Replication

Johan Montelius

KTH

HT15

Replication - why

- Performance
  - latency
  - throughput
- Availability
  - service respond despite crashes
- Fault tolerance
  - service consistent despite failures

Challenge

A replicated service should, to the users, look like a non-replicated service.

What do we mean by "look like"?

- linearizable
- sequential consistency
- causal consistency
- eventual consistency

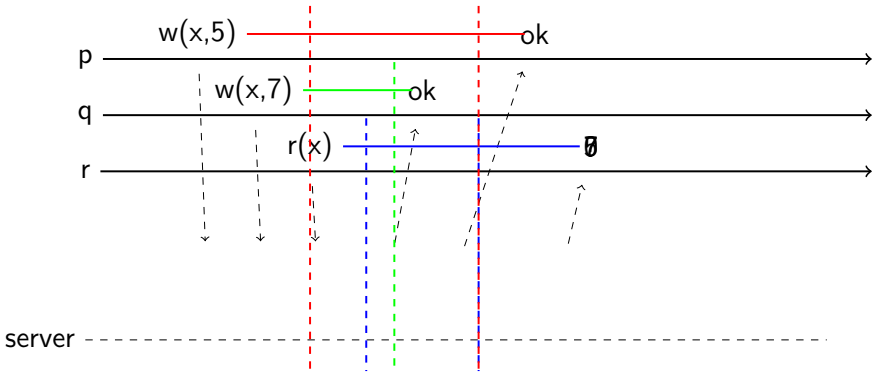
linearizable

A replicated service is said to be *linearizable* if for any execution there is some interleaving of operations that:

- meets the specification of a non-replicated service
- matches the real time order of operations in the real execution

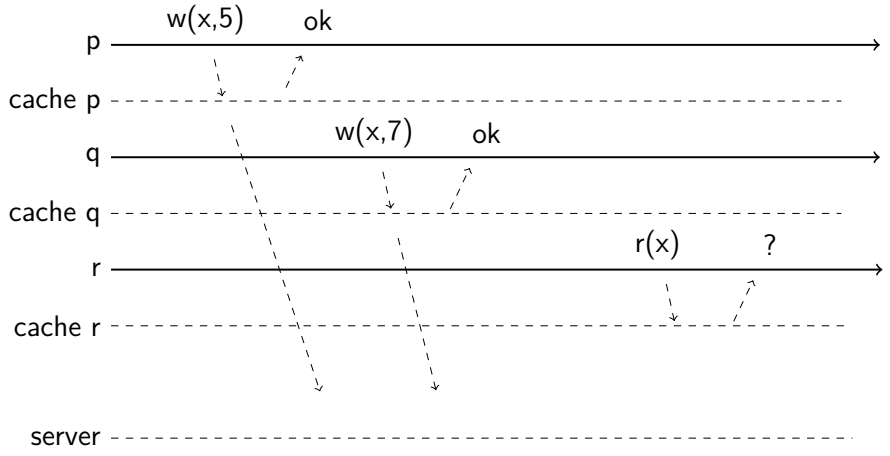
All operations seam to have happened: atomically, at the correct time, one after the other.

linearizable

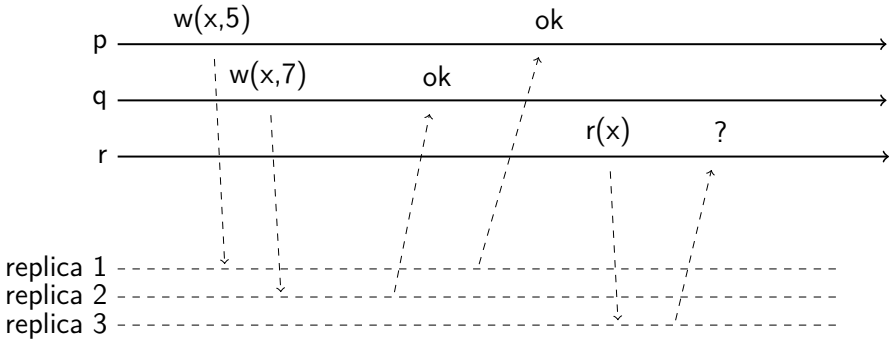


We guarantee that there is a sequence that makes sense.

why would it not make sense?



why would it not make sense?



sequential consistency

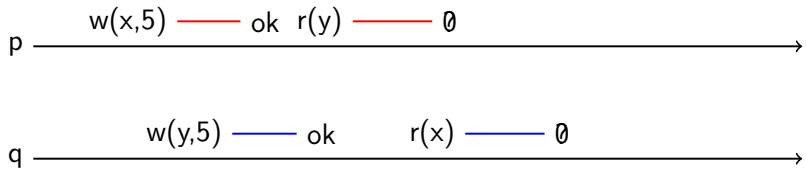
A replicated service is said to be *sequential consistent* if for any execution there is some interleaving of operations that:

- meets the specification of a non-replicated service
- matches the *program order* of operations in the real execution

Don't worry about real time as long as it make sense.

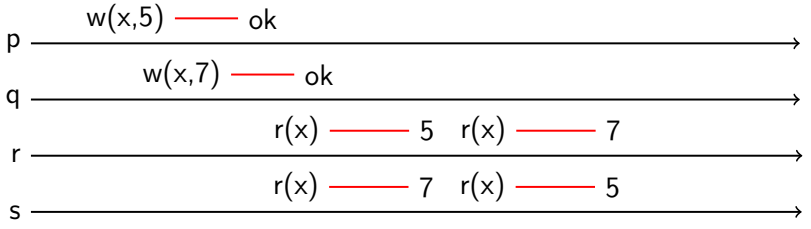
still have to make sense

Assume x and y is initially set to 0



There should exist one total order of the operations that is consistent with the results.

even more relaxed



As long as it make sense for each process.

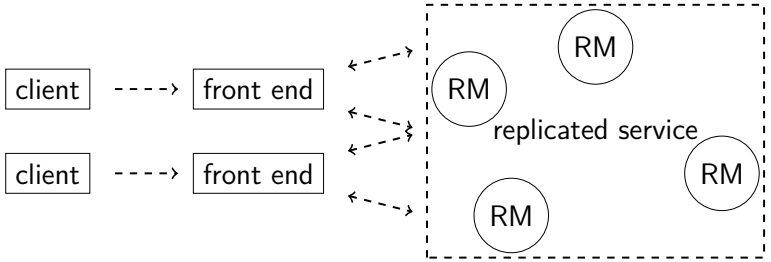
Causal consistency, unordered operations could be seen in different order.

Eventual consistency

There exist a total order that will eventually be visible to all.

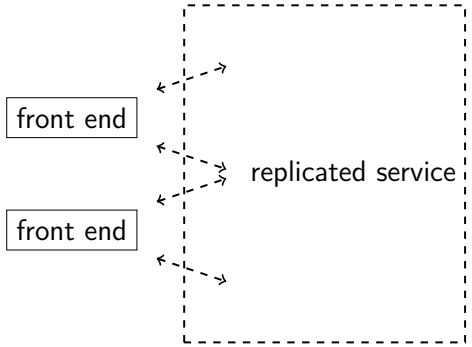
Take DNS servers as an example.

replication system model



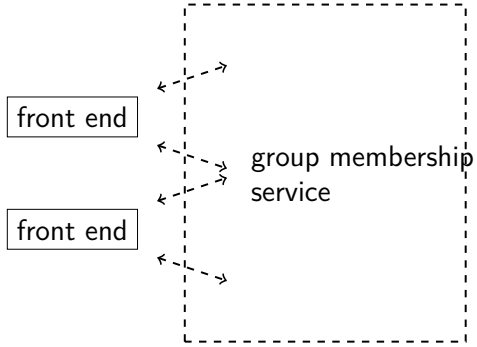
- front end knows about replication scheme, could be implemented on the client side
- replica managers (RM) coordinate operations to guarantee consistency

replication system model



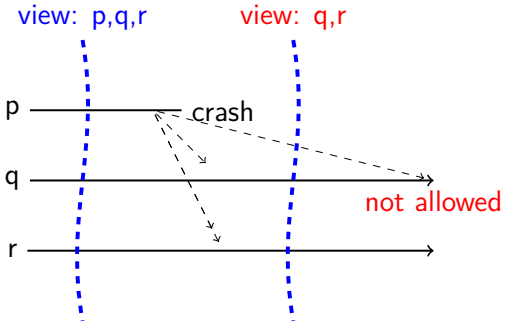
- Request: from front end to one or more replicas
- Coordination: decide on order etc
- Execution: the actual execution of the request
- Agreement: agree on possible state change
- Response: reply received by front-end and delivered to client

group membership service



- adding and deletion of nodes
- ordered multicast
- leader election
- view delivery

view-synchronous group communication

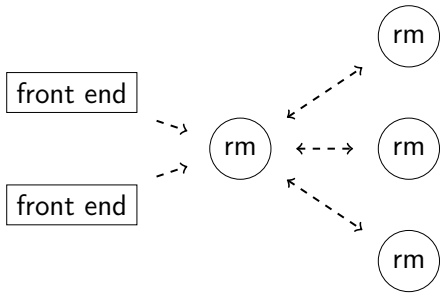


- reliable multicast
- delivered in same view
- never deliver from excluded node
- never deliver not yet included node

passive and active replication

- Passive replication: one primary server and several backup servers
- Active replication: servers on equal term

Passive replication



- Request: front end sends request to primary
- Coordination: primary checks if it is a new request
- Execution: executes and stores response
- Agreement: sends updated state and reply to backup servers
- Response: sends reply to front-end

what about crashes

Primary crashes:

- backups will receive new view with primary missing
- new primary is elected

if front end re-sends request

- either the reply is known and is resent
- or the execution proceeds as normal

Passive replication - consistency

The primary replica manager will serialize all operations. We can provide

linearizability.

pros and cons

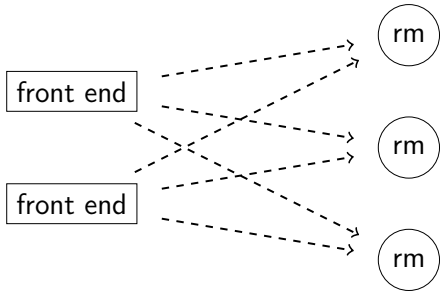
Pros

- All operations passes through a primary that linearize operations.
- Works even if execution is in-deterministic

Cons

- delivering state change can be costly
- View-synchrony could be expensive

Active replication



- Request: front end multicast to all
- Coordination: reliable total order delivery
- Execution: all replicas execute request
- Agreement: no need
- Response: all replicas reply to front end

Active replication - consistency

Sequential consistency:

- all replicas execute the same sequence of operations
- all replicas produce the same answer

Linearizability:

- total order multicast does not guarantee real-time order
- linearizability not guaranteed if front-end acknowledge operation before it has been processed by replicas

Active replication

Pros

- no need to send state changes
- no need to change existing servers
- read request could possibly be sent directly to replicas
- could survive Byzantine failures

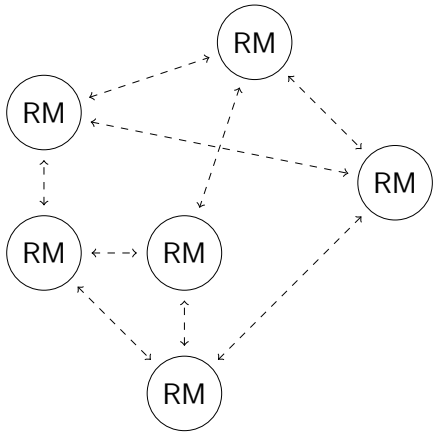
Cons:

- requires total order multicast
- requires deterministic execution

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.
Can we build a system that responds even if all nodes are not available?
Relax the guarantees for consistency.

Gossip architecture

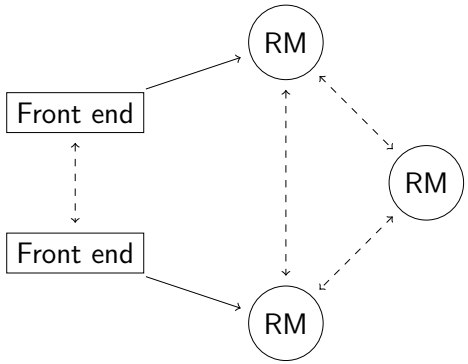


- replica managers interchange update messages
- updates propagate through the network
- sequential consistency not guaranteed
- we want to provide causal consistency

Vector clocks

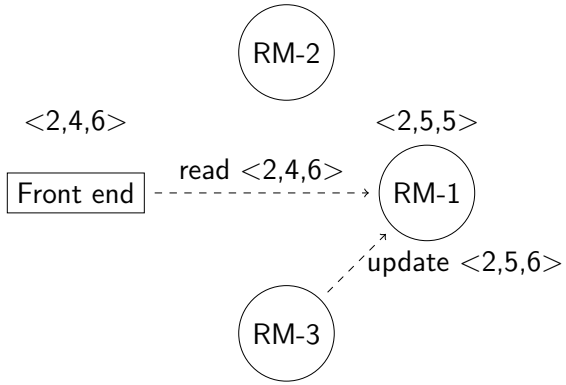
A vector clock with one index per replica manager.
Each update operation will be tagged with a vector clock timestamp.
Some updates are concurrent!

The front end



- one index per replica manager
- front ends keep vector clocks
- replica mangers apply updates in order
- causal consistency guaranteed

The front end

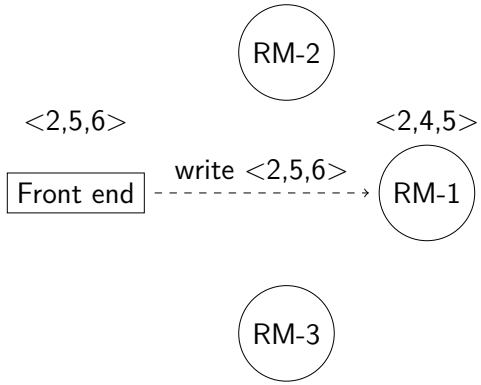


- send a query with timestamp
- check current time, wait for updates
- update will arrive
- update state and clock
- reply
- update clock

The replica manager

The replica manager as a *hold-back queue*, operations that are too early to execute.
As updates arrive it will execute updates, and pending read operations.
Updates are partially ordered.

update operation



- operation with timestamp
- increment clock
- reply with unique timestamp
- update clock
- wait for updates
- update when safe
- update clock

Implementation

Read operations: on hold until safe to answer.

Update operations from front end.

- front end adds *unique id*
- replica checks that it is not a duplicate
- replica replies with unique timestamp
- placed in update log

Gossip operations

- interchange part of update log with *partners*
- place in update log
- provide information on which message a replica has seen
- remove applied operations that has been seen by all replicas

Execute operations

- apply *stable* operations
- in *happen before* order

stable operations and order of execution

- An operation in the log is stable if its time stamp, as provided *by the front end*, is less than or equal to the *value timestamp*.
- Operations must be executed in the order as described by the replica managers in their replies to the front ends.

Causal, forced and immediate

Sometimes we would like to av stronger consistency guarantees:

- Forced: causal and total order in relation to *forced updates*.
- Immediate: causal and total order in relation to *all updates*.

Will of course require that we do some more book keeping.

Gossip architectures

- How many replicas can we have?
- Have hundreds of read-only replicas and a handful of update replicas.
- Will an application cope with causal consistency only?
- How eager should the gossiping be?
- False ordering - we order things that are not necessarily in causal relation to each other.

Summary

- Replication: performance, availability, fault tolerance
- Consistency: linearizable, sequential consistency, ...
- Passive or active replication
- Gossip architectures