Distributed Systems

ID2201

replication

Johan Montelius

The problem

- The problem we have:
  - servers might be unavailable
- The solution:
  - keep duplicates at different servers

Building a fault-tolerant service

- When building a fault-tolerant service with replicated data and functionality the service:
  - should produce the same results as a non-replicated service
  - should respond despite node crashes
  - ... if the cost is not too high

What is a correct behavior

- A replicated service is said to be *linearizable* if for *any execution* there is *some interleaving* that ...
  - meets the specification of a non-replicated service
  - matches the *real time* order of operations in the real execution

A less restricted

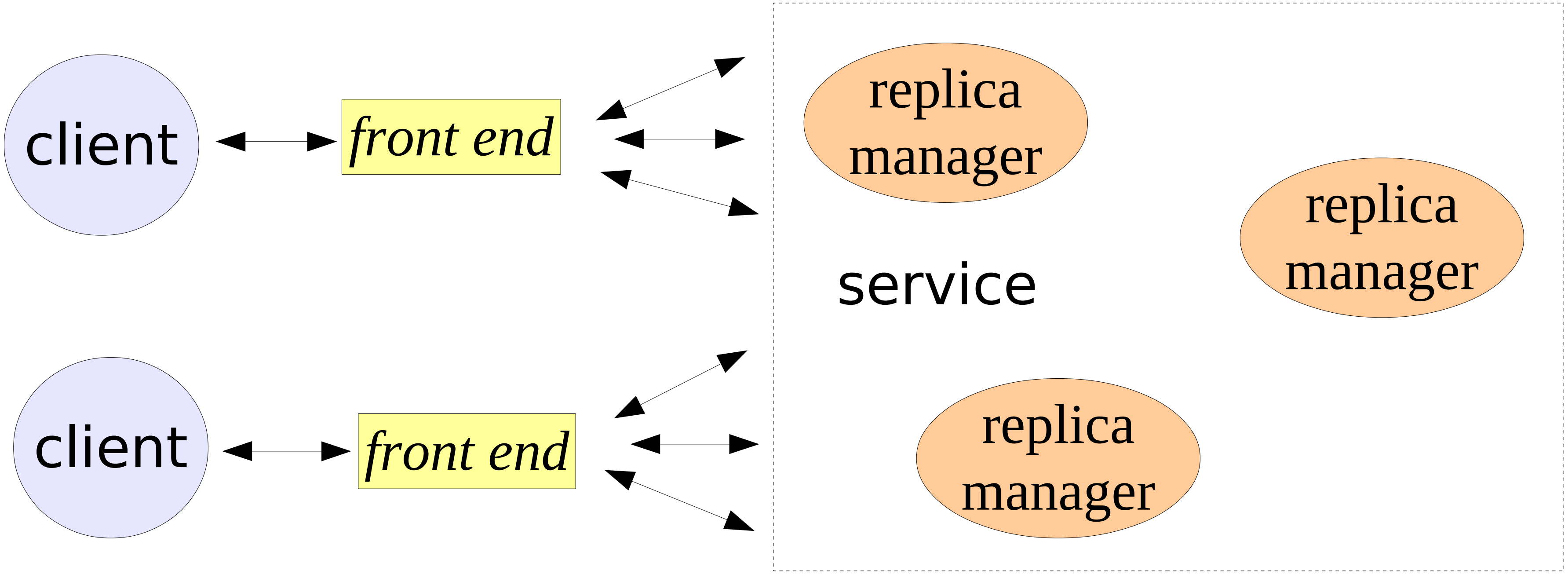
- A replicated service is said to be *sequentially consistent* if for any *execution* there is *some interleaving* that ...
  - meets the specification of a non-replicated service
  - matches the *program order* of operations in the real execution

even less restricted

- Eventual consistency
  - sooner or later, but until then?
- Causal consistency
  - if a caused b and you read b then you should be able to read a
- Read your writes
  - at least during a session?
- Monotonic reads
  - always read something new

System model

- Asynchronous system, nodes fail only by crashing.



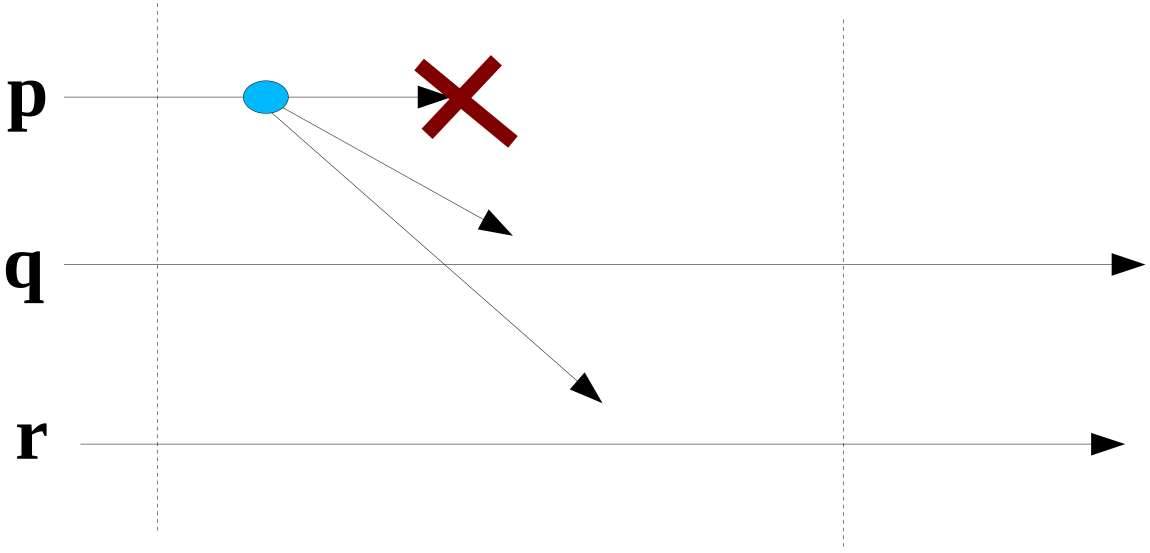
Group membership service

- A dynamic group:
  - nodes can be added or removed
  - needs to be done in a controlled way
  - system might halt until the group is updated
- A static group:
  - if a node crashes it will be unavailable until it is restarted
  - should continue to operate even if some nodes are down

View synchrony

- A group is monitored and if a node is suspected to have crashed, a new view is *delivered*.
- Communication is restricted to *within a view*.
- Inside *a view,* we can implement leader election, atomic multicast etc.

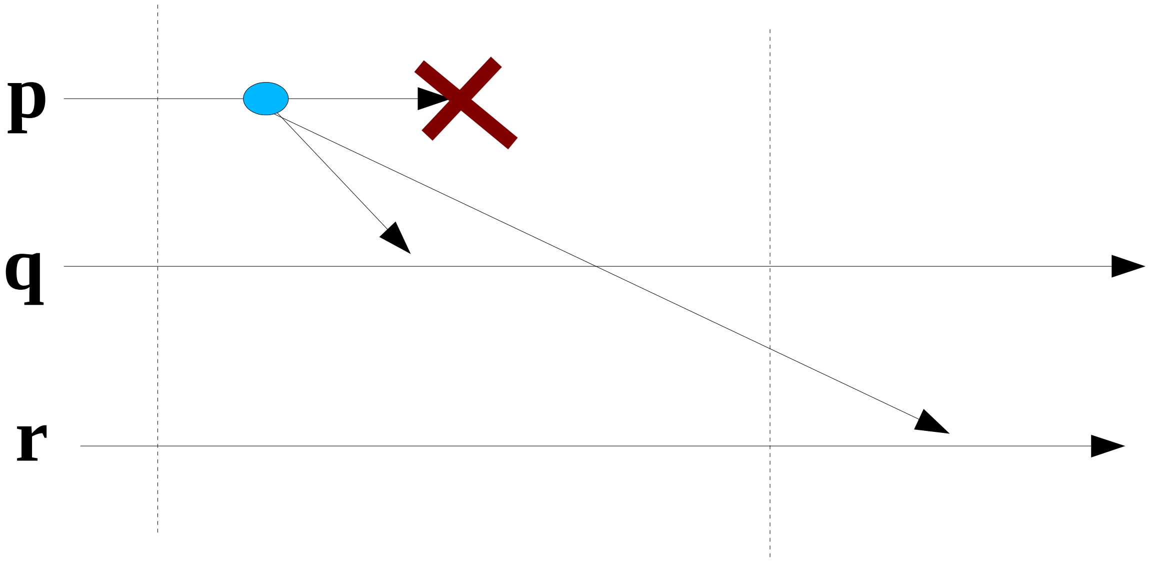
view-synchronous communication



v(g) = {p,q,r}

v(g) = {q,r}

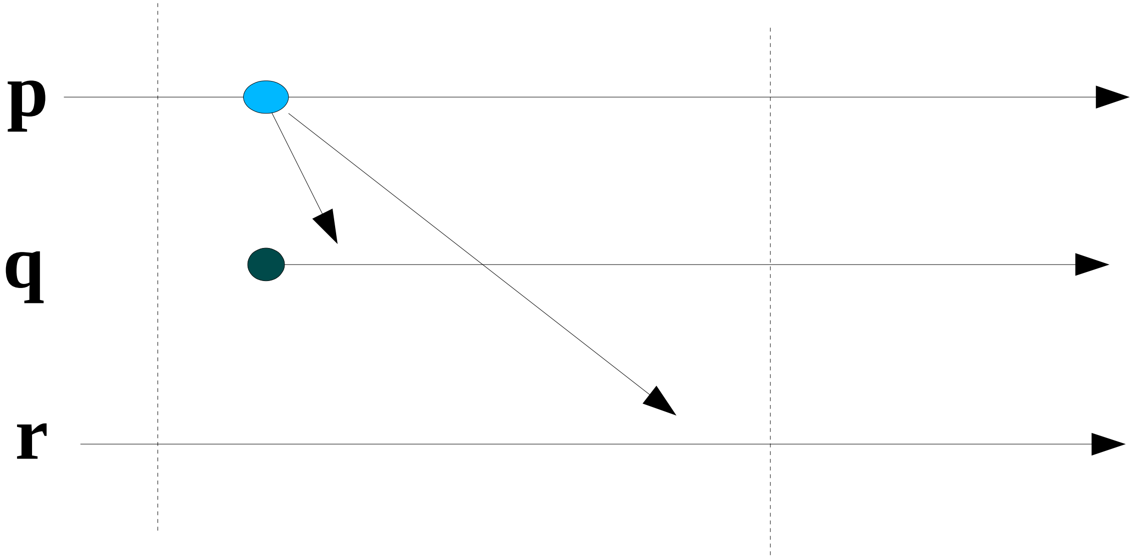
disallowed



v(g) = {p,q,r}

v(g) = {q,r}

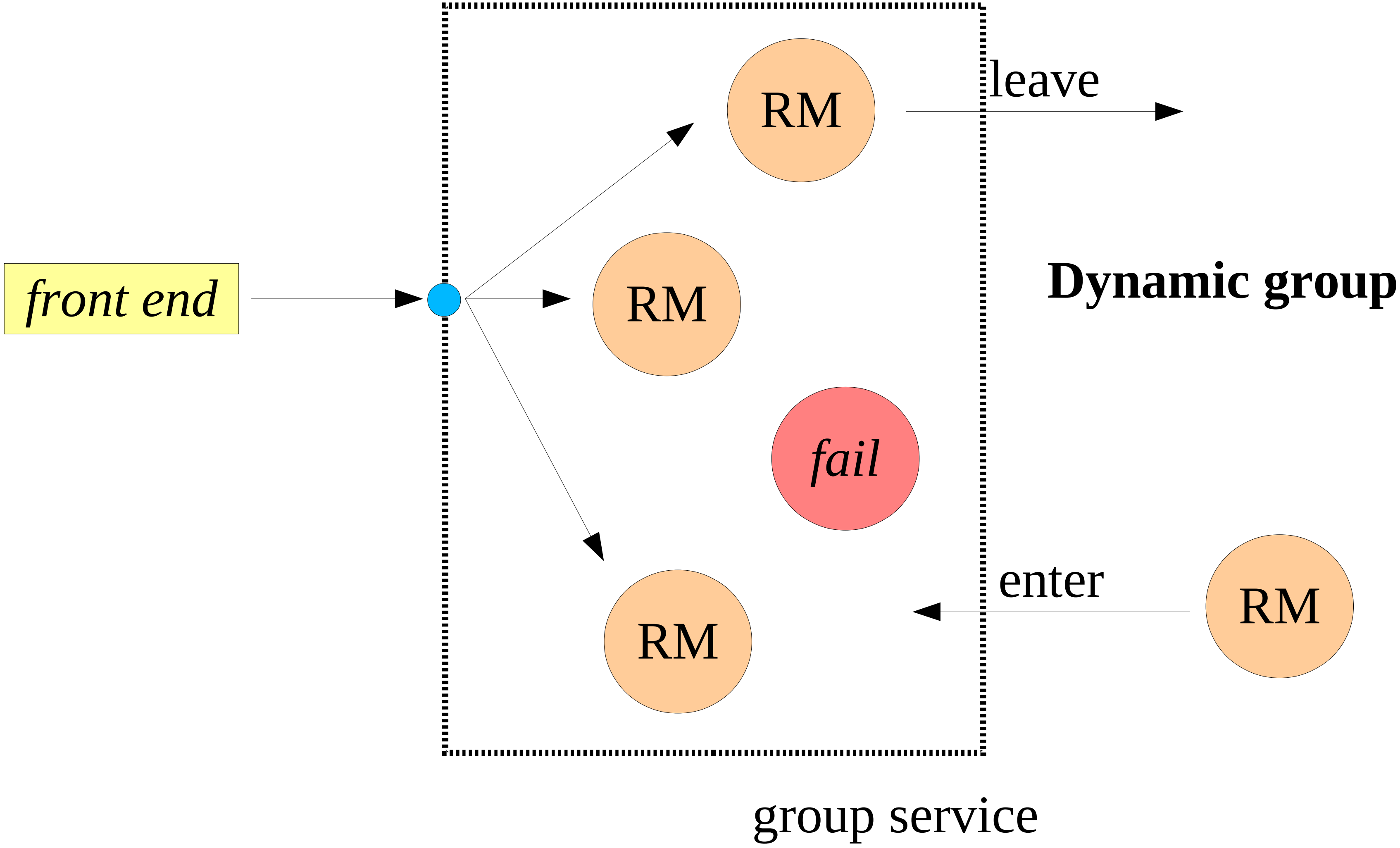
disallowed



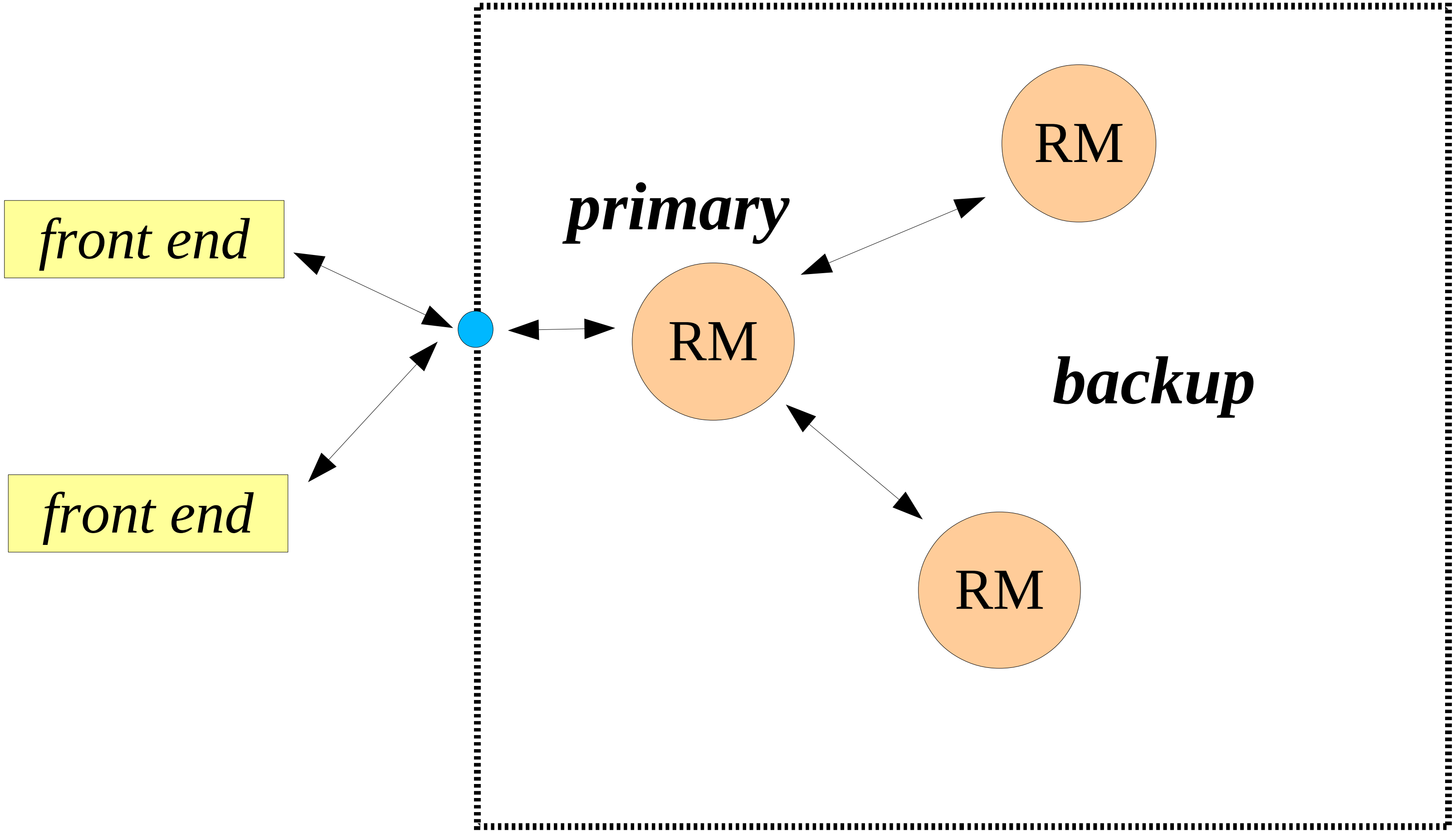
v(g) = {p,r}

v(g) = {p,q,r}

Group membership service



passive replication



passive replication

- Request
  - request with a unique identifier
- Coordination
  - primary checks if it is a new request
- Execution
  - primary execute, and store response
- Agreement
  - send updated state and reply to all backup nodes
- Respond
  - send reply to front-end

Is it linearizable?

- The primary replica manager will serialize all operations.
- If the primary fails, it retains linearizability if backup takes over where the primary left of.

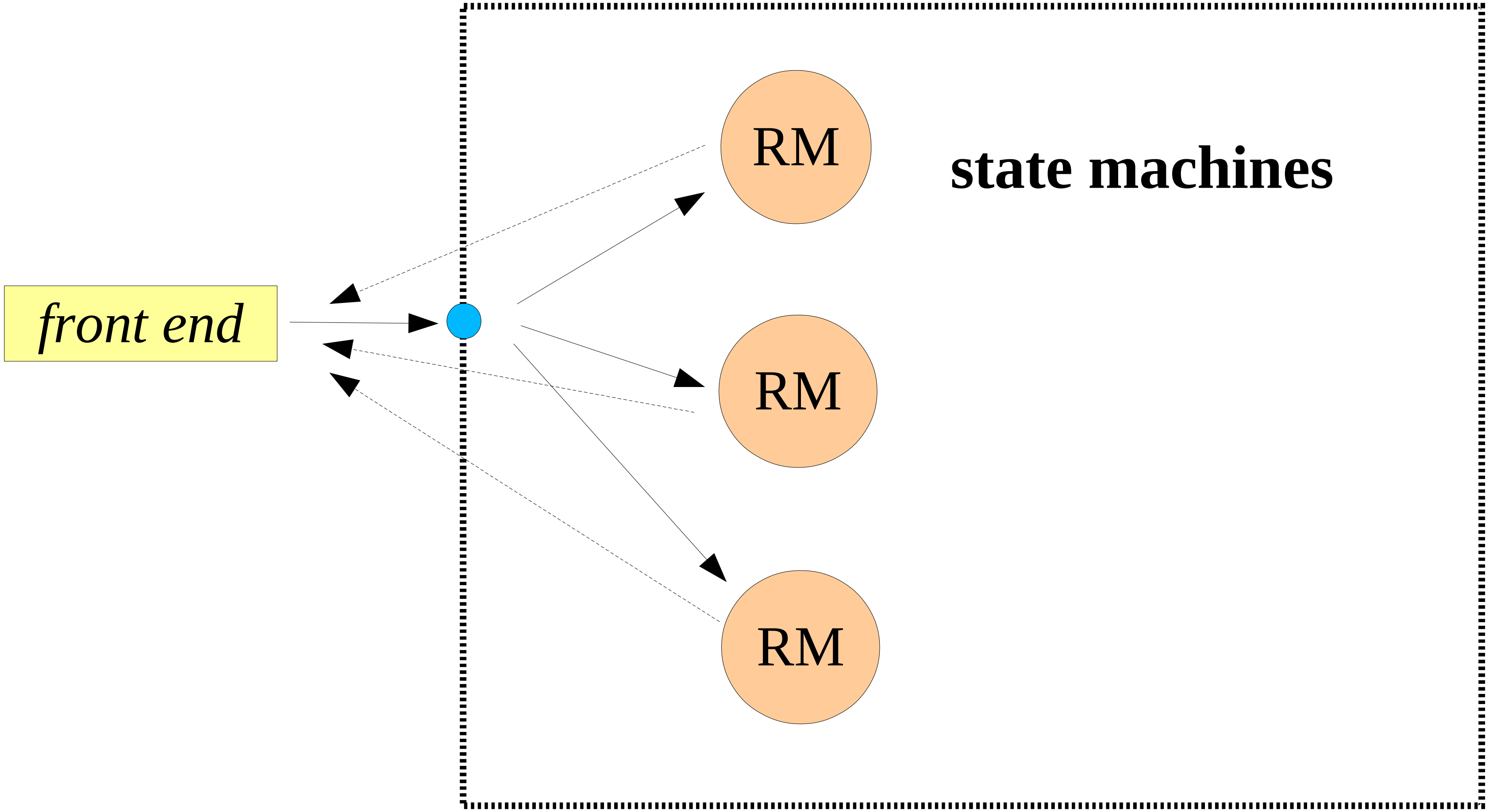
primary crash

- Primary crash:
  - backups will receive *new view* with primary missing
  - new primary is elected
- Request is resent:
  - if agreement was reached last time, the reply is known and is resent
  - if not, the execution is redone

Pros and cons

- Pros
  - All operations passes through a primary that linerarize operations.
  - Works even if execution is in-deterministic
- Cons
  - delivering state change can be costly

active replication



Active replication

- Request
  - multicasted to group, unique identifier
- Coordination
  - deliver request in ***total order***
- Execution
  - all replicas are identical and deterministic
- Agreement
  - not needed
- Response
  - sent to front end, first reply to client

Active replication

- Sequential consistency:
  - All replicas execute the same sequence of operations.
  - All replicas produce the same answer.
- Linearizability:
  - Total order multicast does not (automatically) preserve real-time order.

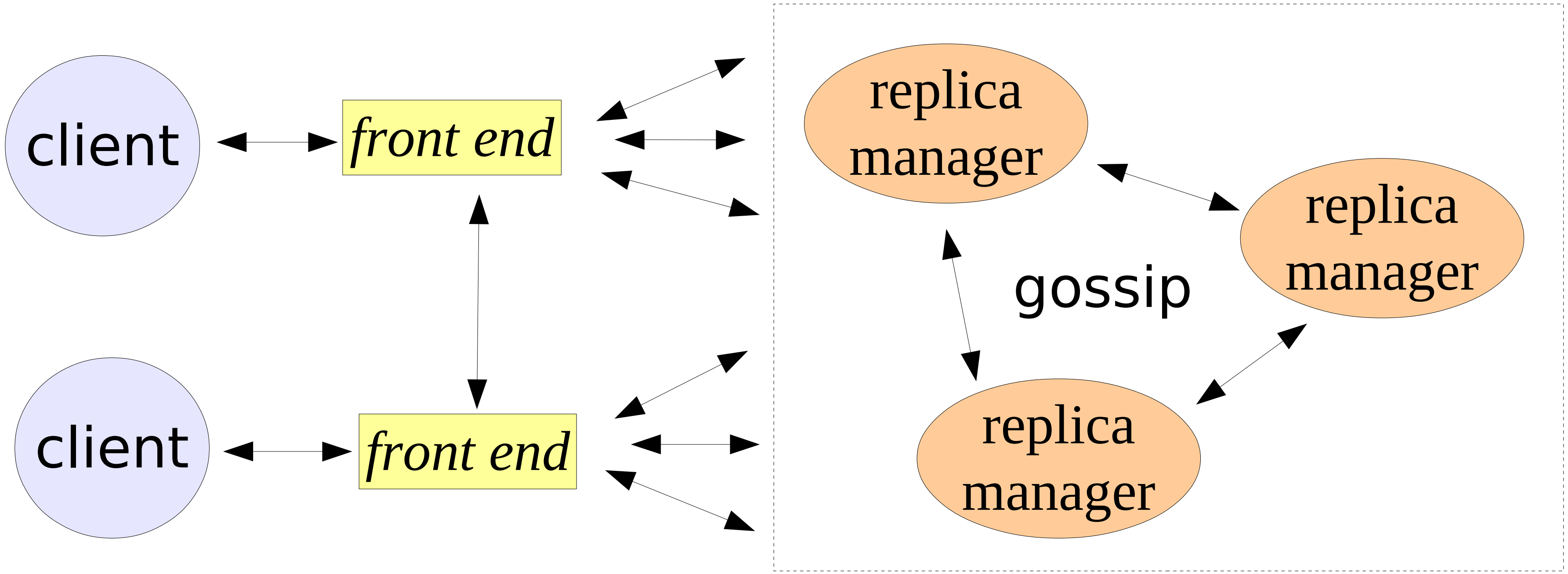
Pros and cons

- Pros
  - no need to change existing servers
  - no need to send state changes
  - could survive Byzantine failures
- Cons
  - requires total order multicast
  - requires deterministic execution

High availability

- Both replication schemes require that servers are available.
- If a server crashes it will take some time to detect and remove the faulty node.
  - depends on network
  - is this acceptable
- Can we build a system that responds even if all nodes are not available?

Gossip architecture



Relaxed consistency

- Increase availability at the expense of consistency.
  - causal update ordering
  - forced (total and causal) update ordering
  - immediate update ordering (total order with respect to all other updates)

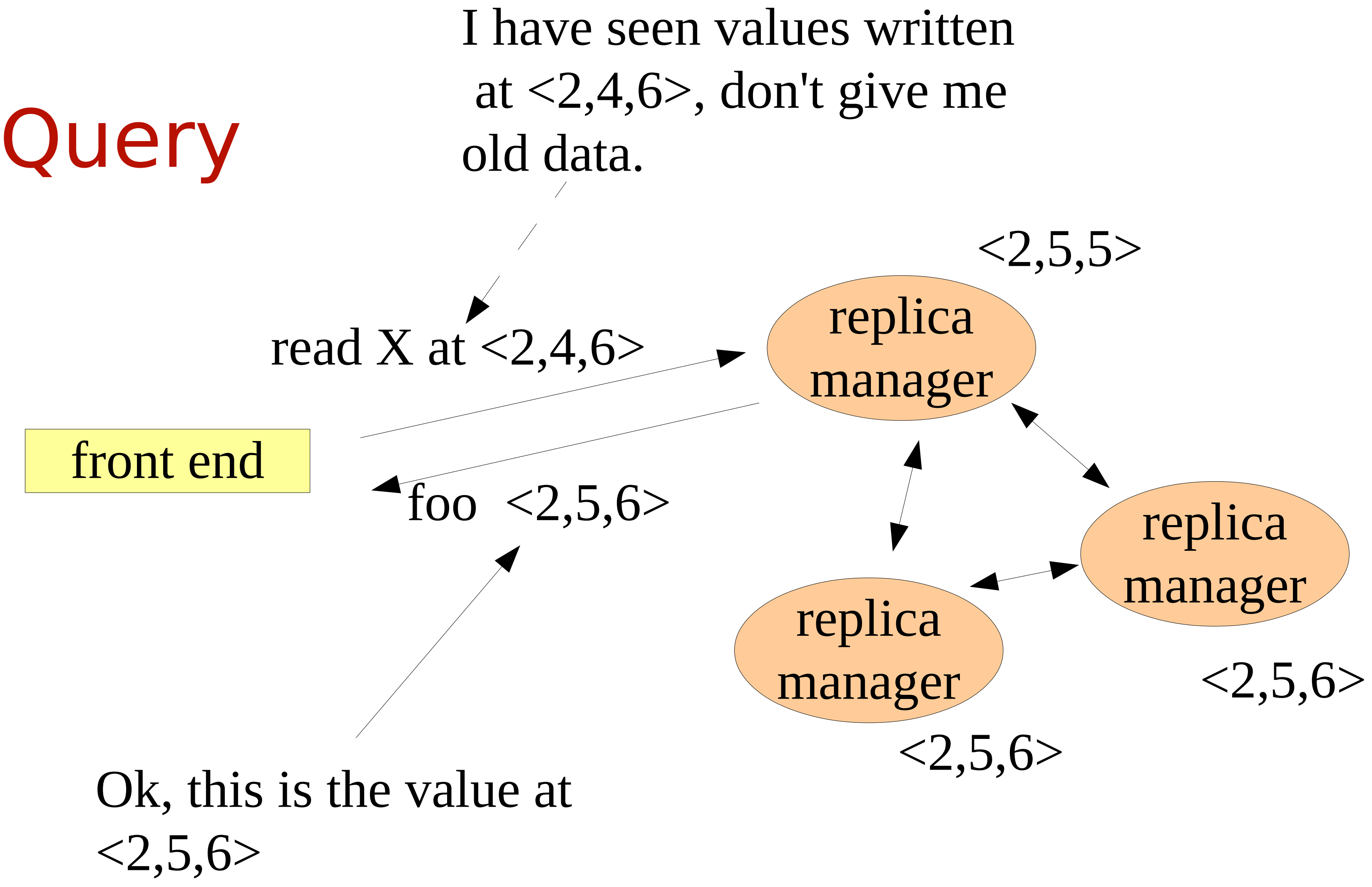
Example: bulletin board

- Adding messages:
  - causal ordering
- Adding a user:
  - forced ordering
- Removing a user:
  - immediate ordering
    - All replicas should agree on what messages are before the removal of the user.

Implementation

- Front ends keep a vector clock that reflects the last seen value.
- The vector holds an entry for each replica in the system.
- The vector clock is updated as the front end sees replies from the replicas.
- The front end is responsible for fault tolerant replication.

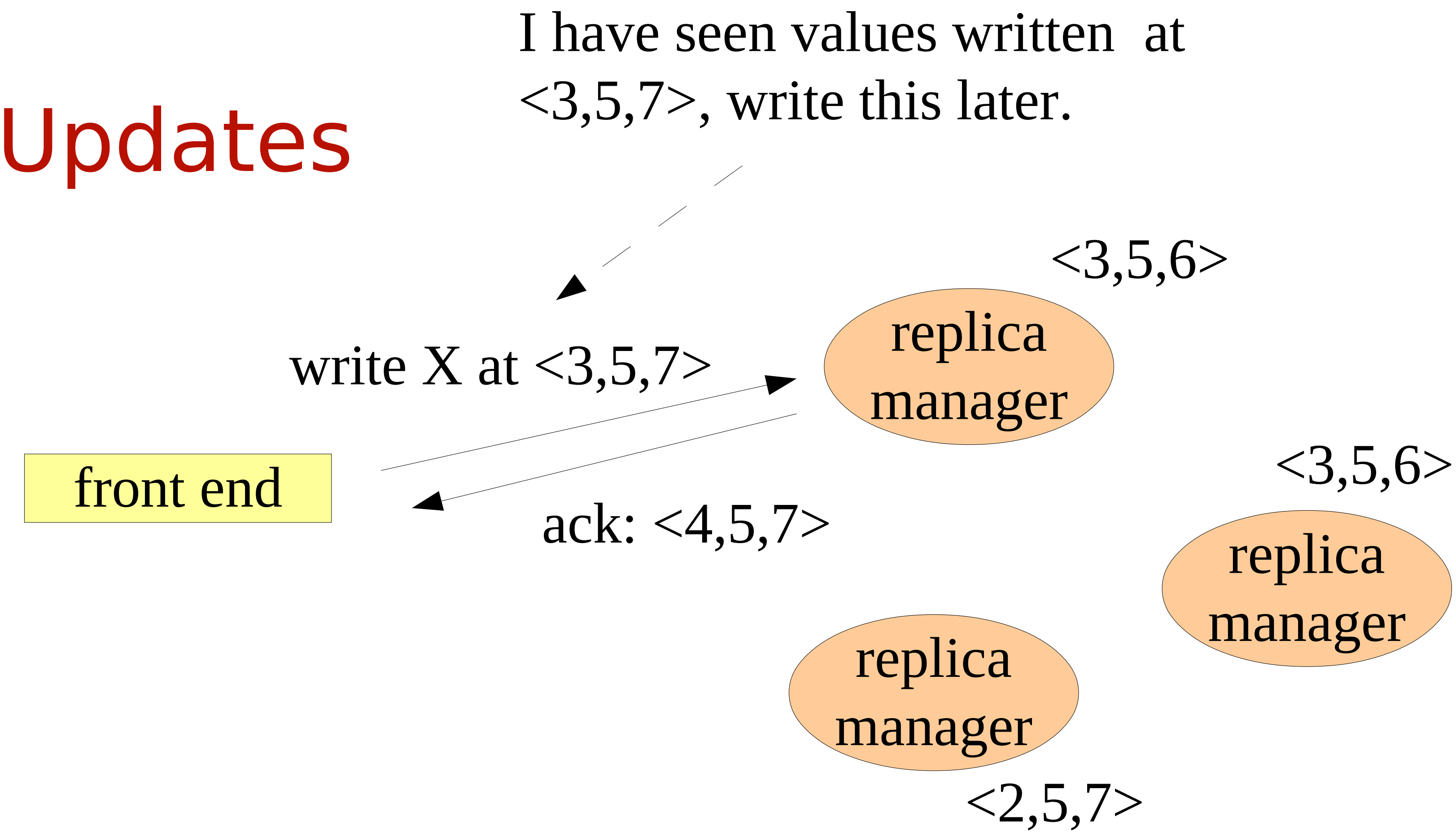
Query


I have seen values written at <2,4,6>, don't give me old data.
read X at <2,4,6>
front end
foo <2,5,6>
replica manager
<2,5,5>
replica manager
<2,5,6>
replica manager
<2,5,6>
Ok, this is the value at <2,5,6>

Query

- A front end sends a query request to any replica manager.
- Query contains vector time stamp.
- Replica manager must hold query until it has all information that *happen-before* the query.
- Replica manager returns response and new time stamp.

Updates



I have seen values written at <3,5,7>, write this later.
<3,5,6>
write X at <3,5,7>
replica manager
front end
<3,5,6>
ack: <4,5,7>
replica manager
replica manager
<2,5,7>

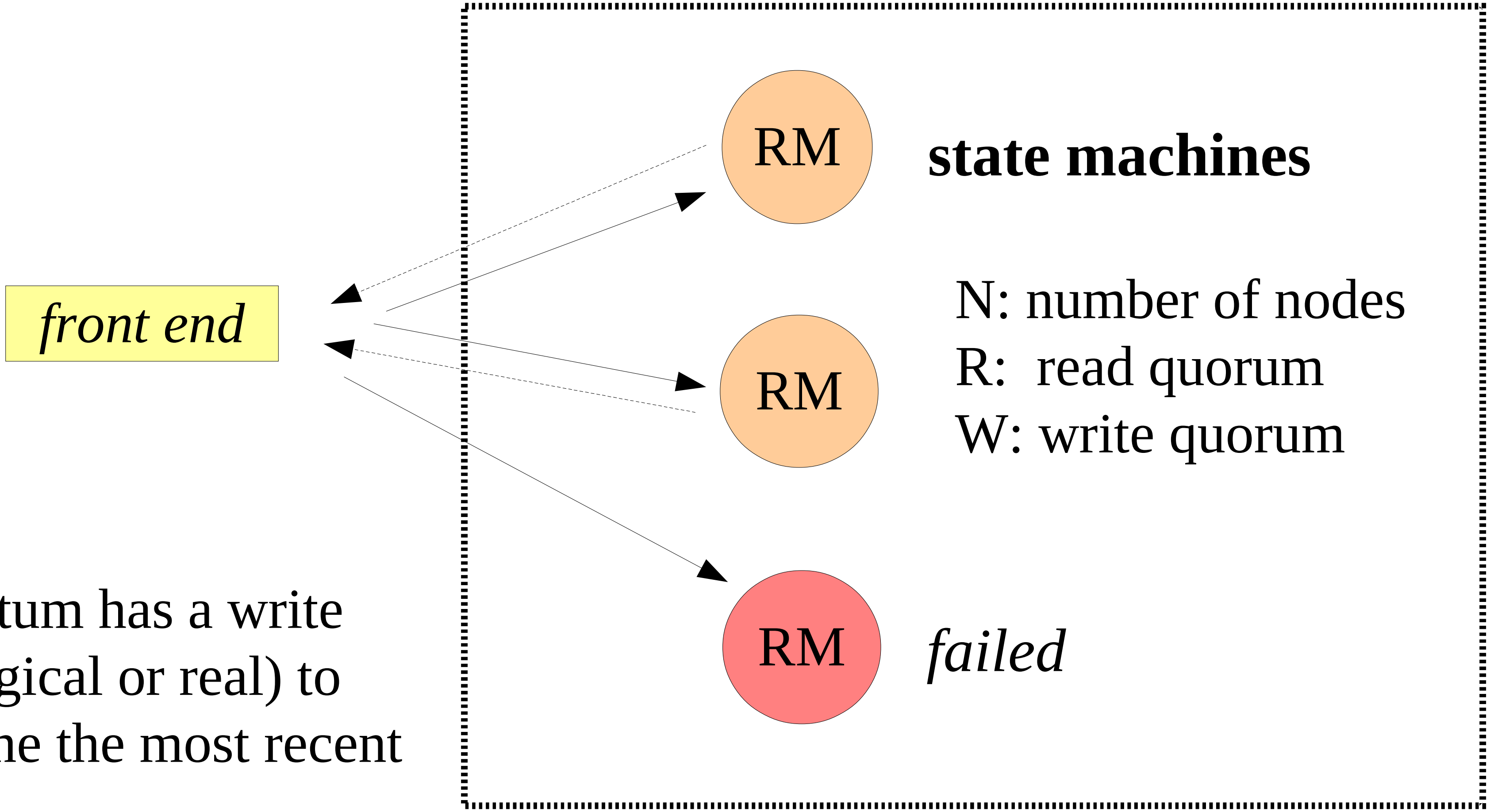
Updates

- Front end sends updates to one (or more) replica manager.
- The update is scheduled by the replica manager to be executed in causal order.
- Updates is sent to remaining replica mangers using the gossip protocol.

Gossip architectures

- Performance at the price or causal consistency.
- Forced and immediate consistency more expensive.
- Can the application live with causal consistency?
- Highly available, only one replica needs to be available.

Quorum based



N: number of nodes
R: read quorum
W: write quorum

Each datum has a write time (logical or real) to determine the most recent

Sequential consistent

- Assume the read or write quorum must be taken in order for the operation to take place.
- R + W > N
  - A read operation will overlaps with the most recent write operation.
  - The time stamp will/might tell which one is most recent.
- W > N/2
  - Two read operations can not occur concurrently.

Summary

- Replicating objects used to achieve fault tolerant services.
- Services should (?) provide single image view as defined by sequential consistency.
- Passive replication
- Active replication
- Gossip based
- Quorum based replication