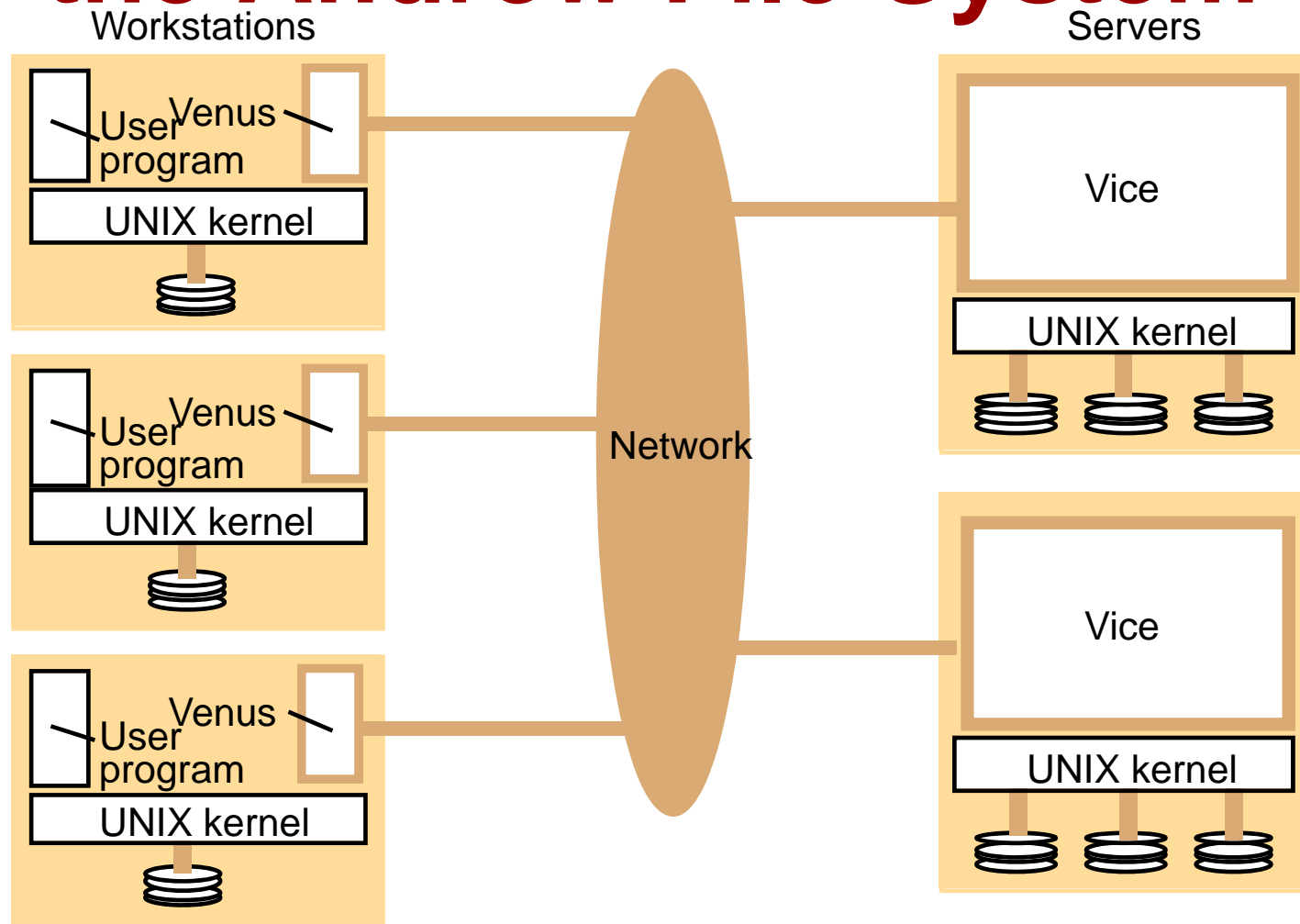
Motivation



The Coda Filesystem

- Limits of AFS
 - Read-only replica
- The objective of Coda
 - Constant data availability
- Coda: extend AFS on
 - Read-write replica
 - Optimistic strategy to resolve conflicts
 - Disconnected operation

Distribution of processes in the Andrew File System



The Coda architecture

- Venus/Vice
 - Vice: replica manager
 - Venus: hybrid of front end and replica manager
- Volume storage group (VSG)
 - The set of servers holding replicas of a file volume
- Available volume storage group (AVSG)
 - Vice know AVSG of each file
- Access a file
 - The file is serviced by any server in AVSG

Coda Operation

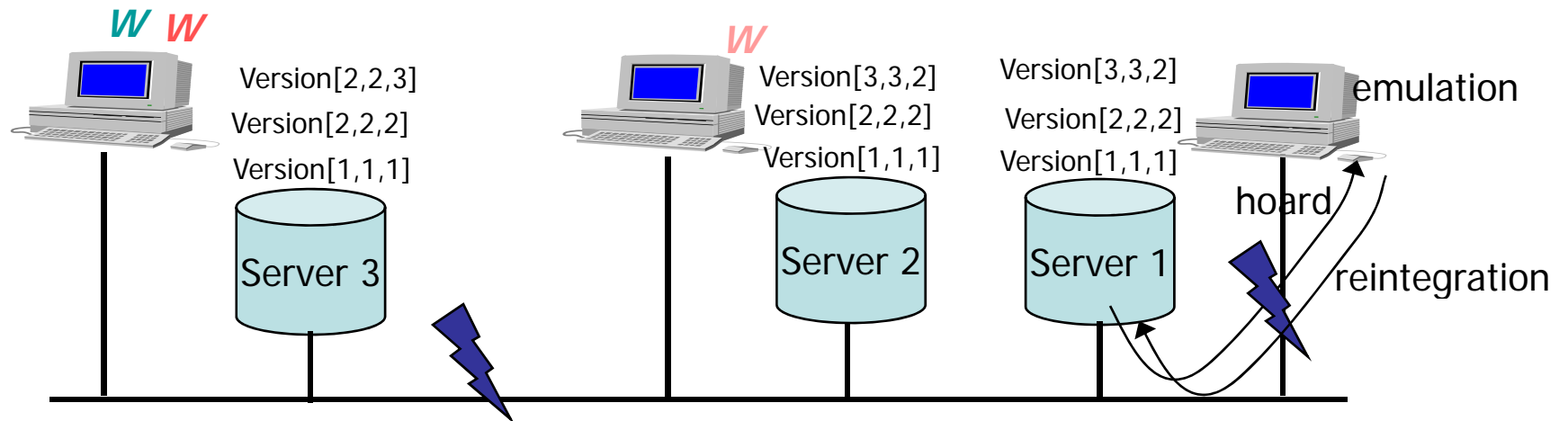
- On close a file
 - Copies of modified files are broadcast in parallel to all of the servers in the AVSG
 - Allow file modification when the network is partitioned
 - When network partition is repaired, new updates are reapplied to the file copies in other partition
 - Meanwhile, file conflict is detected
- Disconnected operation
 - When the file's AVSG becomes empty, and the file is in the cache
 - Updates in the disconnected operation apply on the server later on when AVSG becomes nonempty
 - if there are conflicts, resolve manually

Replication Strategy

- Coda version vector (CVV)
 - Attached to each version of a file
 - Each element of the CVV is an estimate of the number of modifications performed on the version of the file that is held at the corresponding server
- Example: $CVV = (2,2,1)$
 - The replica on server1 has received 2 updates
 - The replica on server2 has received 2 updates
 - The replica on server3 has received 1 updates

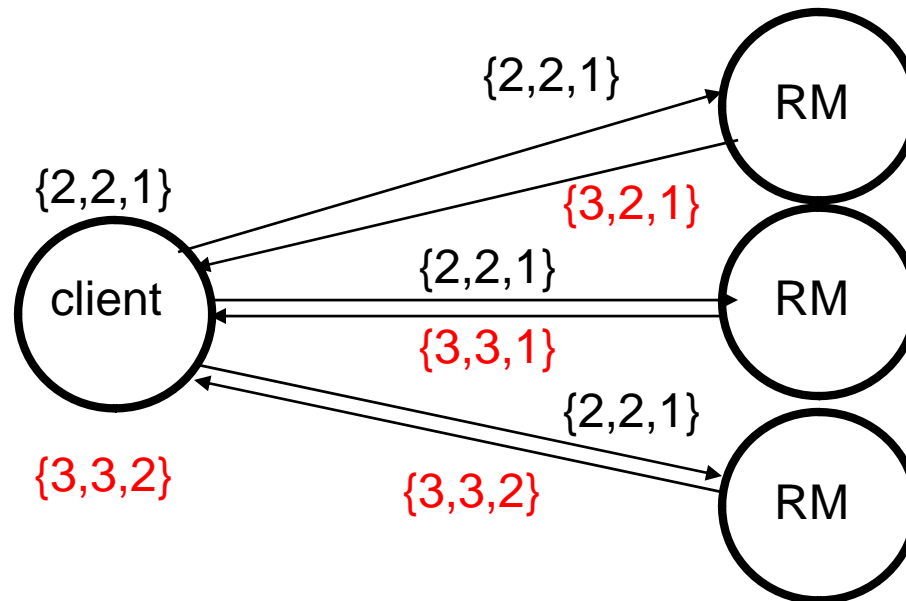
Coda File System

1. Normal case:
 - Read-any, write-all protocol
 - Whenever a client writes back its file, it increments the file version at each server.
2. Network disconnection:
 - A client writes back its file to only available servers.
 - Version conflicts are detected (not resolved) automatically when network is reconnected
3. Client disconnection:
 - A client caches as many files as possible (in hoard walking).
 - A client works in local if disconnected (in emulation mode).
 - A client writes back updated files to servers (in reintegration mode).



Construction of CVV

- When a modified file is closed
 - Multicast the file with current CVV to AVSG
 - Each server in AVSG increase the corresponding element of CVV, and return it to the client
 - The client merge all returned CVV as the new CVV, and distribute it to AVSG



Example

- **File F is replicated at 3 servers: $s1, s2, s3$**
 - $VSG = \{s1, s2, s3\}$
 - F is modified at the same time by $c1$ and $c2$
 - Because network partition, AVSG of $c1$ is $\{s1, s2\}$, AVSG of $c2$ is $\{s3\}$
- **Initially**
 - The CVVs for F at all 3 servers are $[1, 1, 1]$
- **$C1$ updates the file and close**
 - the CVVs at $s1$ and $s2$ become $[2, 2, 1]$
 - There is an update applied on $s1$ and $s2$ since beginning
- **$C2$ updates the file twice**
 - The CVV at $s3$ become $[1, 1, 3]$
 - There are two updates applied on $s3$ since beginning

Example contd.

- When the network failure is repaired
 - C2 modify AVSG to $\{s1,s2,s3\}$ and requests the CVVs for F from all members of the new AVSG
 - Let $v1$ be CVV of a file at server1, and $v2$ the CVV of the file at server2
 - $v1 \geq v2$, or $v1 \leq v2$: $[2,2,2]$ vs $[2,2,1]$ no conflict
 - Neither $v1 \geq v2$, nor $v2 \geq v1$: conflict
 - C2 find $[2,2,1] \langle \rangle [1,1,3]$, that means conflict happens
 - Conflict means concurrent updates when network partitioned
 - C2 manually resolve the conflict

Venus - states

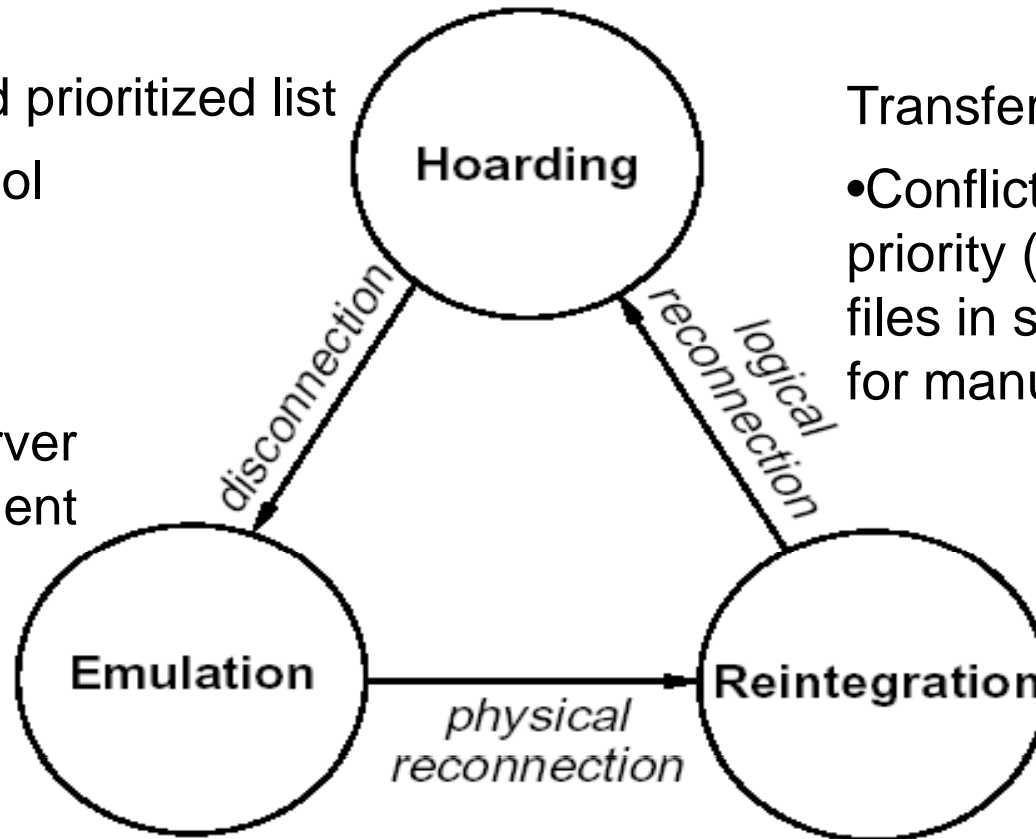
Hoarding: Filling up cache in advance with appropriate files

- LRU
- users specified prioritized list
- “Bracketing” tool

Transfer updates to AVSG

- Conflict: server copy has priority (store conflicting files in separate directory for manual resolution)

Behavior of server emulated on client



END

Distributed Systems
Lecture 11
Replication

Josva Kleist

Unit for Distributed Systems and
Semantics

Aalborg University

Replication

- Fault tolerance
- Increased availability
- Increased performance

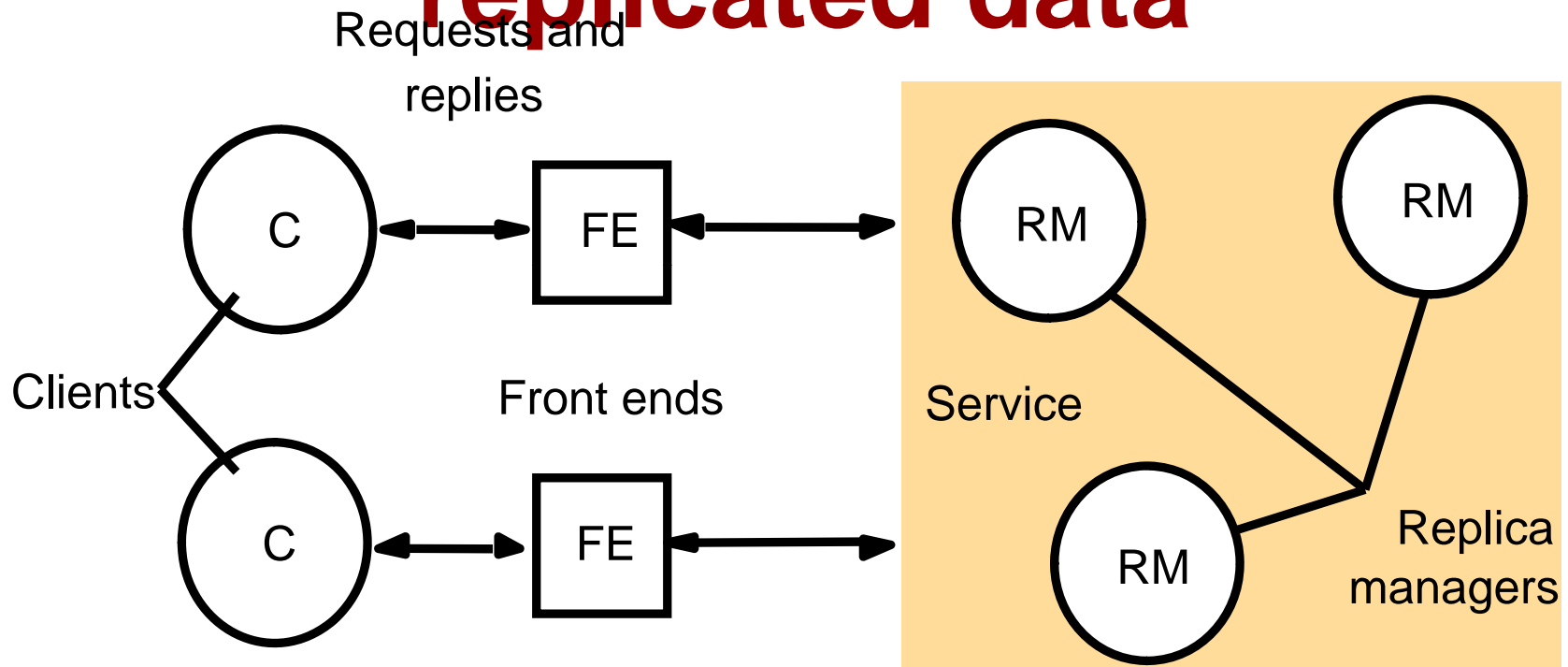
Requirements

- Replication transparency
- Consistency (possibly in a modified form)

Assumptions

- Asynchronous system in which processes may fail only by crashing.
- No network partitions.
- Replica managers apply operations recoverably.
- Sometimes we require a replica manager to be a state machine.

A basic architectural model for the management of replicated data



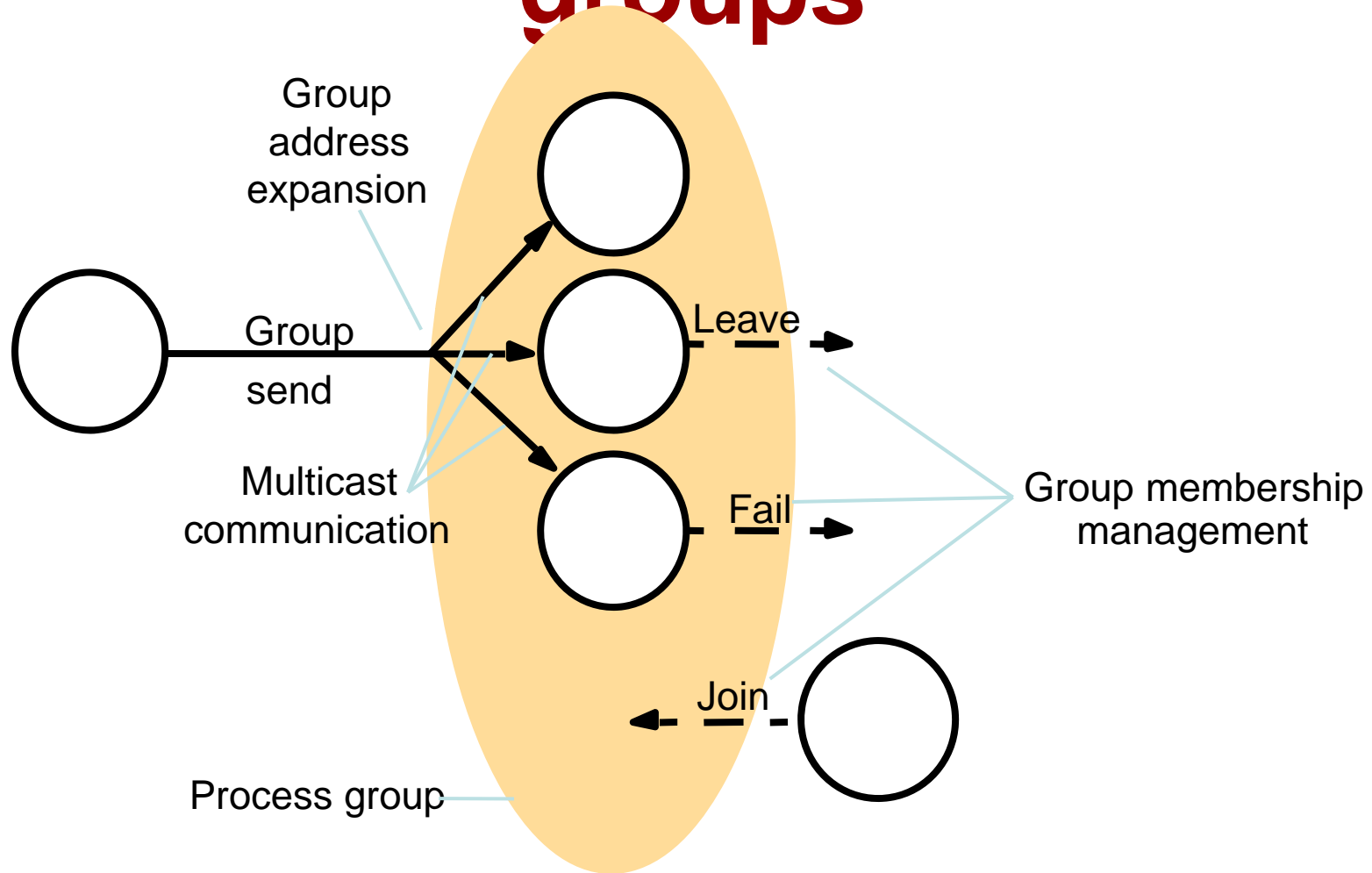
General replication model

- Coordination
- Execution
- Agreement
- Response

Coordination

- *FIFO*: If FE issues request r then r' , then any correct RM that handles r' handles r before it.
- *Causal*: If the issue of request r happens before req r' , then any correct RM that handles r' handles r before it.
- *Total*: If a correct RM handles r before r' , then any correct RM that handles r' handles r before it.

Services provided for process groups



Views

- We need a consistent perception of who is a member of a group.
- A *view* is a list of who is member of a group.
- The group membership service delivers *views* to the members of a group.

Requirements for view delivery

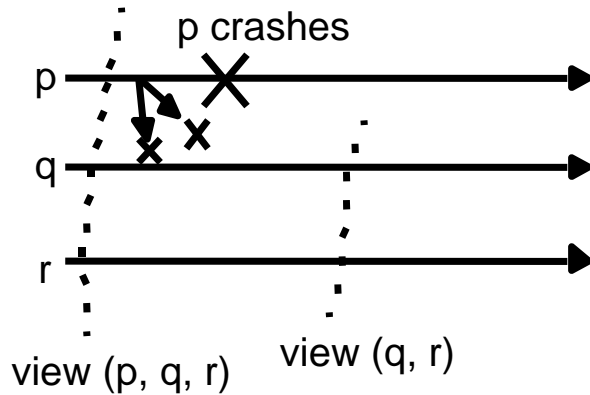
- *Ordering*: If a process P delivers $view(g)$ and then $view'(g)$, then no Q , deliver $view'(g)$ before $view(g)$.
- *Integrity*: If a process P delivers $view(g)$, then $P2 view(g)$.
- *Non-triviality*: If a process Q joins a group and is or becomes indefinitely reachable from process $P^1 Q$, then eventually Q is always in the view that P delivers. Similarly, if the group partitions and

View synchronous group communication

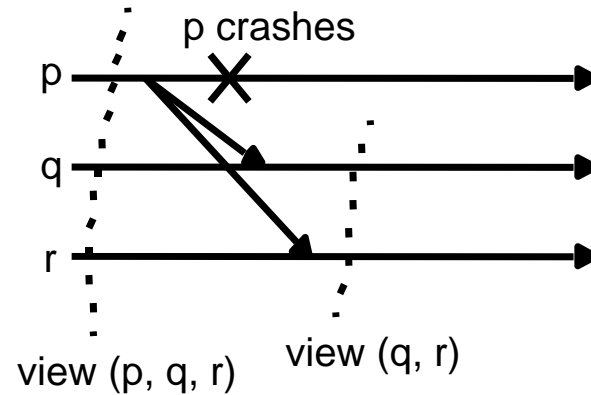
- *Agreement*: Correct processes deliver the same set of messages in any given view.
- *Integrity*: If process P delivers message m , then P will not deliver m again.
Furthermore, if $P \in group(m)$ and m was supplied to a *multicast* operation by $sender(m)$.
- *Validity*: Correct processes always deliver the messages that they send. If the system fails to deliver a message to any

View-synchronous group communication

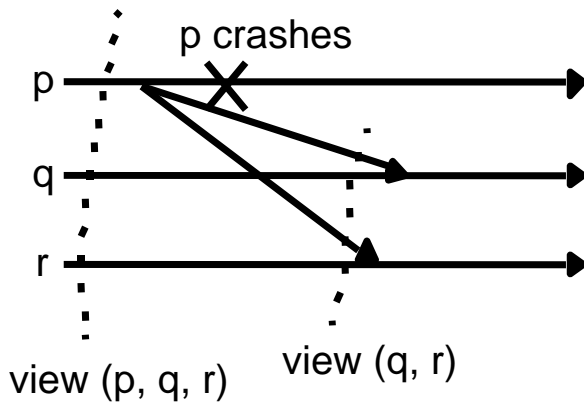
a (allowed).



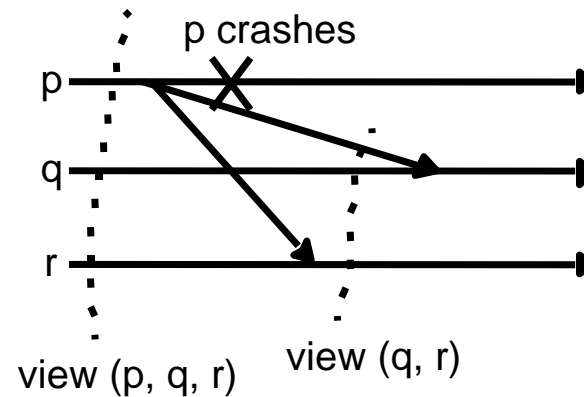
b (allowed).



c (disallowed).



d (disallowed).



Fault-tolerance

- Provide a service that is correct event 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 from an implementation with replicated data and one provided by a single correct replica manager (*replication transparency*).

Consistency problems

Client 1

setBalance_B(x,1)

setBalance_A(y,2)

Client 2:

getBalance_A(y) → 2

getBalance_A(x) → 0

Linear consistency

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of the operations in the interleaving is consistent with the real time at which the operations occurred in the actual execution.

Sequential consistency

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of the operations in the interleaving is consistent with the program order in which each individual client executed them.

Sequential consistency

Client 1

setBalance_B(x,1)

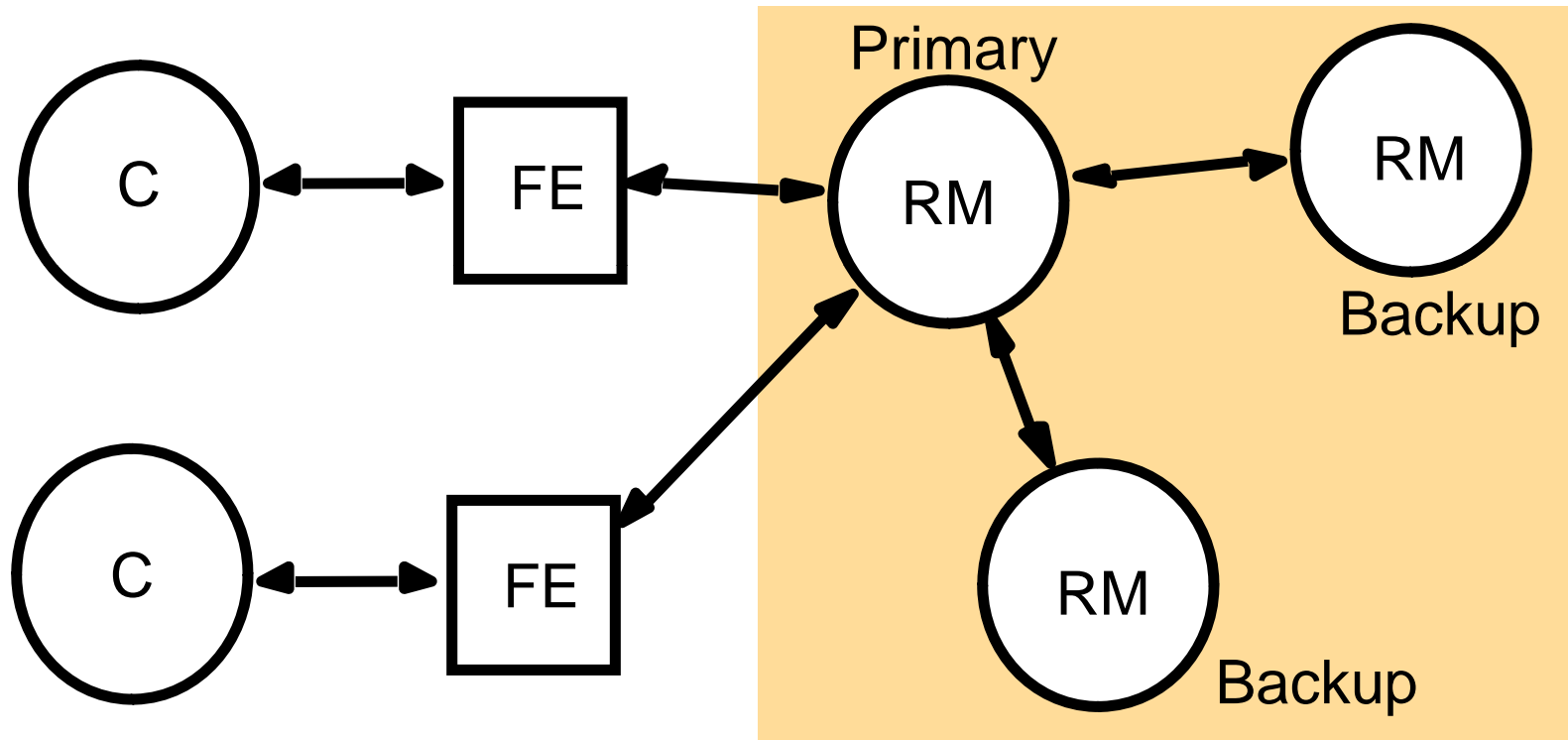
setBalance_A(y,2)

Client 2:

getBalance_A(y) → 0

getBalance_A(x) → 0

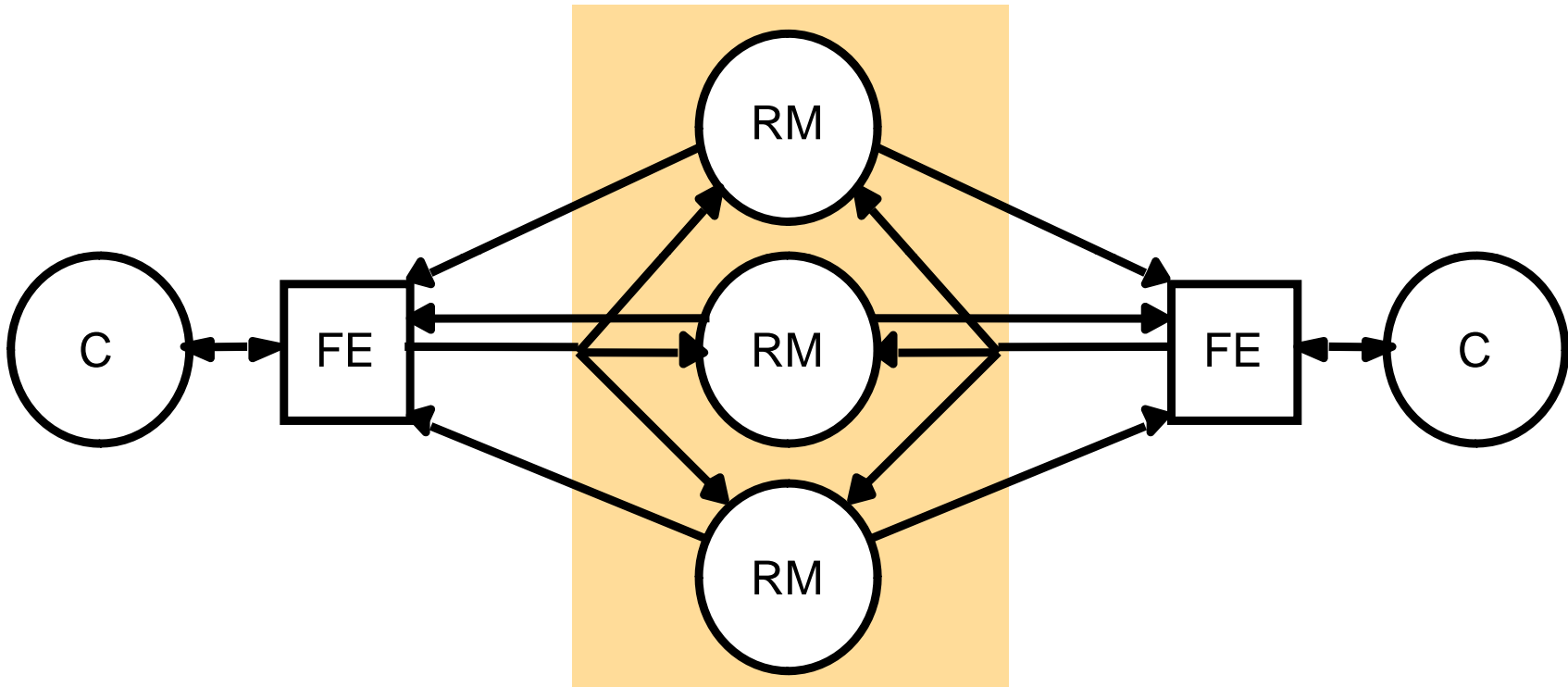
The passive (primary-backup) model for fault tolerance



Passive replication

- *Request:* FE issues the request, containing a unique ID, to the primary RM.
- *Coordination:* The primary RM takes each request atomically, in the order in which it receives it. It checks the unique identifier to catch duplicate requests, and if so simply re-sends the response.
- *Execution:* The primary RM executes the request and stores the response
- *Agreement:* If the request is an update

Active replication



Active replication

- *Request:* The FE issues the request, containing a unique ID, to the group of RM's, using a totally ordered, reliable multicast primitive. The FE is assumed to fail by crashing at worst. It does not issue the next req. until it has received a response.
- *Coordination:* The group communication system delivers the request to every correct RM in the same total order.

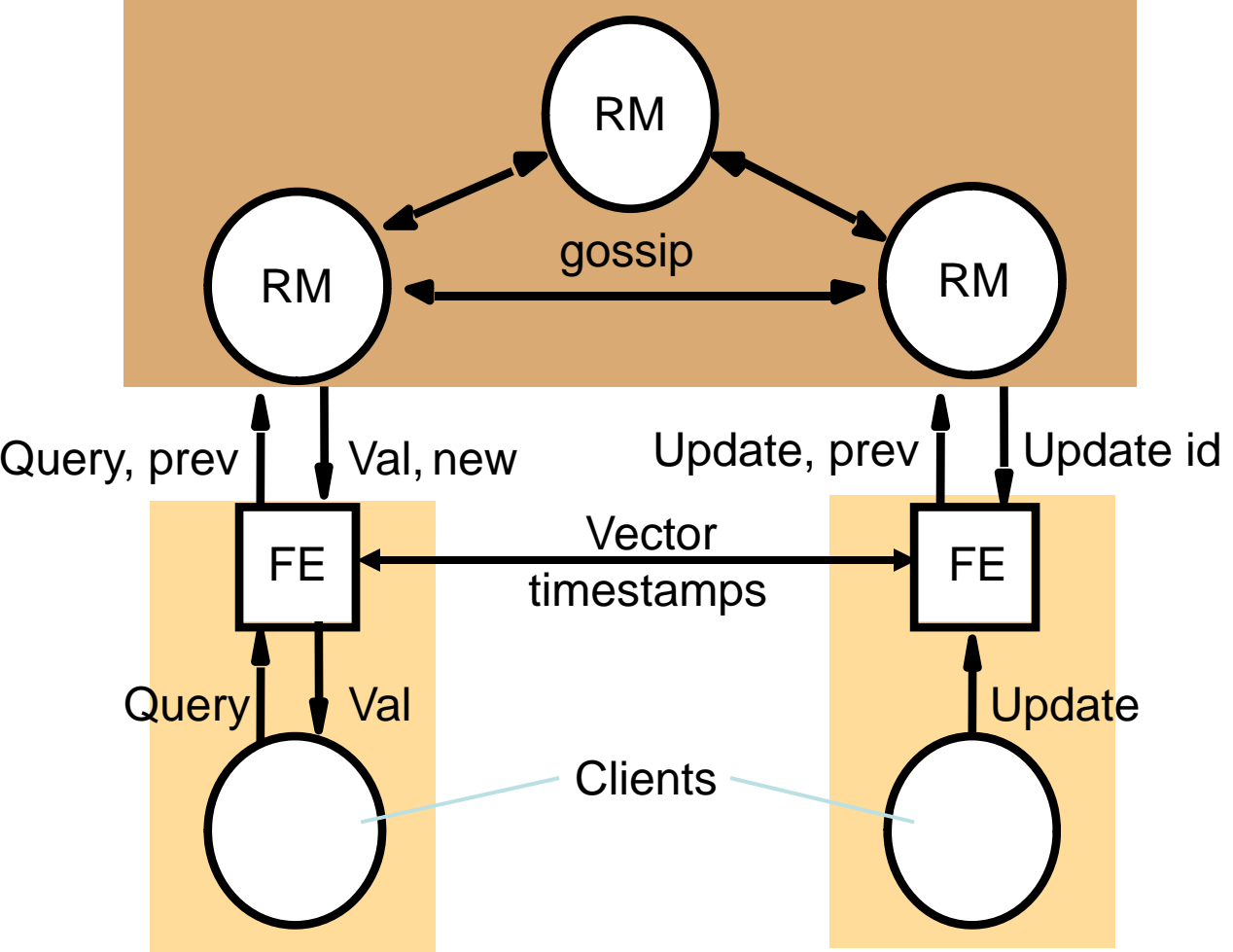
High availability services

- Obtain access to a service for as much time as possible.
- Provide reasonable response times.
- Possibly relax the consistency requirements to replicas.

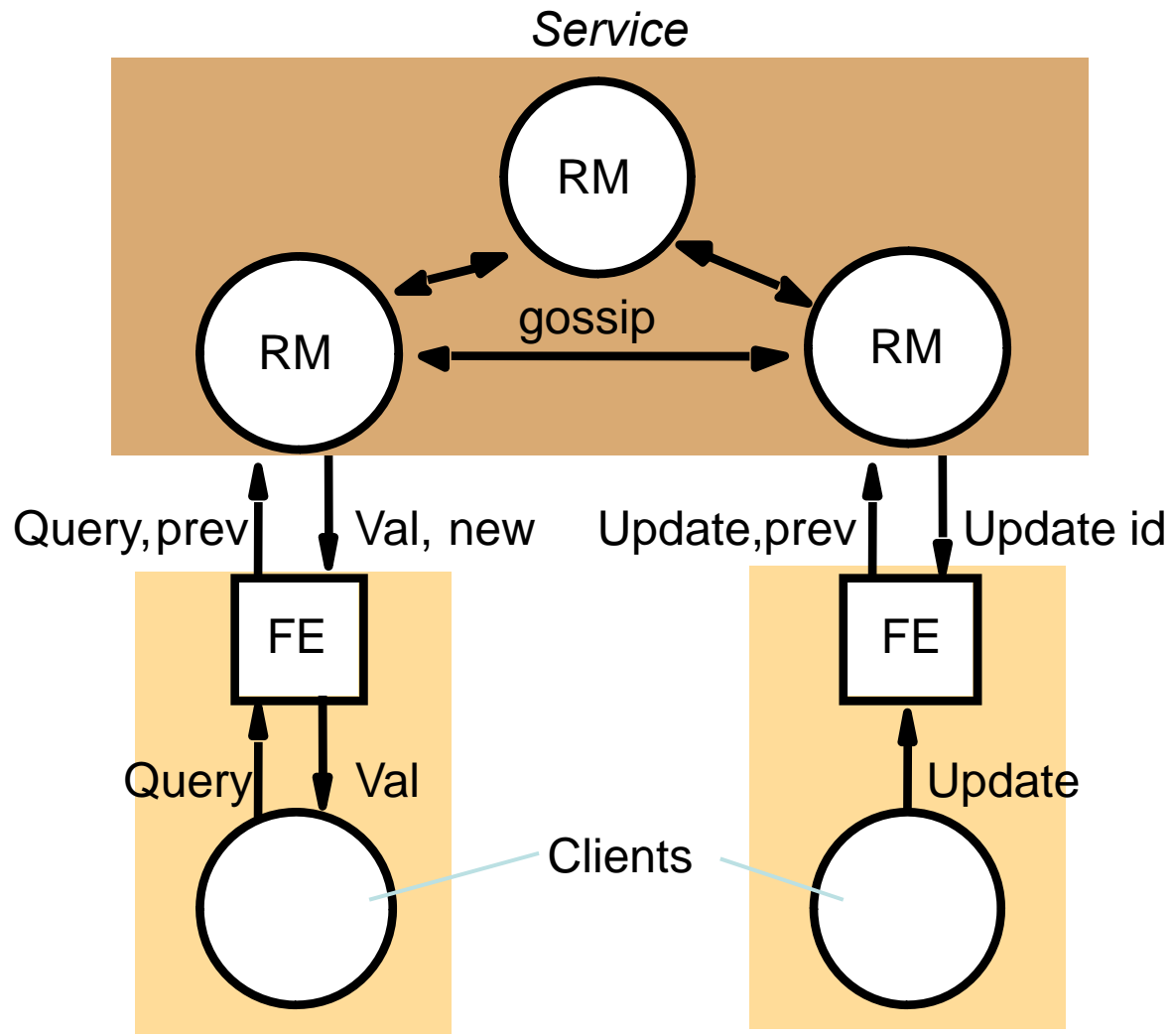
Case study: The gossip architecture

- Provide high availability by replicating data close to users.
- Provide clients with a consistent service over time.
- Allow for relaxed consistency between replicas:
 - Causal,
 - forced (total and causal),
 - immediate (consistent to any other update at all RM's).

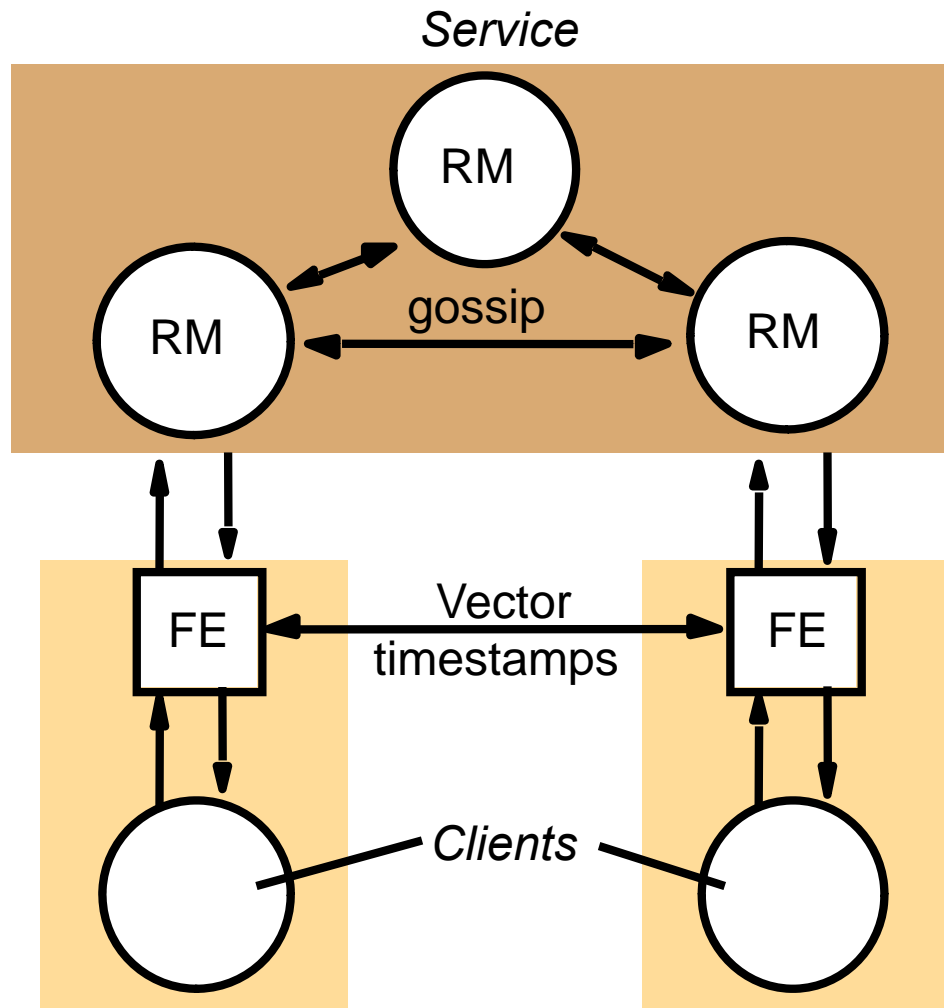
Query and update operations in a gossip service



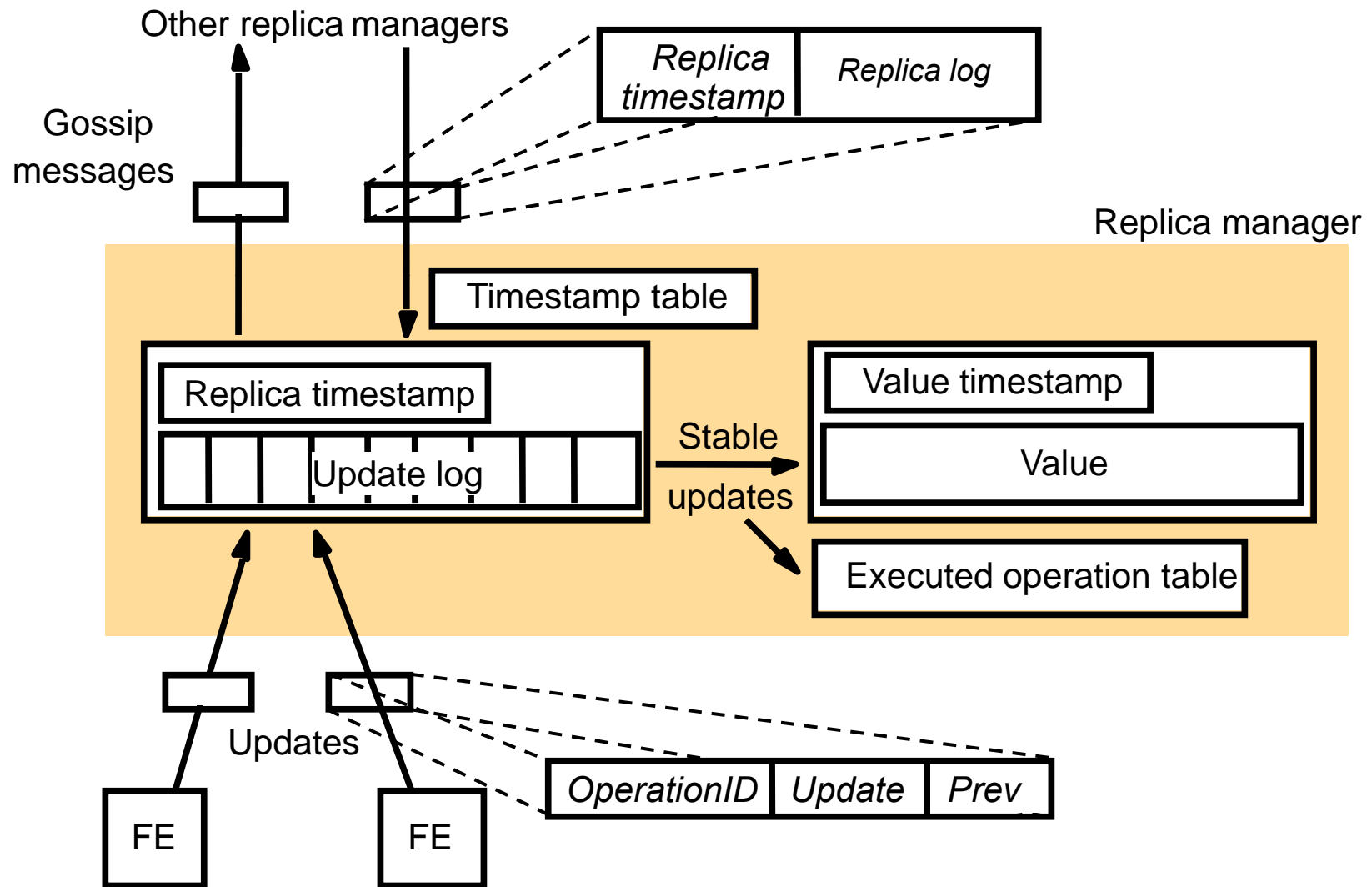
Query and update operations in a gossip service



Front ends propagate their timestamps whenever clients communicate directly



A gossip replica manager, showing its main state components



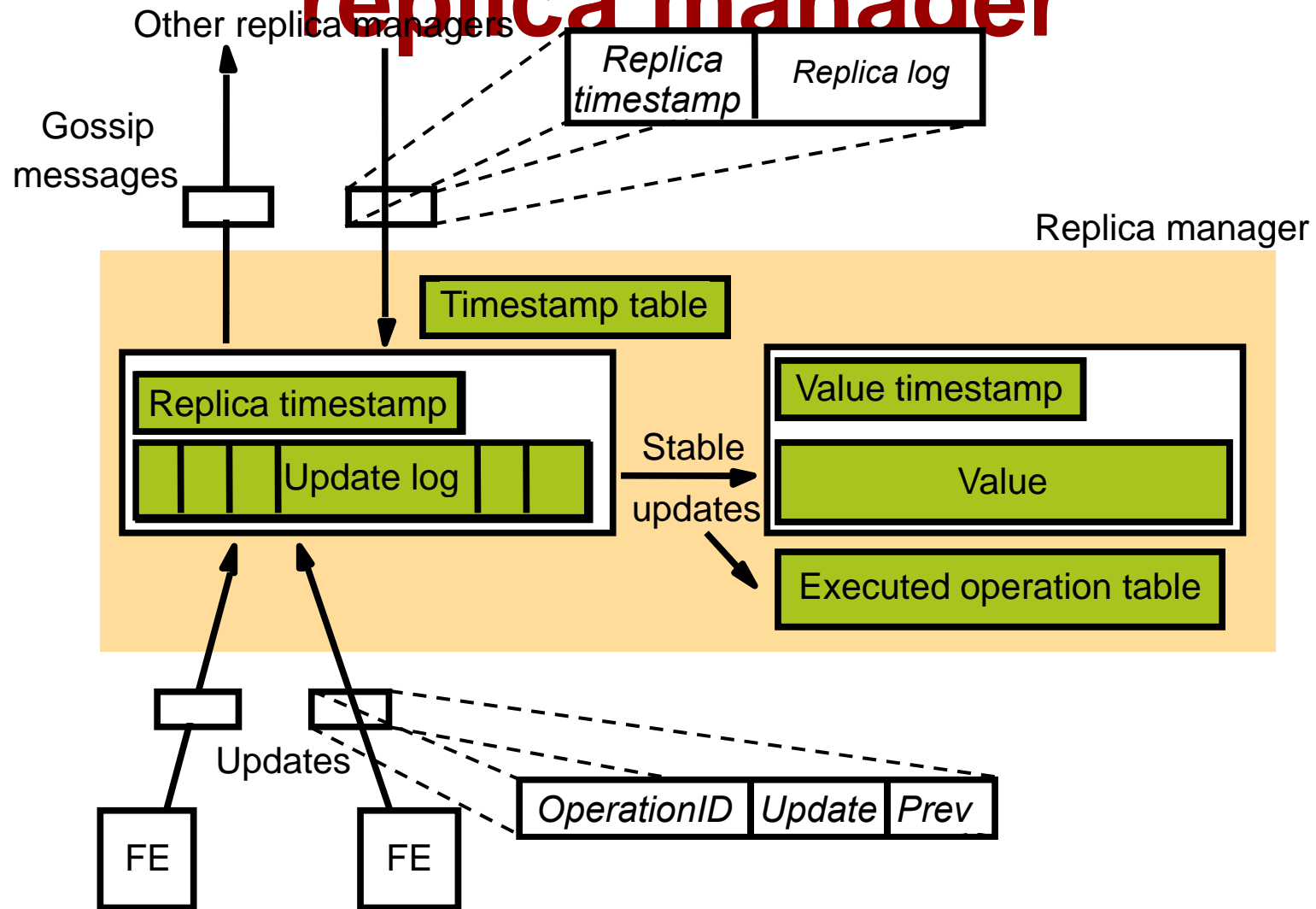
Gossip processing of queries and updates

- *Request*: A FE sends a request to a single RM. A FE can communicate with different RM's.
- *Answer*: If the req is an update then the RM replies as soon as it has received the update.
- *Coordination*: The RM that receives a request does not process it until it can apply the request according to the required ordering constraints. This may

Gossip

- *Execution*: The RM executes the request.
- *Answer*: If the req is a query then the RM replies at this point.
- *Agreement*: The RM update one another by exchanging *gossip messages*, which contain the most recent messages they have received.

Main components of a gossip replica manager



Case study: Bayou

- Provides a high availability by relaxing the consistency guarantees.
- Relies domain specific conflict detection and resolution.

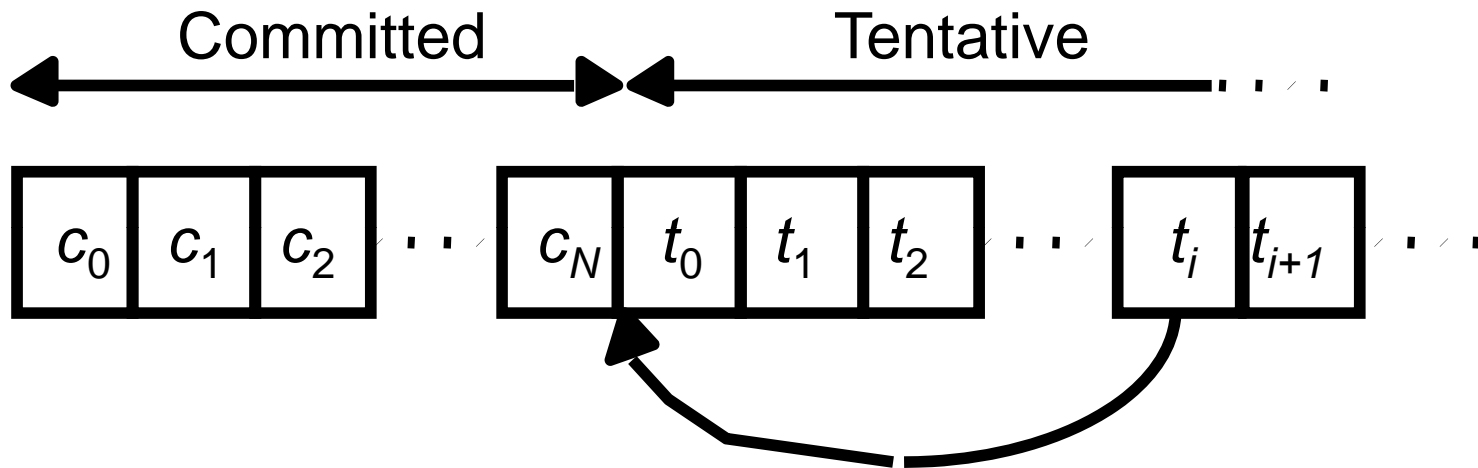
Bayou guarantee

Eventually, every replica manager receives the same set of updates and eventually applies those updates in such a way that the replica manager's databases are identical.

Approach

- Any user can make updates, and all the updates are applied and recorded at whatever RM they reach.
- When updates received at any two RM's are merged, the RM's detect and resolve conflicts, using any domain specific criteria.

Committed and tentative updates in Bayou



Tentative update t_i becomes the next committed update and is inserted after the last committed update c_N .

Case study: Coda

- A descendant of the Andrews file system.
- Coda allow a file to be replicated (in AFS on *read-only* files could be replicated).
- Coda allows for for portable computers to become disconnected while still having access to their files.

Terminology

- The set of servers holding replicas of a file volume is known as the *volume storage group (VSG)*.
- The set of servers available to a client wishing to open a file is called a *available volume storage group (AVSG)*.
- A client is considered *disconnected* if its AVSG is empty.

Workings

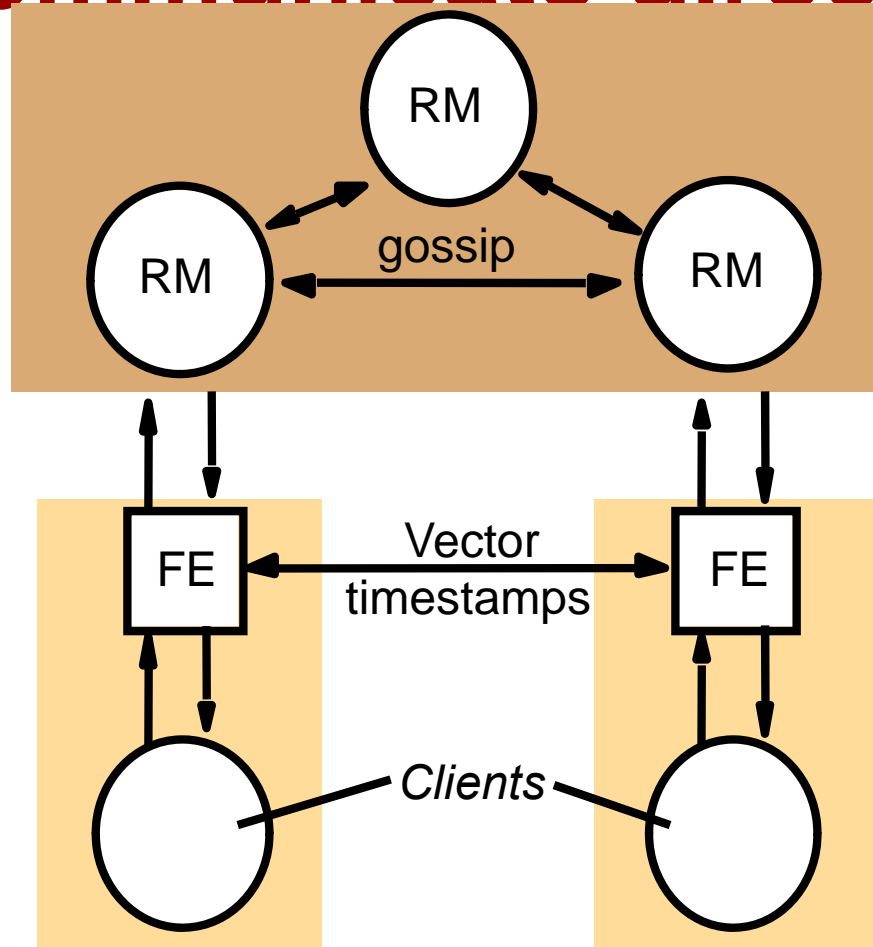
- On *opening* a file, a client request the file from one of the servers in its AVSG.
- *Call-back-promices* are (as in AFS) used to inform a client of a modified file.
- On *closing* a file, the modified file is *broadcast* to all servers in the AVSG.

Replication strategy

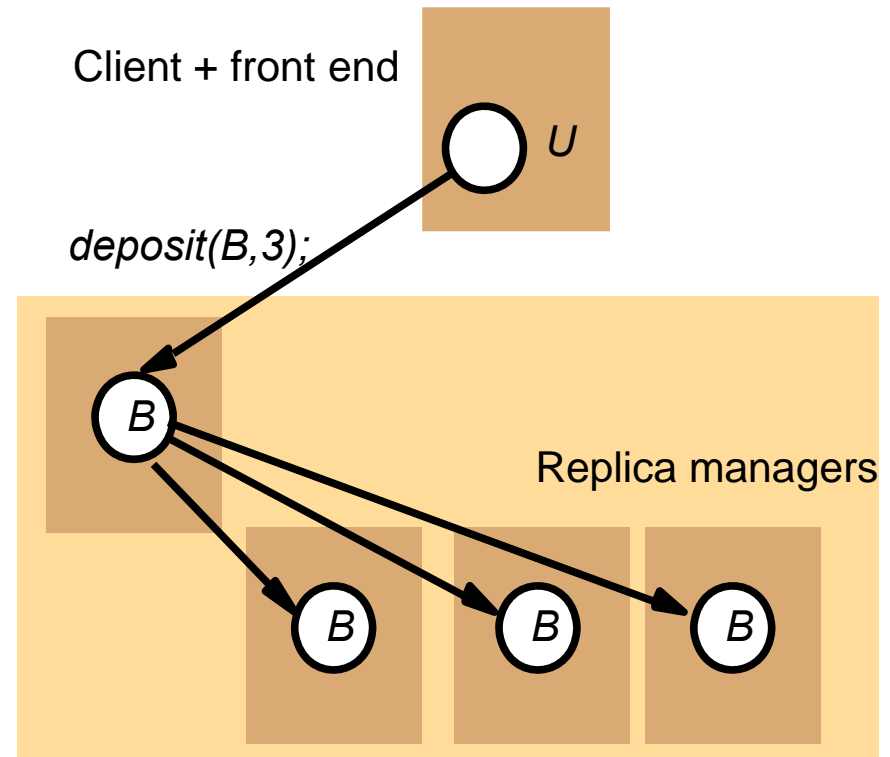
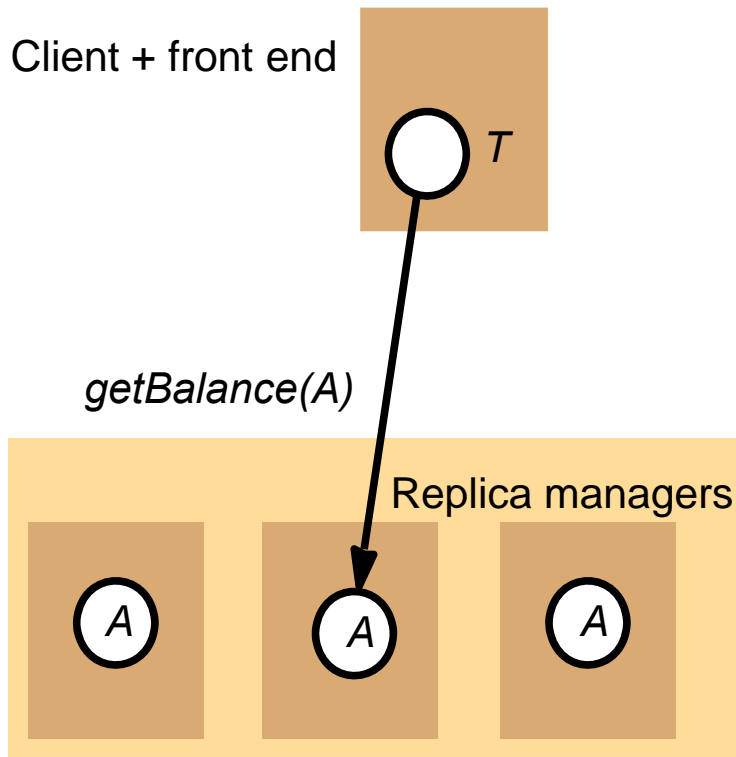
- Each version of file is attached a *Coda version vector (CVV)*.
- A CVV is a vector timestamp containing one element for each server in the file's VSG.
- Allows detection of conflicts:
 - If the CVV at one site is greater than or equal to all the corresponding CVVs at other sites, then there is no conflict between the replicas of the file.
 - On the other hand, if two CVVs are not related

End of show

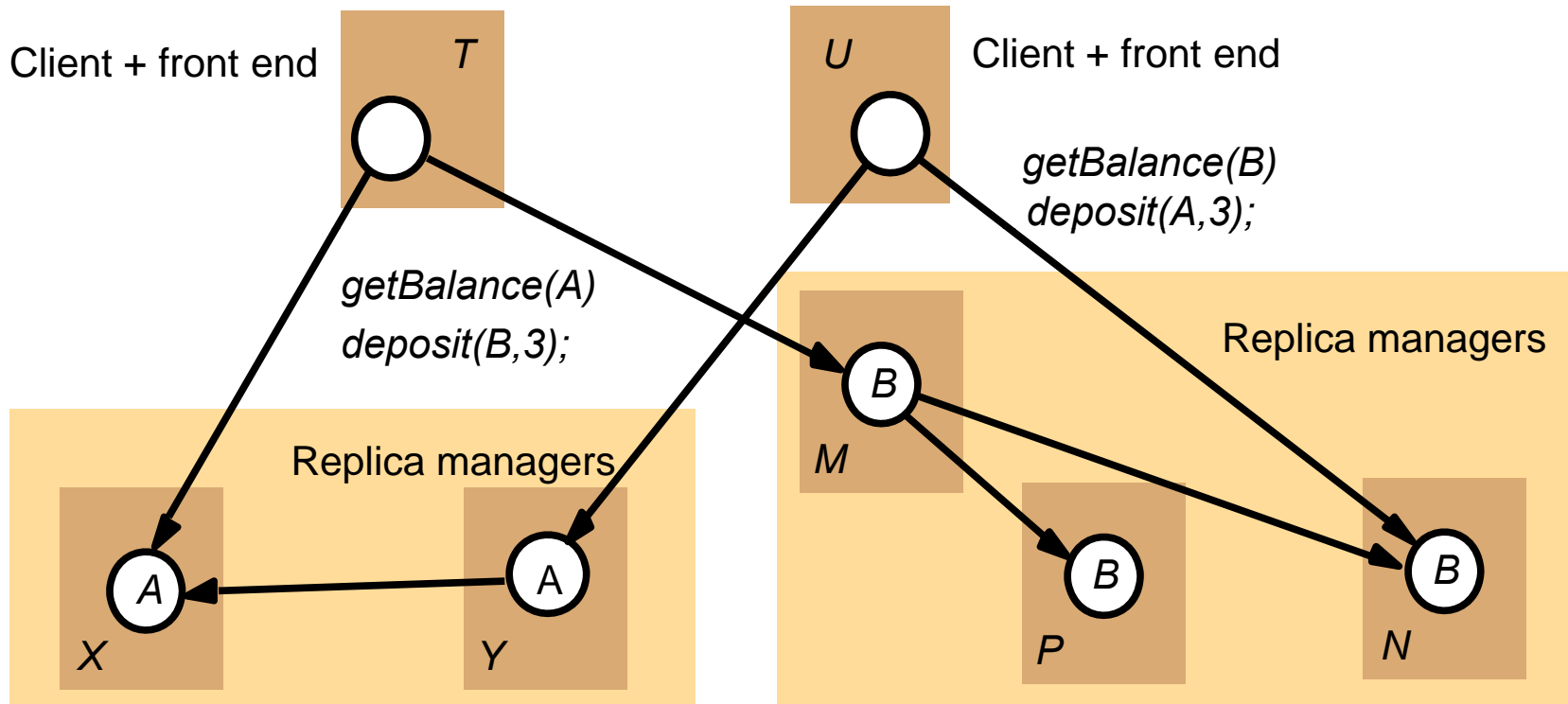
Front ends propagate their timestamps whenever clients communicate directly



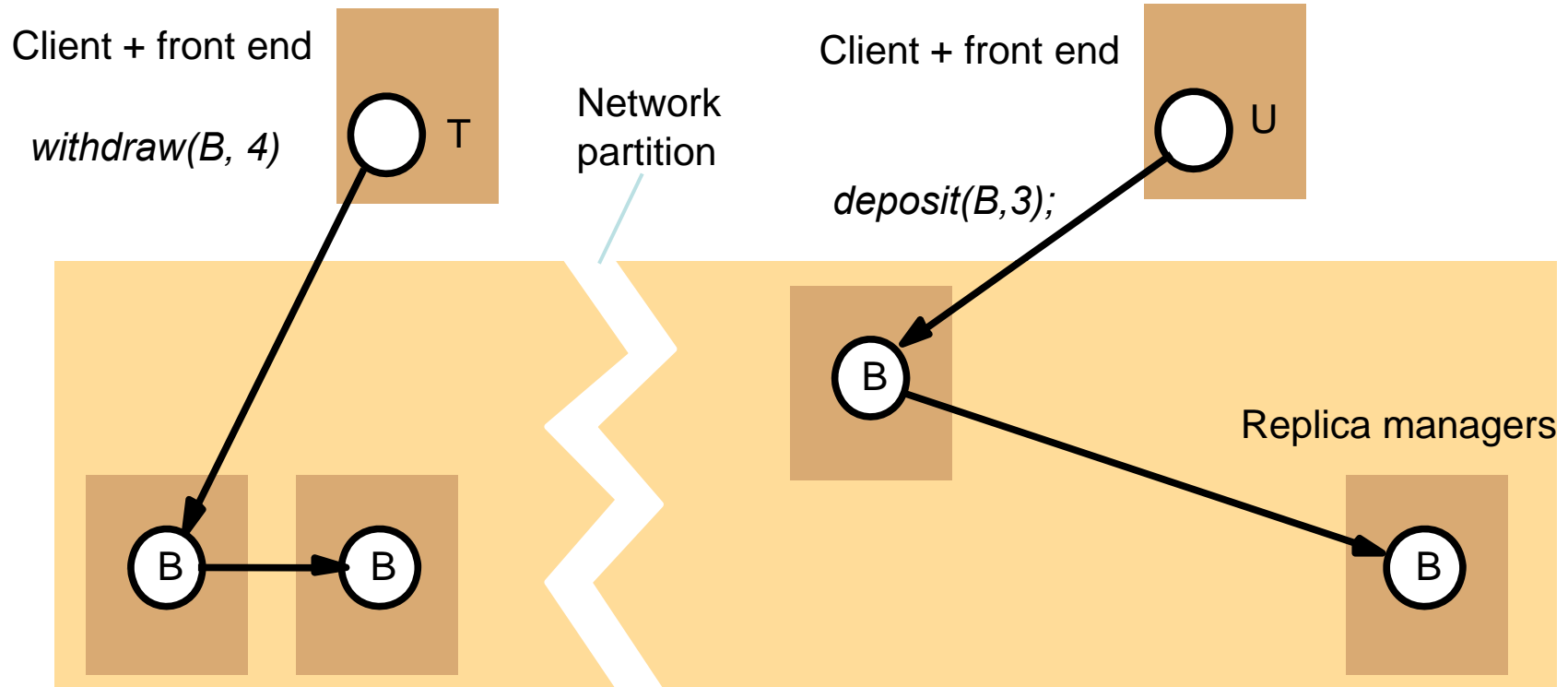
Transactions on replicated data



Available copies



Network partition



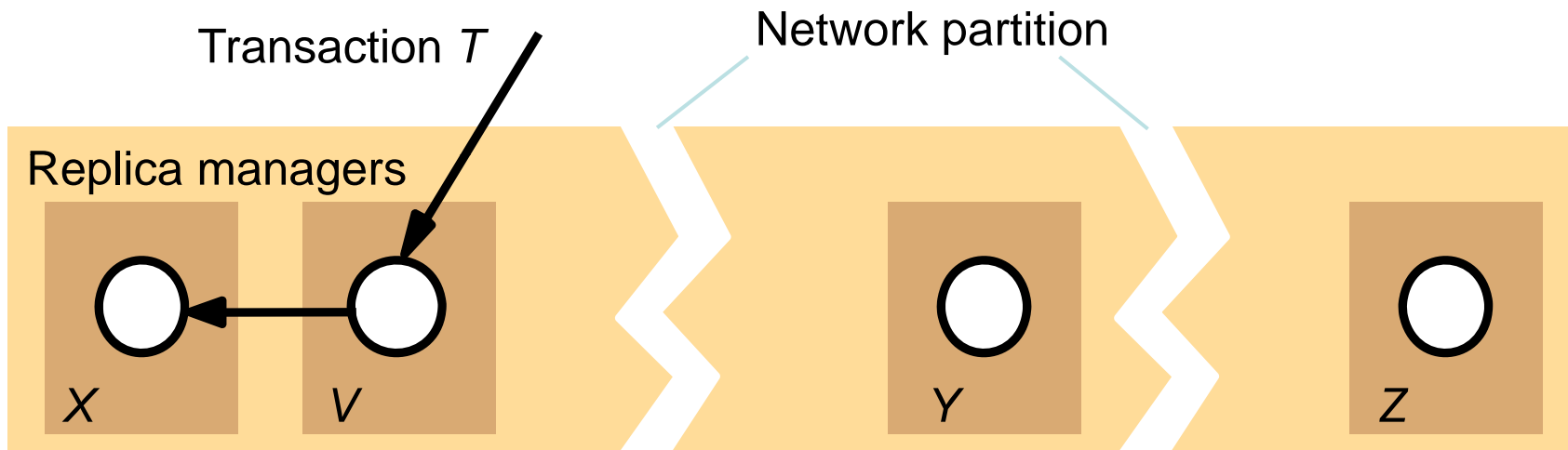
Gifford's quorum consensus examples

		Example 1	Example 2	Example 3
<i>Latency</i> (milliseconds)	Replica 1	75	75	75
	Replica 2	65	100	750
	Replica 3	65	750	750
<i>Voting configuration</i>	Replica 1	1	2	1
	Replica 2	0	1	1
	Replica 3	0	1	1
<i>Quorum sizes</i>	<i>R</i>	1	2	1
	<i>W</i>	1	3	3

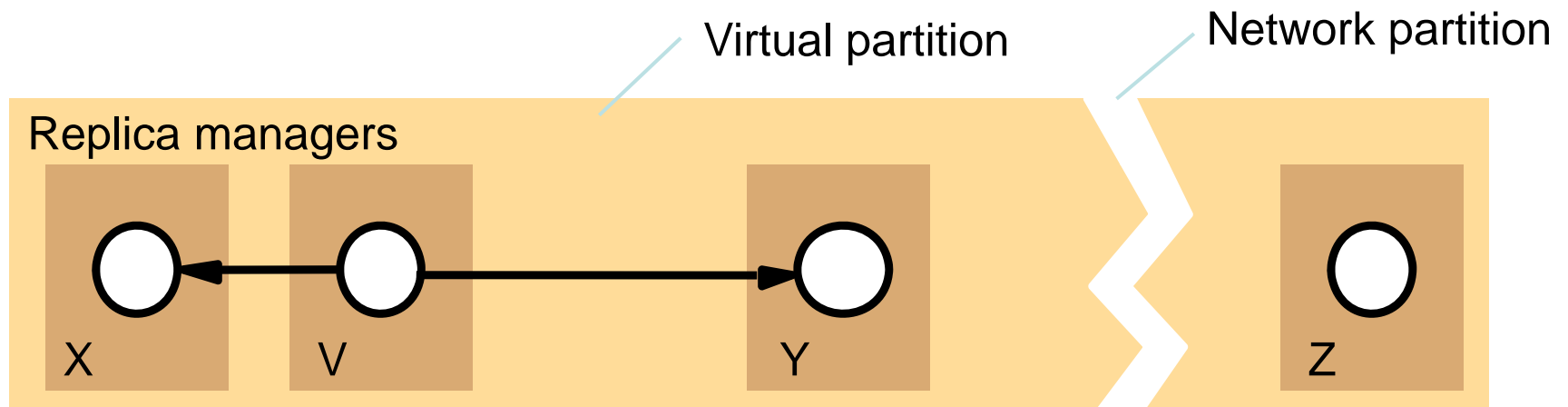
Derived performance of file suite:

<i>Read</i>	Latency	65	75	75
	Blocking probability	0.01	0.0002	0.000001
<i>Write</i>	Latency	75	100	750
	Blocking probability	0.01	0.0101	0.03

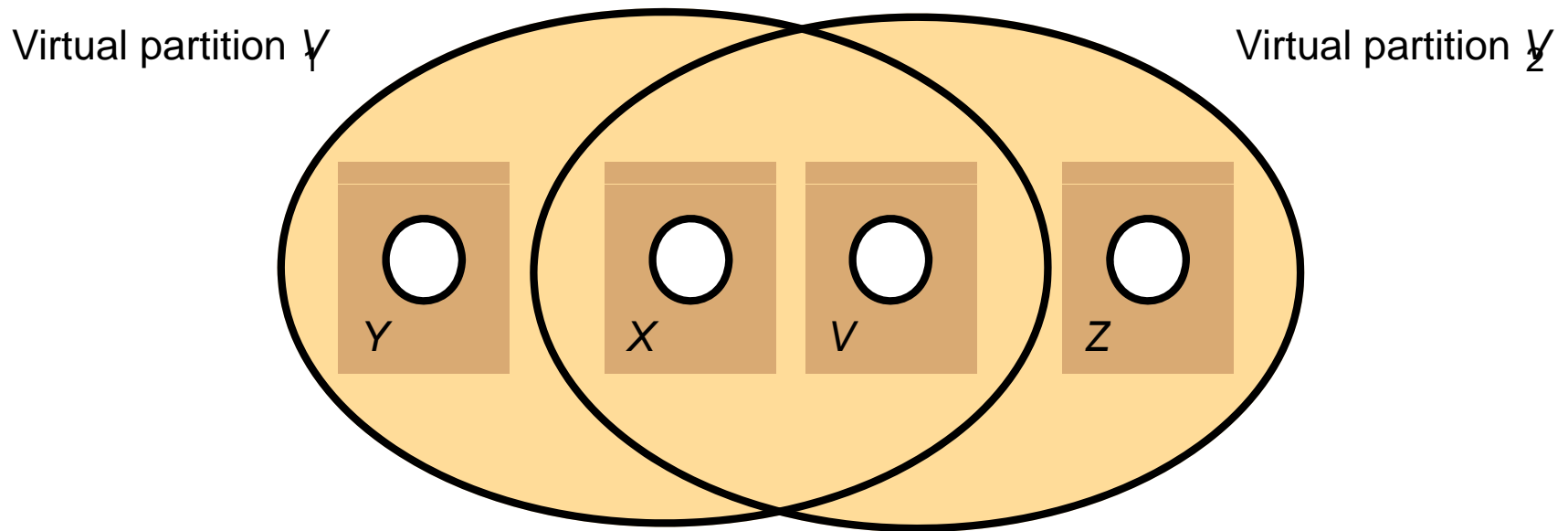
Two network partitions



Virtual partition



Two overlapping virtual partitions



Creating a virtual partition

Phase 1:

- The initiator sends a *Join* request to each potential member. The argument of *Join* is a proposed logical timestamp for the new virtual partition.
- When a replica manager receives a *Join* request, it compares the proposed logical timestamp with that of its current virtual partition.
 - If the proposed logical timestamp is greater it agrees to join and replies *Yes*;
 - If it is less, it refuses to join and replies *No*.

Phase 2:

- If the initiator has received sufficient *Yes* replies to have read and write quora, it may complete the creation of the new virtual partition by sending a *Confirmation* message to the sites that agreed to join. The creation timestamp and list of actual members are sent as arguments.
- Replica managers receiving the *Confirmation* message join the new virtual partition and record its creation timestamp and list of actual members