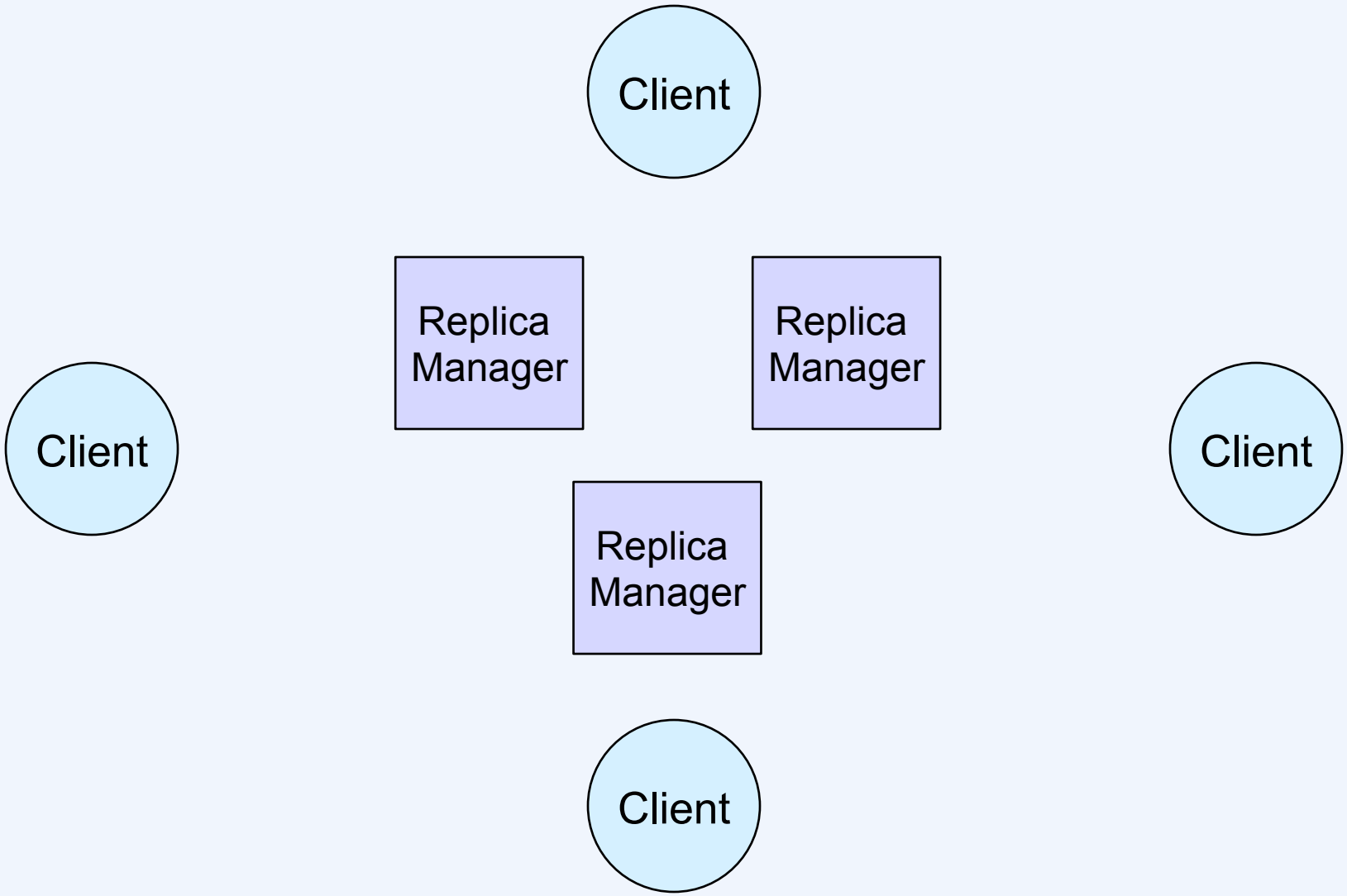
CS 138: Replication and Gossip

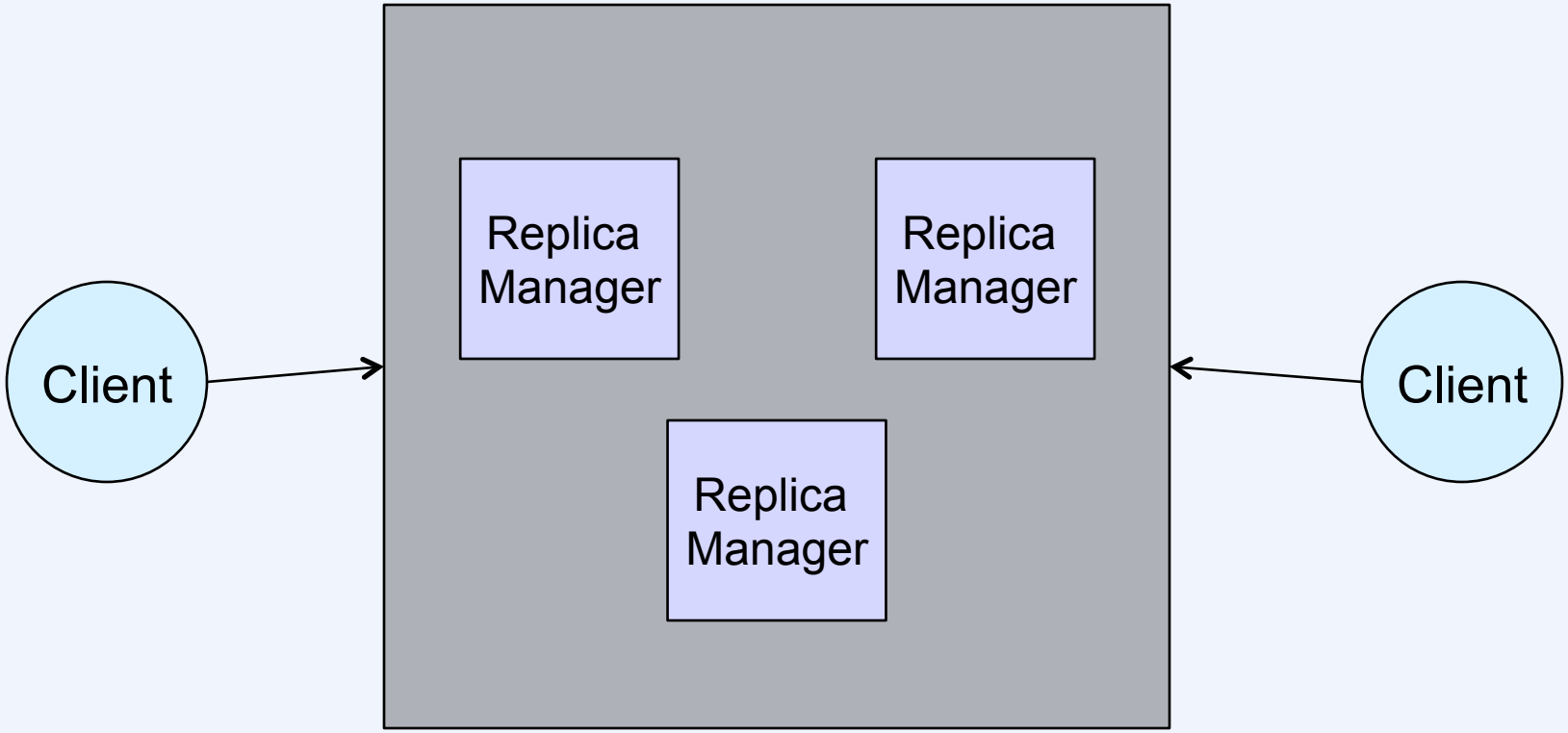
Scenario



Example

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

Linearizability



Linearizability Definition

- Clients perform sequences of operations
  - each operation consists of request, arguments, and result
- A system is *linearizable* iff
  - for any execution of the system, the operations of all the clients can be put into a sequence such that
    - the sequence could have taken place in a system with only one replica manager
    - the operations in the sequence are partially ordered by the real times of their actual occurrences

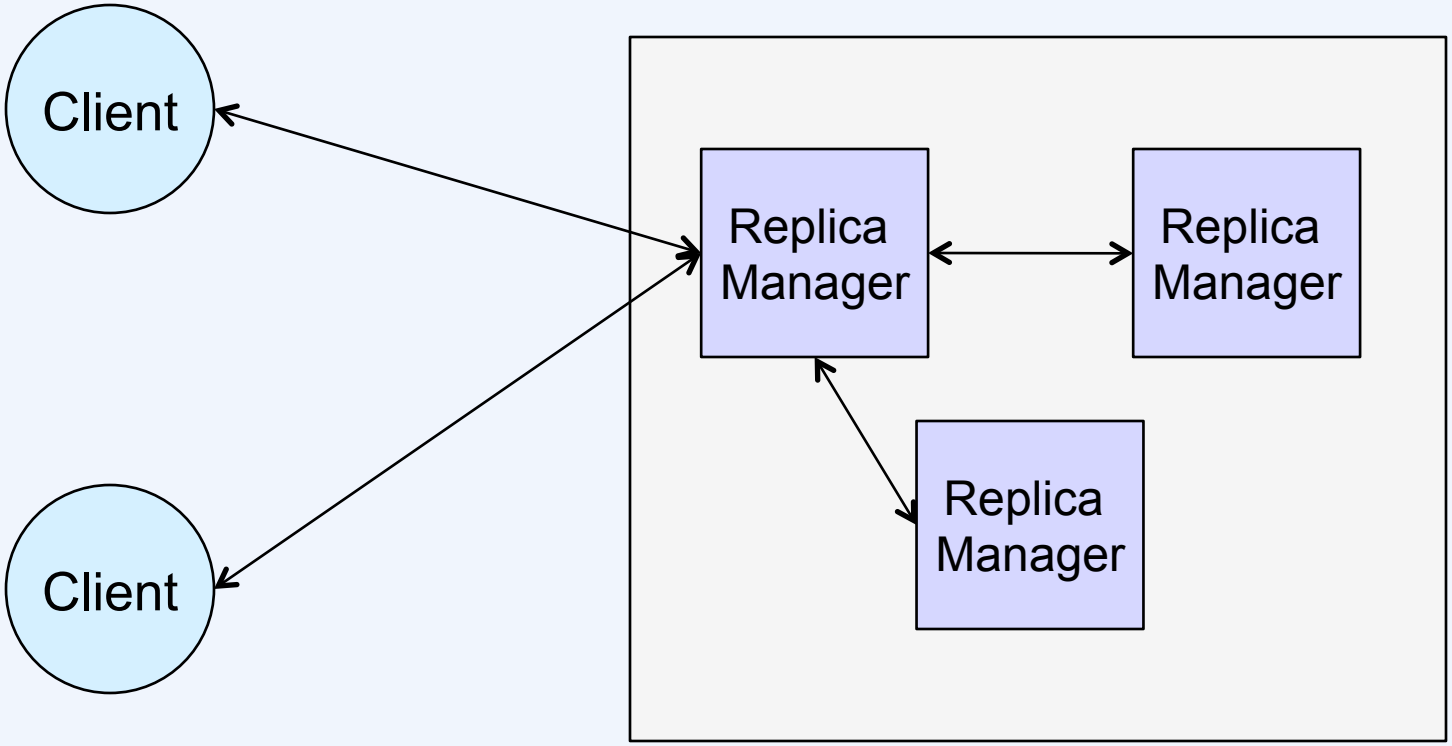
Another Example

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

Sequential Consistency

- Clients perform sequences of operations
  - each operation consists of request, arguments, and result
- A system is *sequential consistent* iff
  - for any execution of the system, the operations of all the clients can be put into a sequence such that
    - the sequence could have taken place in a system with only one replica manager
    - the operations in the sequence are partially ordered by their order in each client

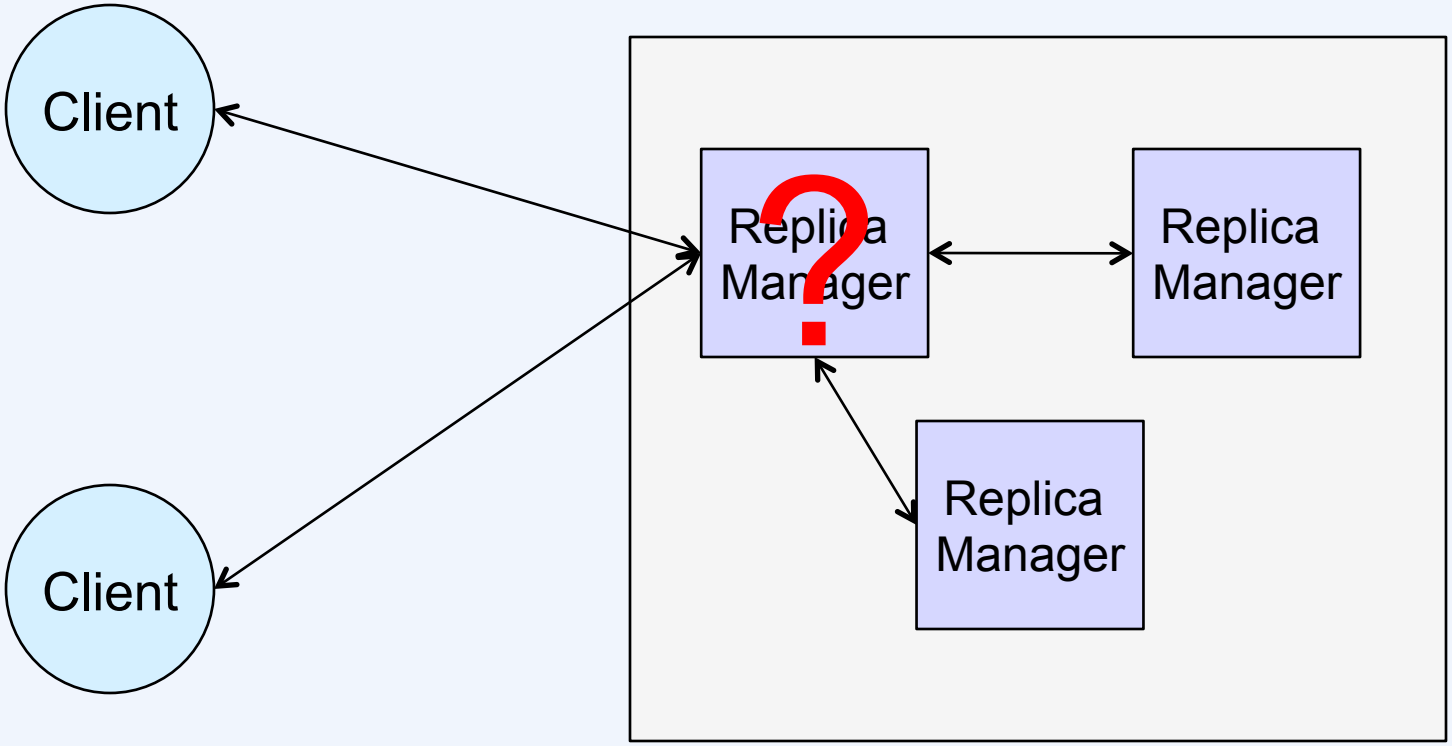
Passive Replication



Passive Replication Sequence

1) Request: client issues request to primary
2) Coordination: primary takes each request atomically, in order received
3) Execution: primary executes request and stores response
4) Agreement: if request is an update, primary sends request to backups
5) Response: once all backups respond, primary sends response to client

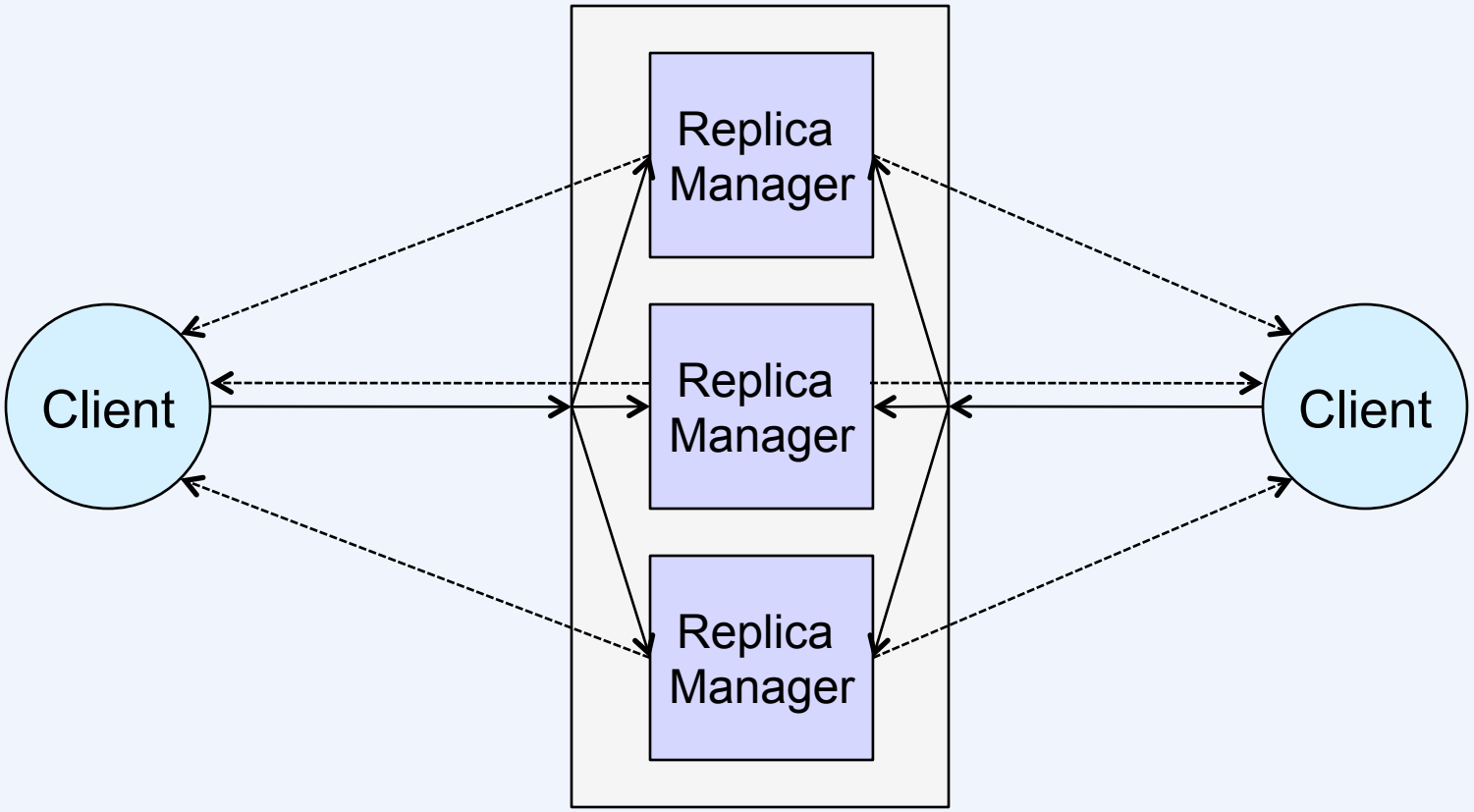
Passive Replication



Raft to the Rescue

- New primary is elected
- Clients communicate with it

Active Replication



Gossip

From “Providing High Availability Using Lazy Replication”
Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat

Scenario

- Distribution-list service
  - multiple, geographically distributed servers
  - replicated database
  - operations
    - post a message
    - add a user
    - ostracize a user

Posting a Message

- Client posts a message
  - contacts nearest server
  - wants quick confirmation
- However …
  - server could crash at any moment
    - message should be replicated at other servers
    - causality constraints must be satisfied
      - but these are pretty weak

Example

- I send a message to Rodrigo
  - via the Providence replica
- Rodrigo reads message
  - via Providence replica
- Rodrigo sends response
  - via Rio de Janeiro replica
- I read response
  - still in Providence (sigh)
- George sends Jake message
  - via San Francisco replica
- Jake reads message
  - via Miami replica
- Jake sends response
  - via Grand Cayman replica
- George reads response
  - via Honolulu replica

Adding a User

- Two different people want to register for the CS138 list as “JCarberry”
- One asks Jordan, the other asks Vivek
- Jordan and Vivek each attempt to add their person
  - simultaneously
- First one to reach the server succeeds, second one fails
- But there are multiple servers
  - each contacts a different server
- Want the same total order at all servers

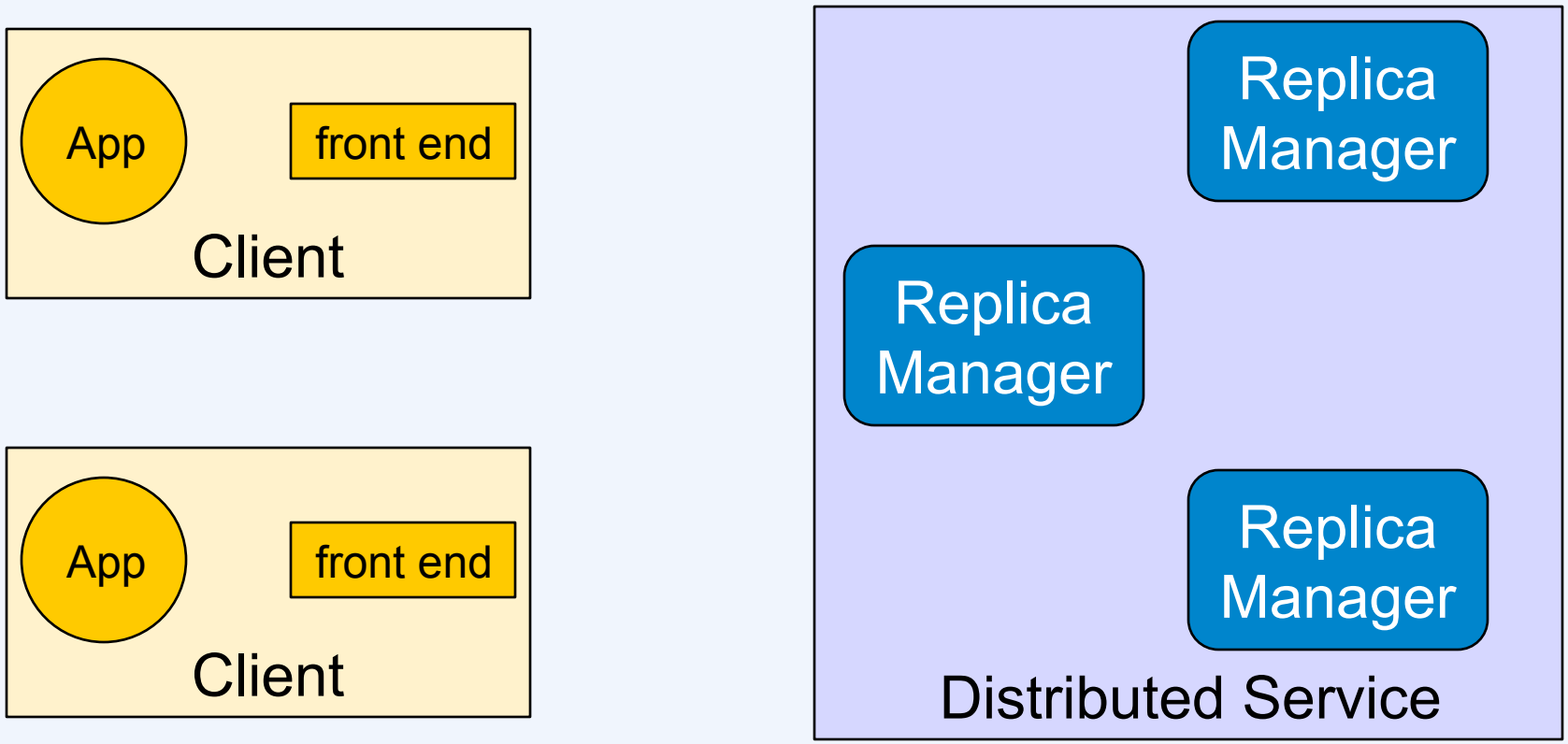
Ostracizing a User

- Junyang defects to CS166
- No longer trusted to receive confidential CS138tas email
- Must be removed from list immediately (if not sooner) at all servers
  - urgent!

Desired Features

- Causal ordering
  - needed for updates
- Forced ordering
  - both causal and total order
  - needed for adding JCarberry
- Immediate ordering
  - forced ordering with minimal delay
  - needed for ostracizing Jake

Model

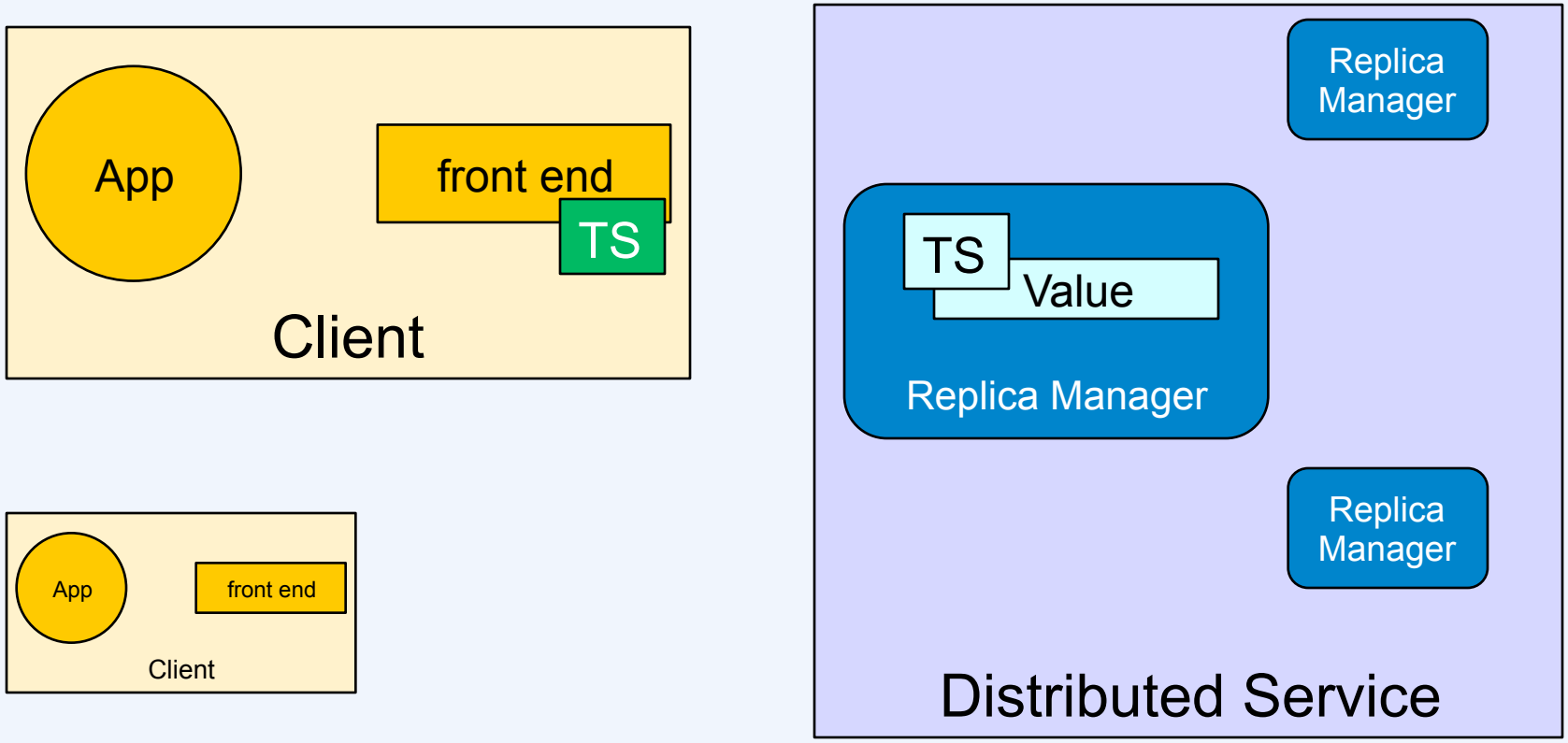


- **Clients normally communicate with one RM**
  - **but it might be busy**
    - **communicate with another**
    - **communicate with many**

Rough Outline
(Causal Ordering)

- Query
  - client sends request to one or more RMs
  - respond when causally possible
- Update
  - client sends request to one or more RMs
  - update and respond when causally possible
  - propagate changes to others via “gossip” messages
    - not specified how this is done
    - allows many possibilities

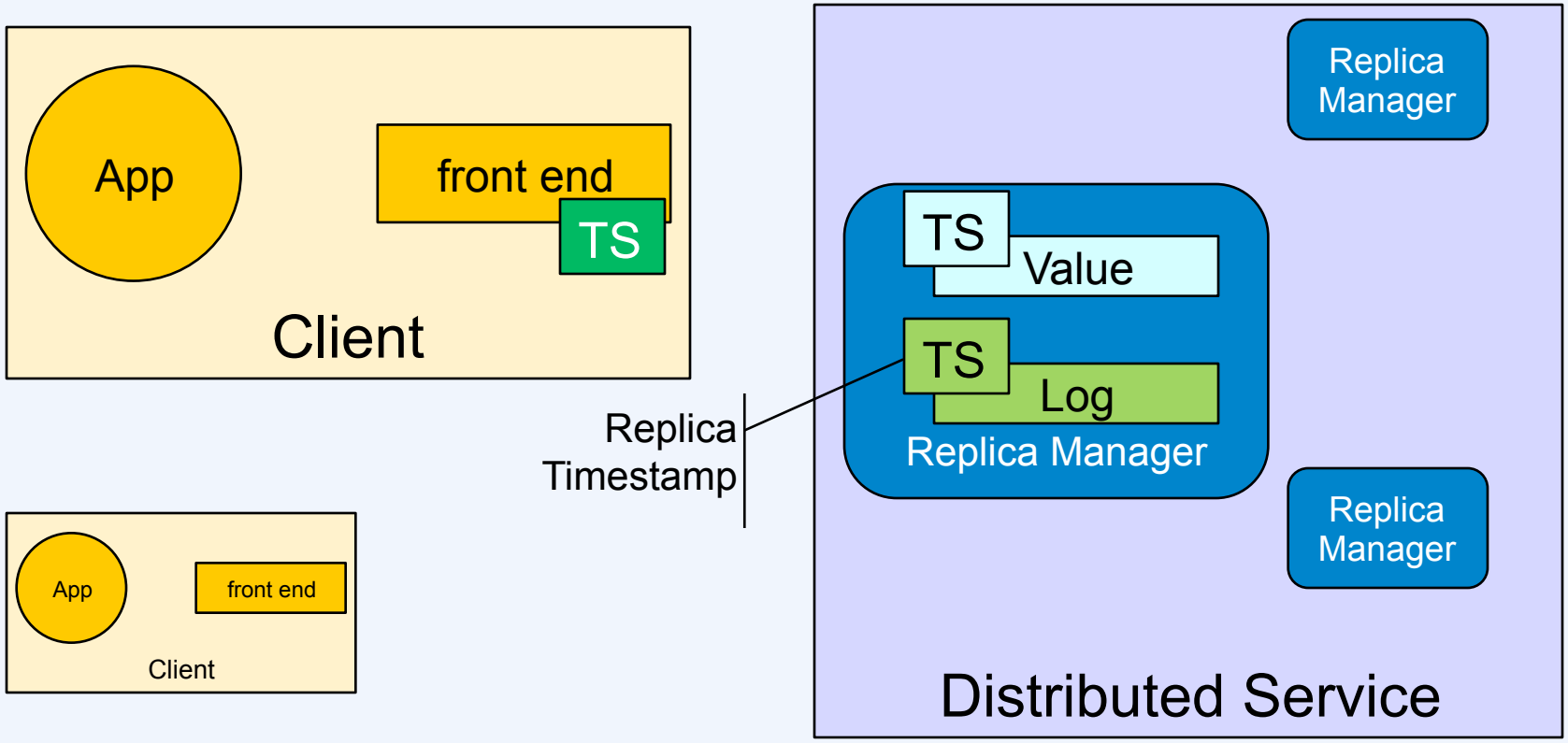
Query Model



Query

- Client sends query q
  - request (q.op)
  - causal dependencies
    - q.prev = client.ts
- RM i responds
  - receives query
  - holds it until
    - $q.prev \leq rm_i.val.ts$
  - returns value and timestamp (rm.val.ts)
- Client
  - updates its own timestamp
    - client.ts = merge (client.ts, rm.val.ts)

Update Model



Update (1)

- Client sends
  - request (u.op)
  - causal dependencies
    - u.prev = client.ts

Update (2)

- RM i responds
  - receives update
    - assigns timestamp
      - rm_i.replica.ts[i] += 1
      - TS = u.prev; TS[i] = rm_i.replica.ts[i]
    - puts in log
      - <u, i, TS> (update, node, timestamp)
    - returns TS
    - when u.prev ≤ rm_i.val.ts
      - updates val (by applying u.op)
      - rm_i.val.ts = merge(rm_i.val.ts, TS)

Update (3)

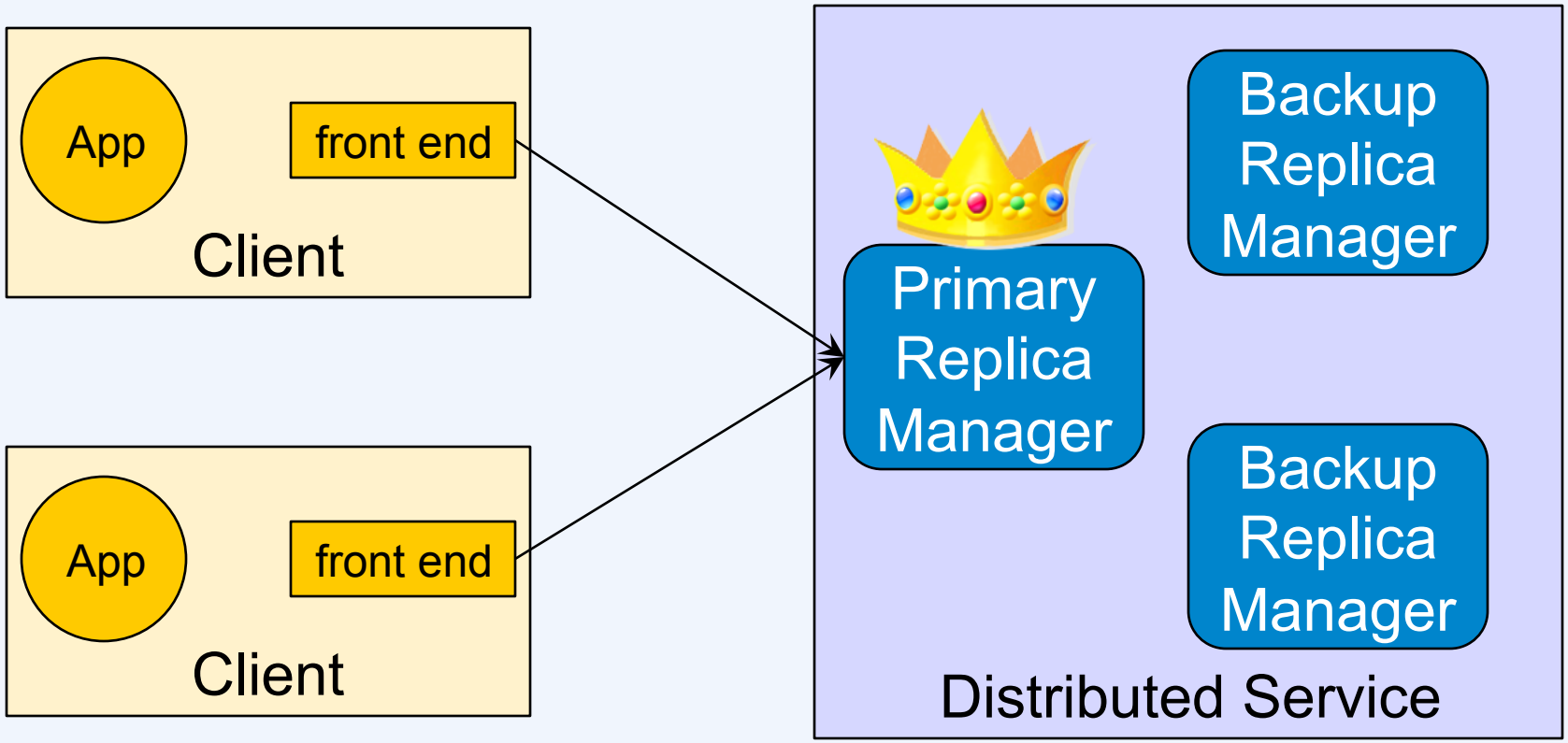
- Client
  - updates its own timestamp
    - client.ts = merge (client.ts, TS)

Gossipping

- RM a initiates gossip
  - sends to RM b:
    - contents of log (rm_a.log)
    - replica timestamp (rm_a.replica.ts)
- RM b receives gossip
  - merges rm_a.log into rm_b.log
  - rm_b.replica.ts = merge(rm_a.replica.ts, rm_b.replica.ts)
  - while there exists request r in rm_b.log such that r.u.prev ≤ rm_b.val.ts && r.processed == false
    - r.processed = true
    - update val (by applying r.u.op)
    - rm_b.val.ts = merge(rm_b.val.ts, r.TS)

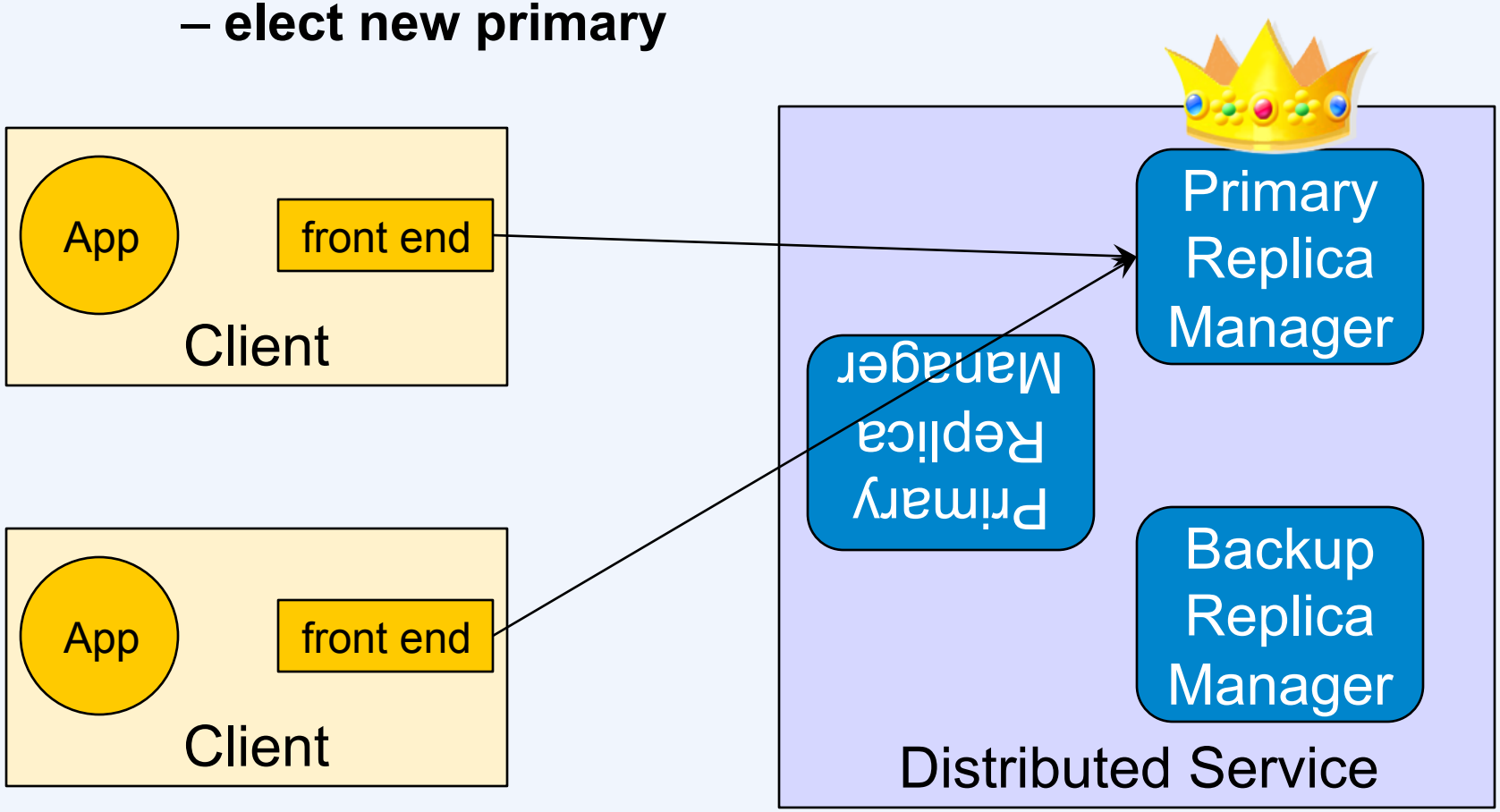
Forced Updates (1)

- **Need a causal order that's also total**
  - **– all clients go through same RM**

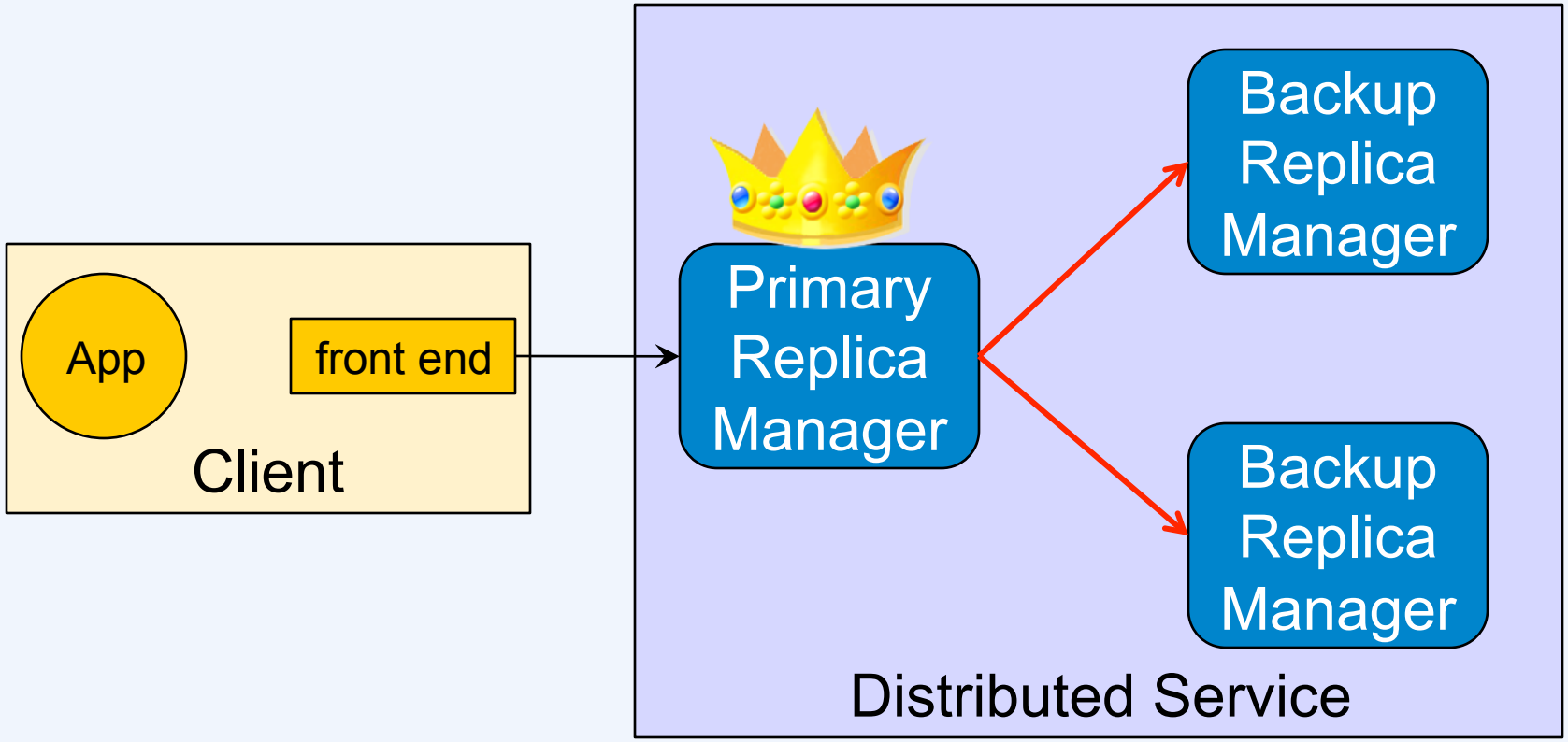


Forced Updates (2)

- **What if primary crashes?**
  - **– elect new primary**

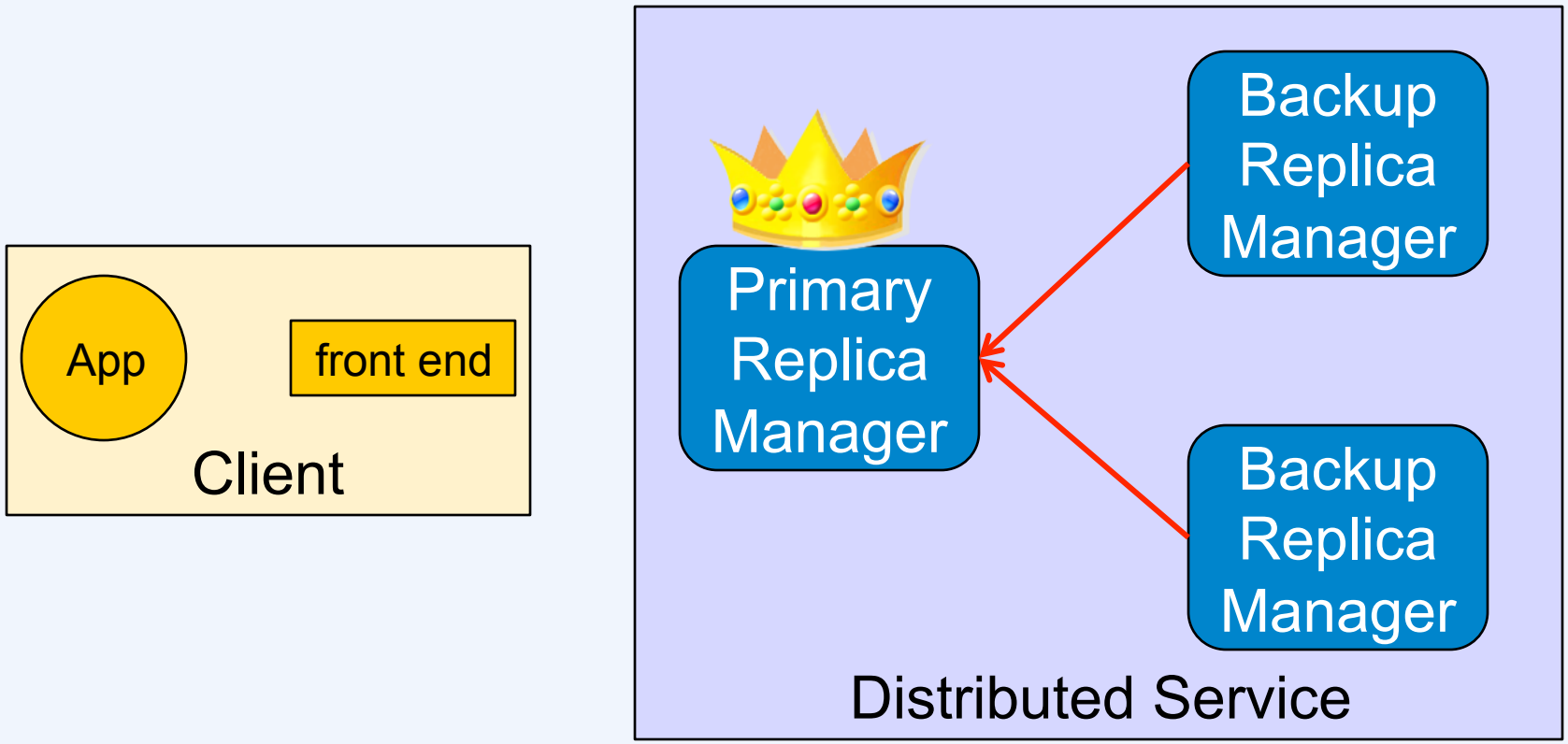


Immediate Updates (1)



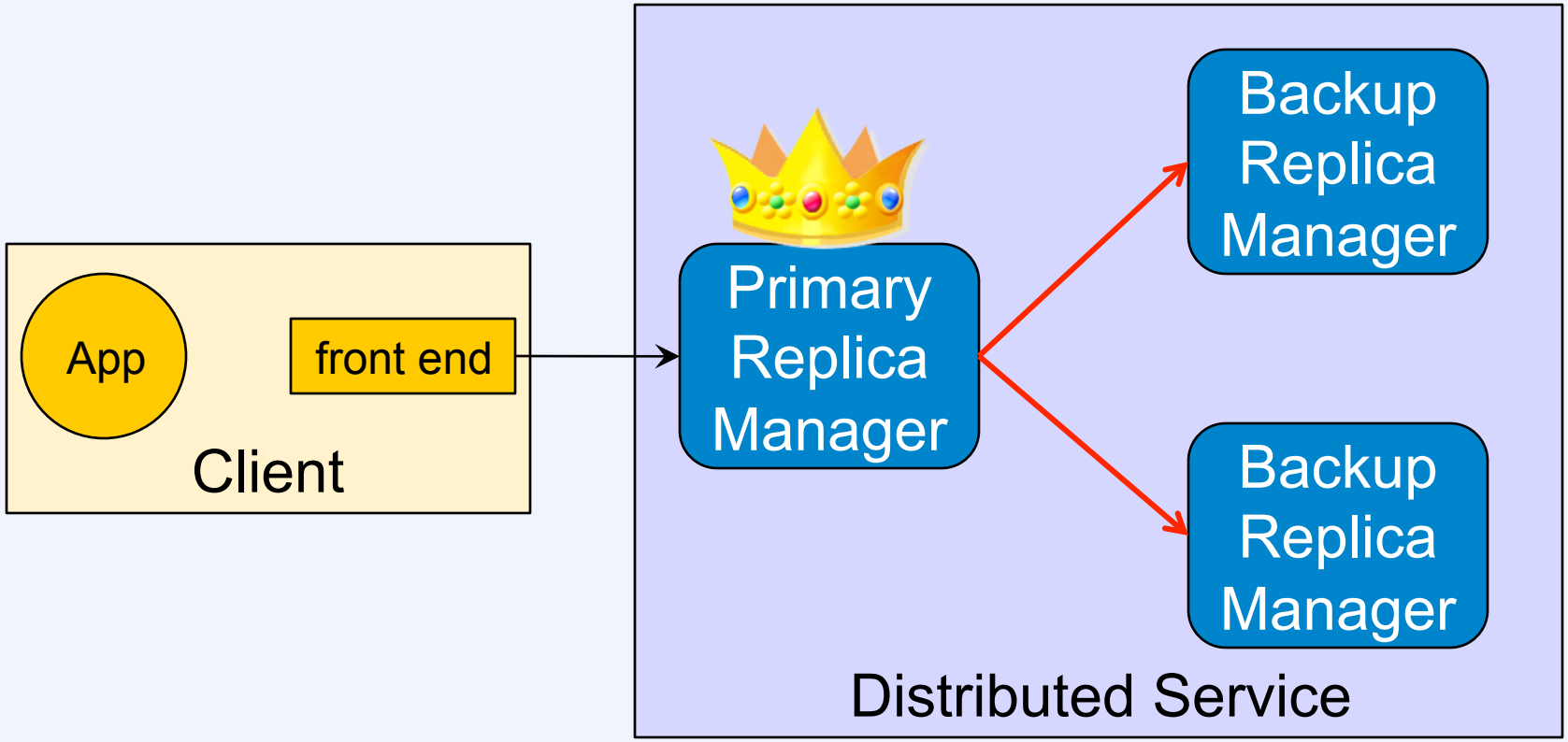
- **Primary requests logs and replica timestamps**
  - **backups respond and stop processing queries**
  - **updates are accepted but not executed**

Immediate Updates (2)



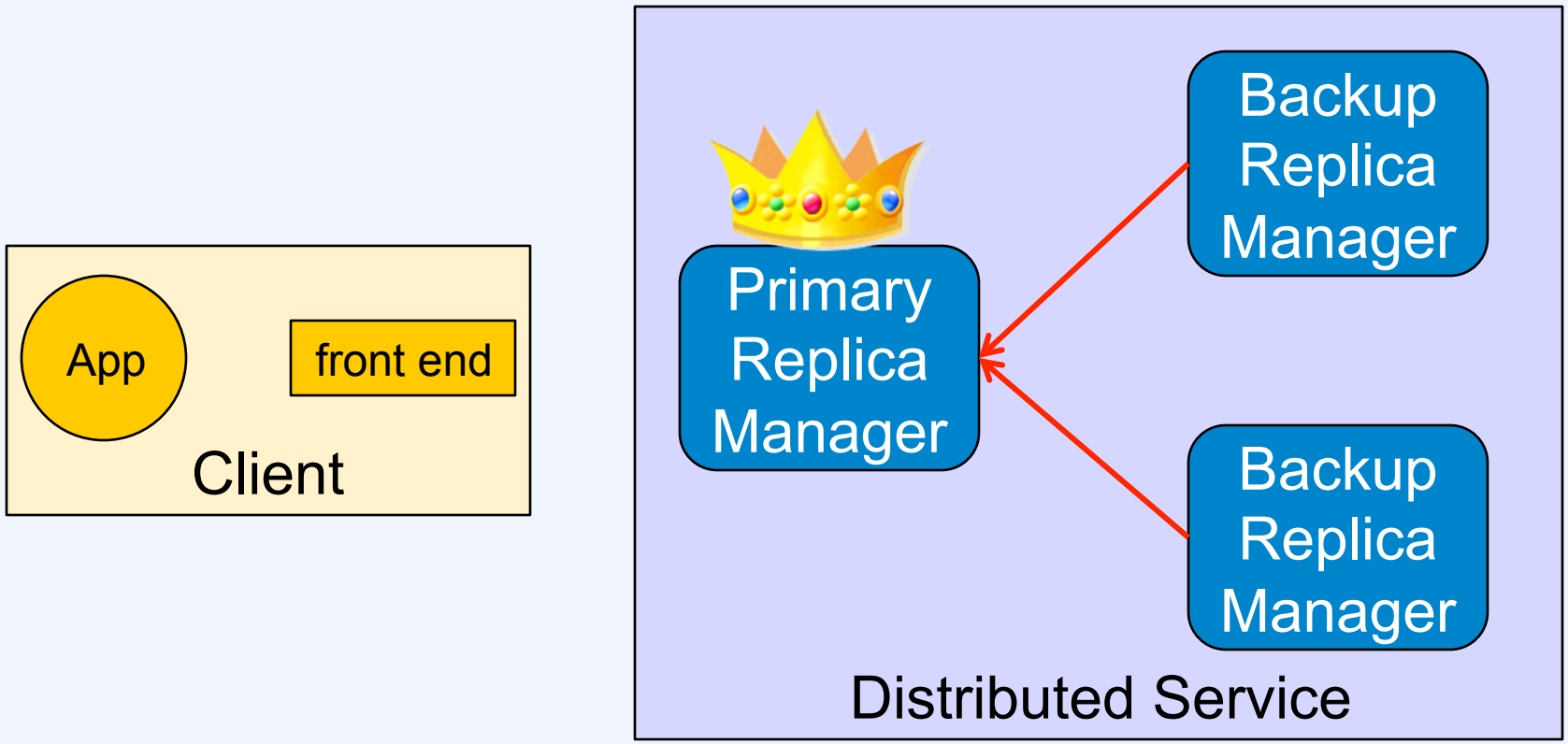
- **Backups respond with logs and timestamps**
  - **primary stops processing queries and updates**
  - **processes logs and timestamps**

Immediate Updates (3)



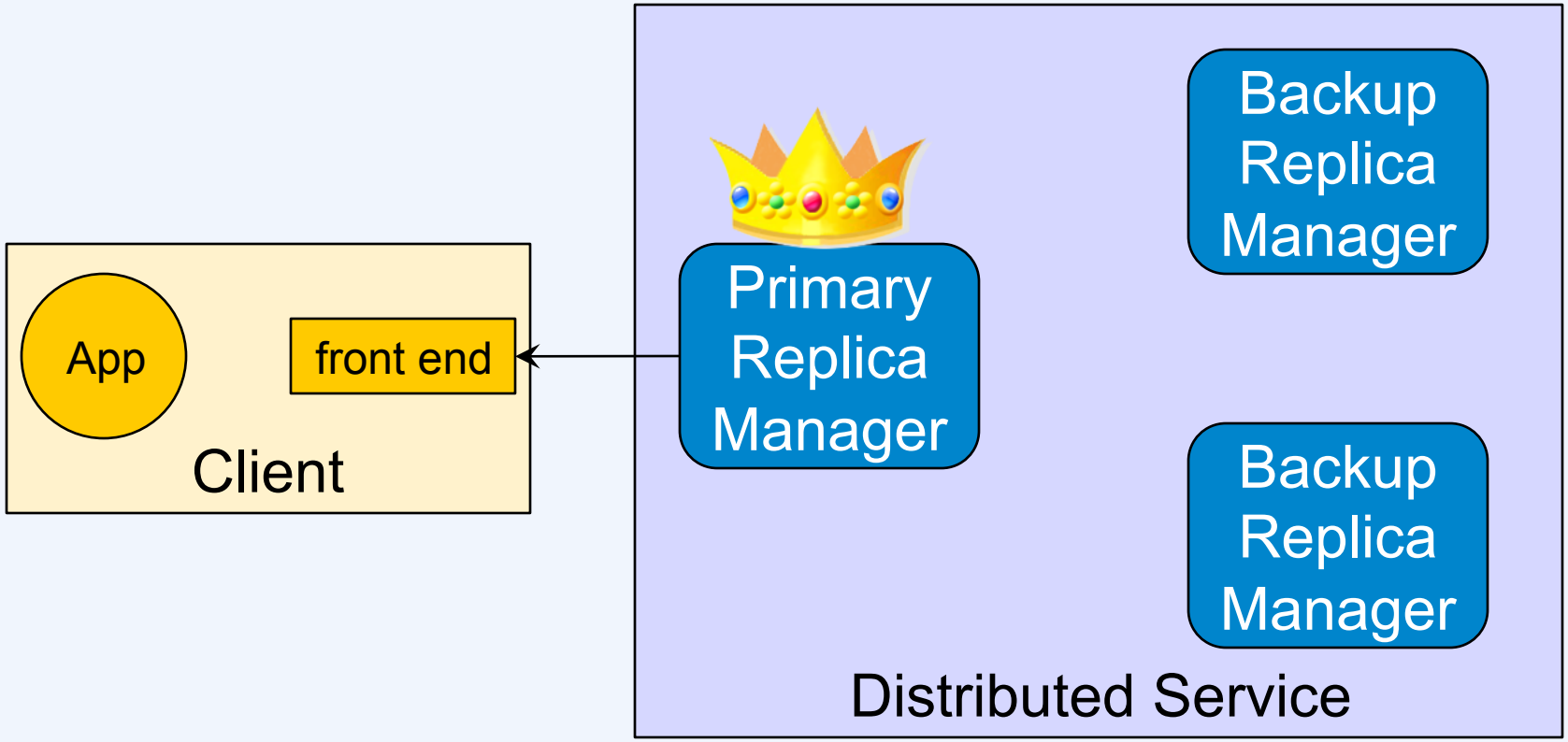
- **Primary assigns timestamp to update**
- **Primary sends update to backups**

Immediate Updates (4)



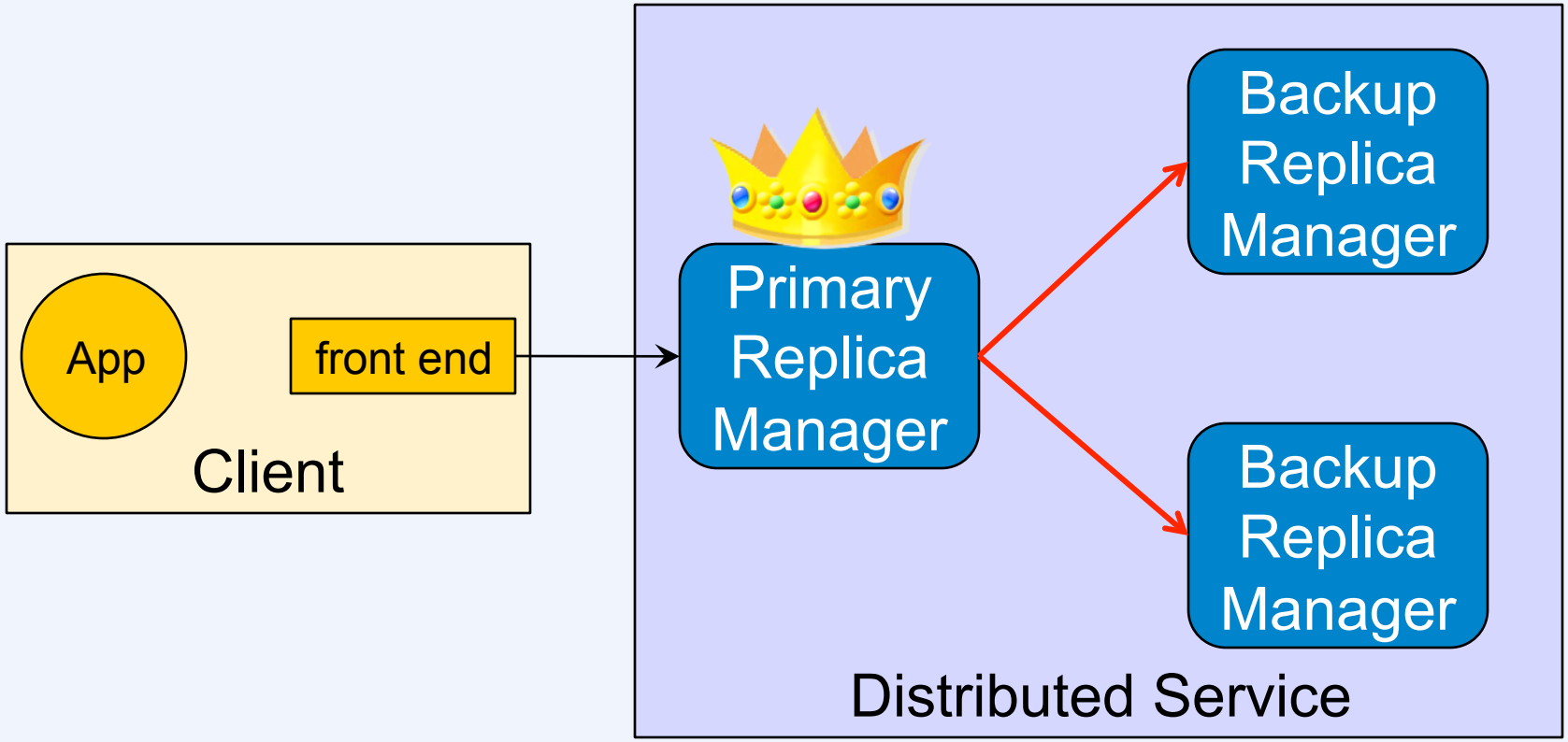
- **Backups acknowledge updates**

Immediate Updates (5)



- **After half the backups respond, primary commits (updates val) and responds to client**
  - **half the backups + primary = majority**

Immediate Updates (6)



- **Primary sends log to each backup (gossips)**

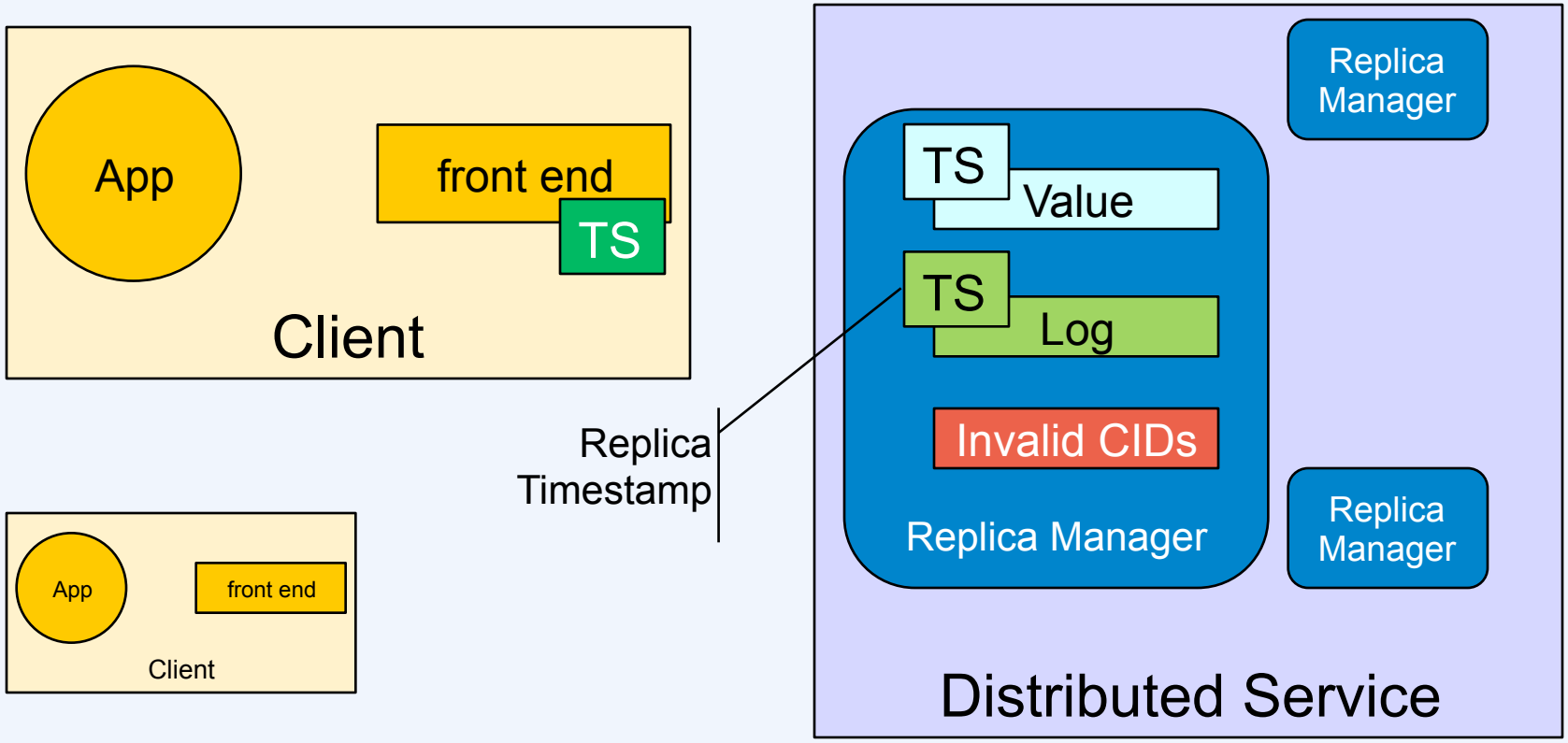
Problem?

- What if client sends update request to multiple RMs?
  - multiple copies of the request are propagated
  - all are executed
  - probably aren't idempotent

Solution

- Client assigns unique ID (CID) to each request
- RMs keep track of CIDs of completed requests
  - completed requests go to *invalid CIDs* list
  - check list before doing a request
    - don't perform requests that have already been performed

Updated Update Model



Another Problem?

- Won't logs and invalid CIDs lists grow without bound?
  - yes …

Bounding Logs (1)

- Each log entry r must be kept on RM i until it is present on all RMs
  - so that gossip from i will inform other RMs
- r.node is the node that created the log entry r
- r.ts is the vector timestamp assigned to the log entry by r.node
- r.ts[r.node] is the logical time on r.node when the entry was created
- r may be removed from i's log when:
  - $\forall k$: r.ts[r.node] $\leq rm_k$.replica.ts[r.node]

Bounding Logs (2)

- How does RM i know rm_k.replica.ts[r.node]?
  - gossip messages contain replica timestamps
    - timestamps on logs
  - each RM keeps a table of the most recent timestamps obtained from all other RMs
    - rm_i.ts_table
    - $\forall k\ rm_i.ts\_table[k] \leq rm_k.replica.ts$
  - RM i may remove log entry r when:
    - $\forall k$: r.ts[r.node] $\leq rm_i.ts\_table[k][r.node]$

Trimming the Invalid CIDs List

- When can an entry be removed?
  - when it will never be received again
- Assuming perfect communication, how can you tell?
  - you can't: client's front-end might send same update to multiple RMs
  - what's more, communication might not be perfect
- More machinery needed …

More Machinery …

- Client front-end puts (real-time) timestamps on all update requests
- After successful transmission of last transmission of an update, it sends "that's all" (TA) message to at least one RM
  - contains CID of update and (real-time) timestamp
    - timestamp of TA is later than that of updates
  - RM puts it in log (and includes it in gossips)
- Assume maximum real time required for any RM i to notify RM j of new info via gossip is δ
  - takes into account clock skew, etc.

Yet More Machinery …

- General idea
  - all equivalent update messages terminated by TA *a* must be received by *a*.timestamp + δ
- Details
  - discard CID *c* from invalid CID list if its TA is in log and no update records for *c* in log
    - all RMs have seen *c*
  - ignore updates if m.time + δ < replica's local time
  - discard TA *a* from log if it appears in all logs and *a*.timestamp + δ < replica's local time
    - no other instances of updates terminated by *a* are still circulating