KTH ROYAL INSTITUTE
OF TECHNOLOGY

Replication

Vladimir Vlassov and Johan Montelius

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 seem to have happened: atomically, **at the correct time**, one after the other.*

*A register that provides linearizability is called **an atomic register.***

Registers

Safe register

- If read does not overlap write, read returns the value written by the most recent write – the register is safe.
- If read overlaps write, it returns any valid value

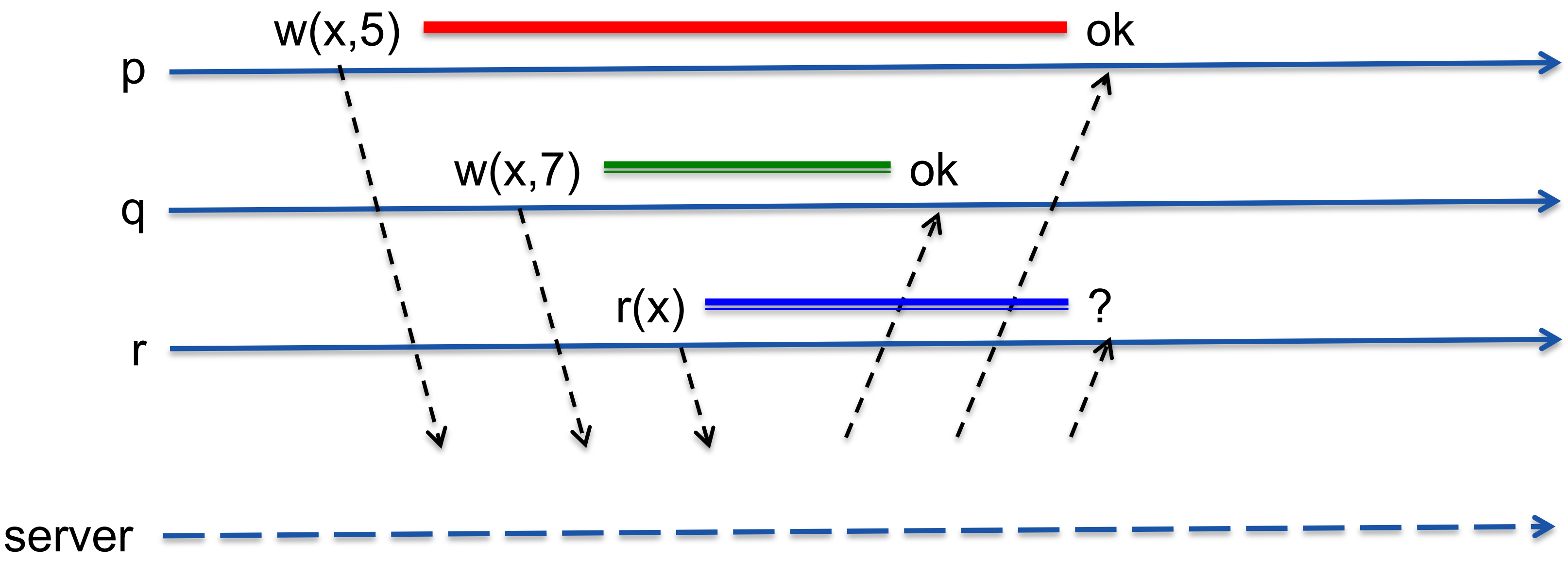
Regular register

- If read does not overlap write, the register is safe
- If read overlaps write, it returns either the old or the new value

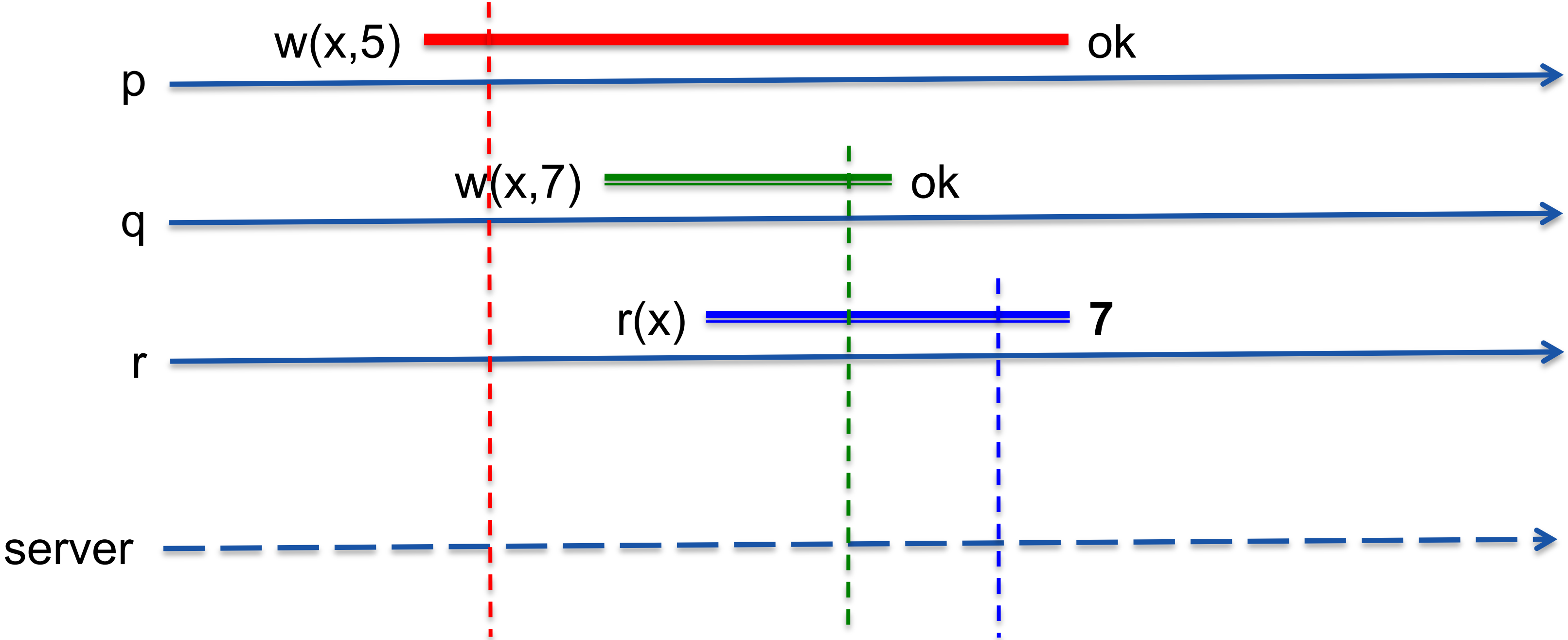
Atomic register (linearizable)

- If read does not overlap write, the register is safe
- If read overlaps with write, it returns either the old value or the new value but not newer than the next read

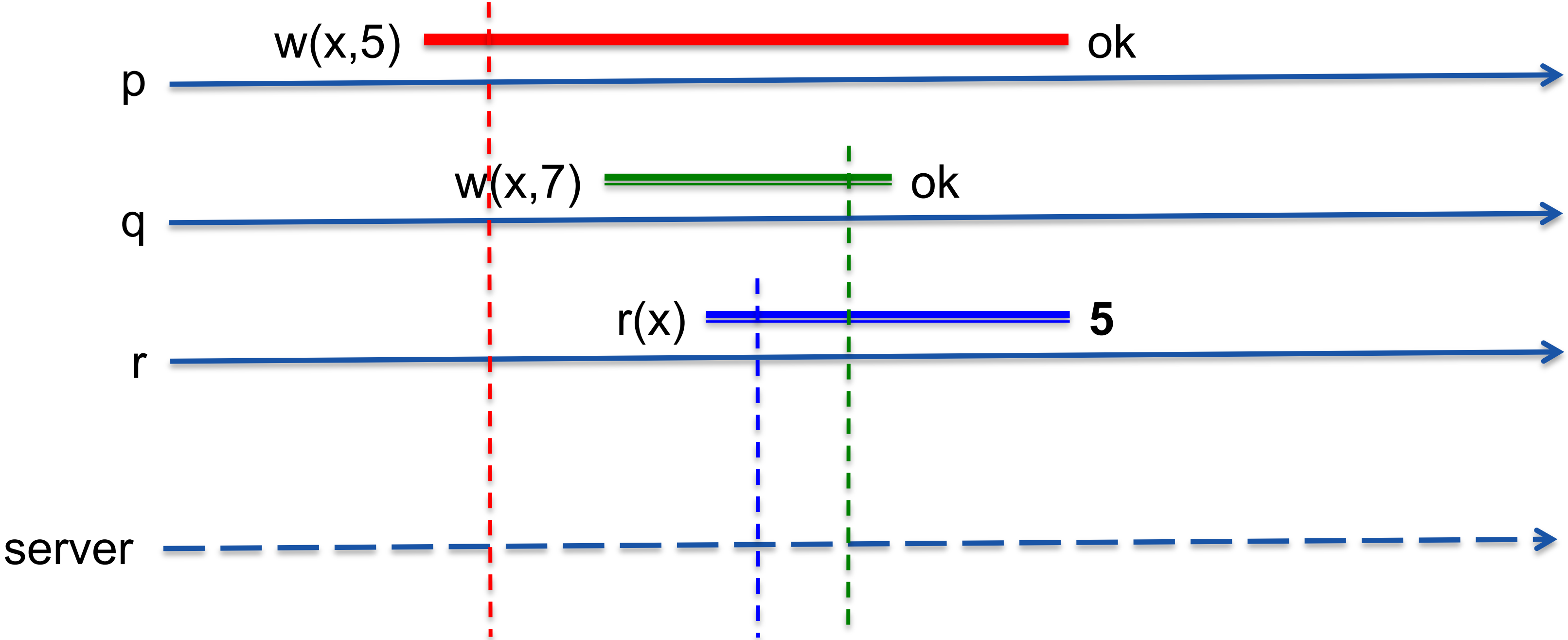
Linearizable



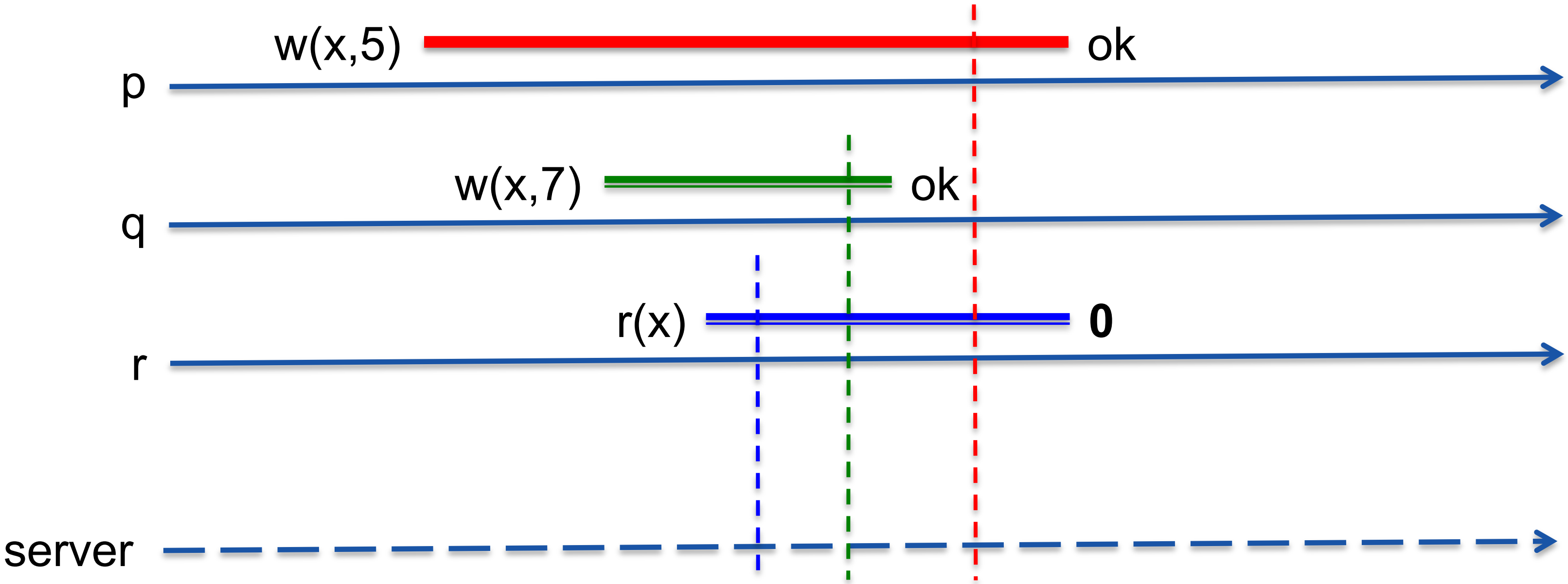
Linearizable



Linearizable

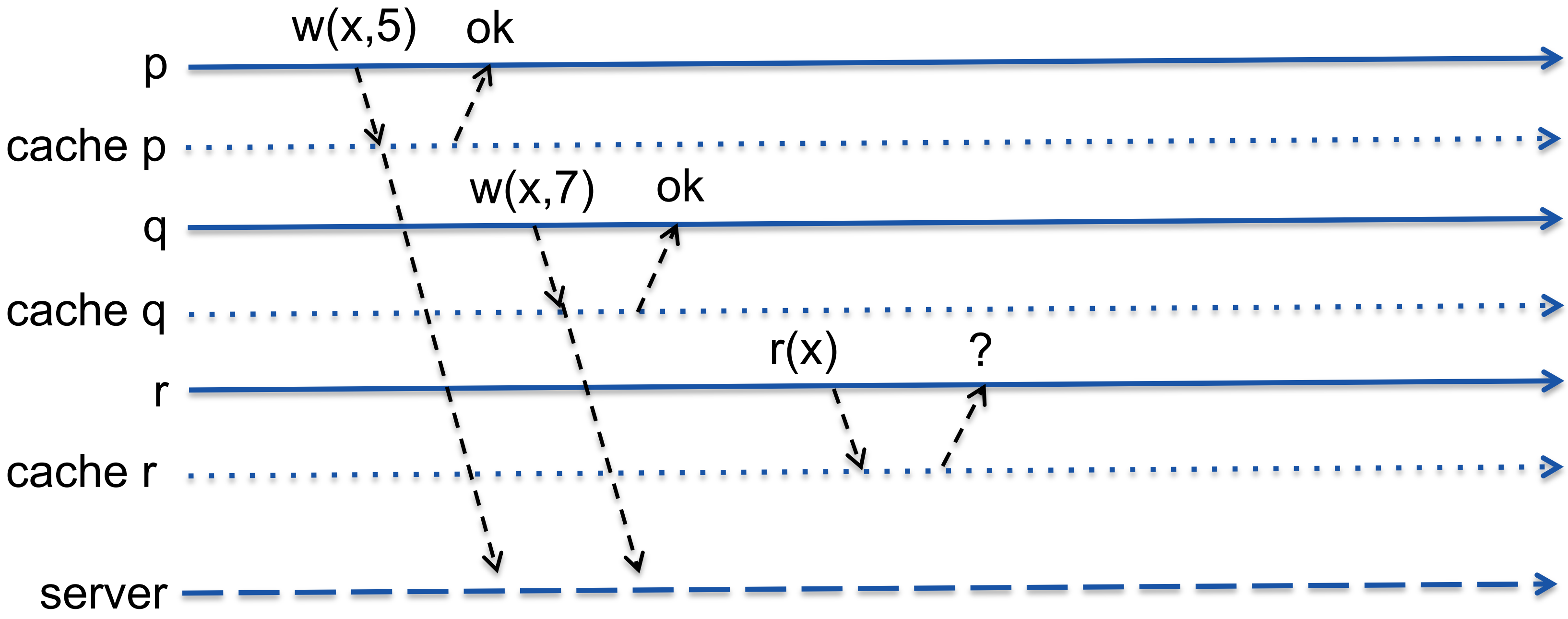


Linearizable



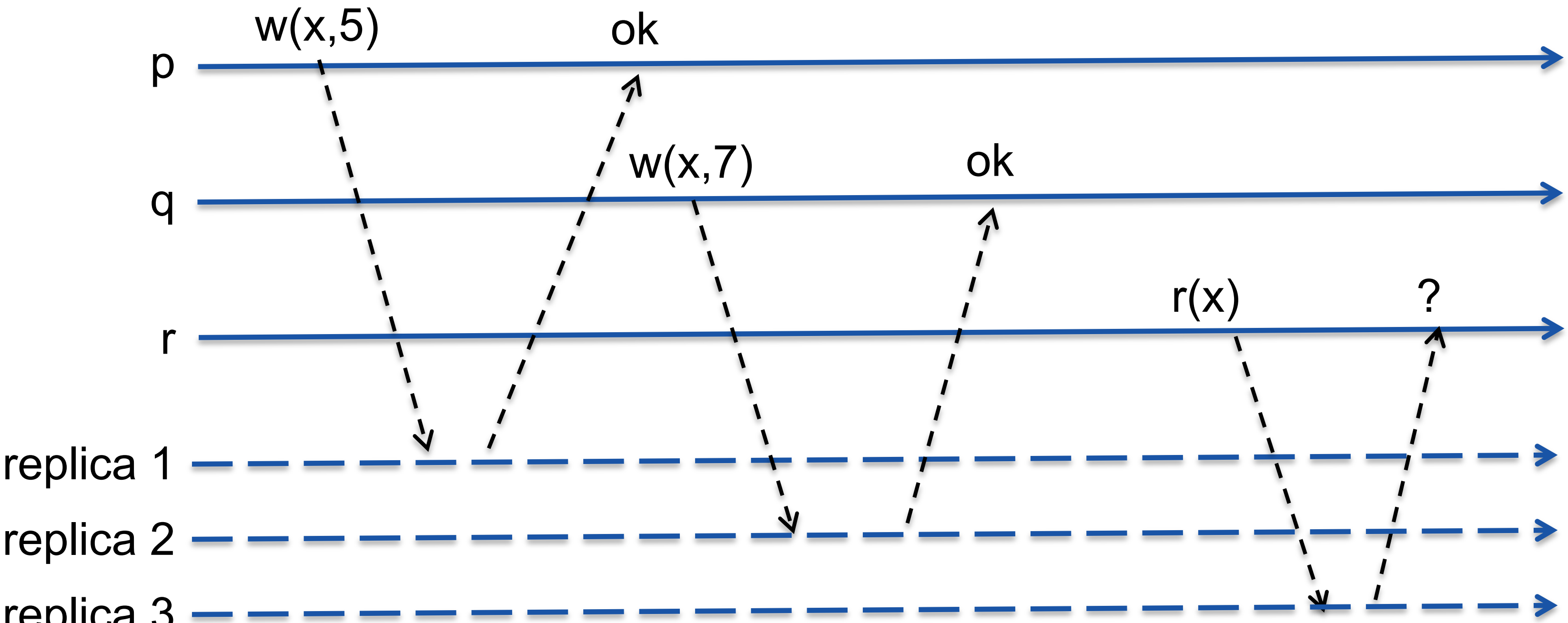
We guarantee that there is a sequence that makes sense.

Why would it not make sense?



What if we use caches?

Why would it not make sense?



What if we use replication?

Sequential consistency

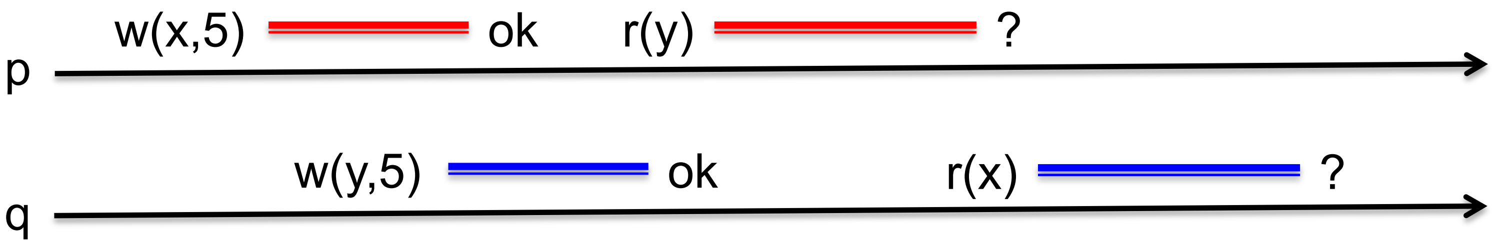
A replicated service is said to be ***sequential consistent*** if, for any execution, there is some interleaving of operations that:

- 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 makes sense.

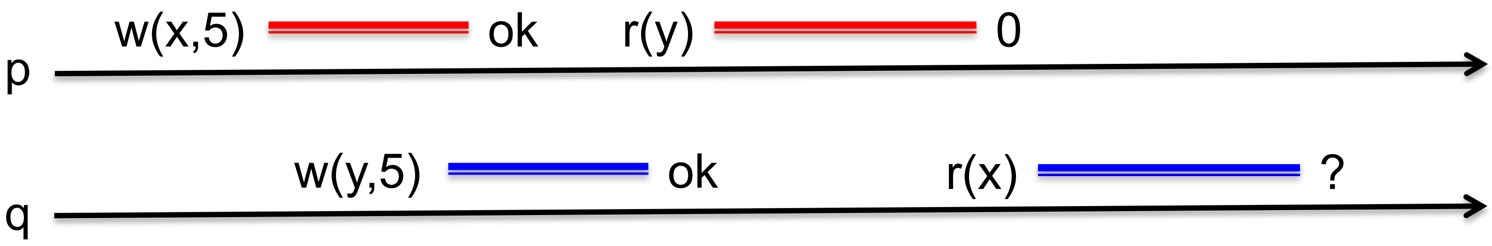
Still have to make sense

Assume x and y is initially set to 0



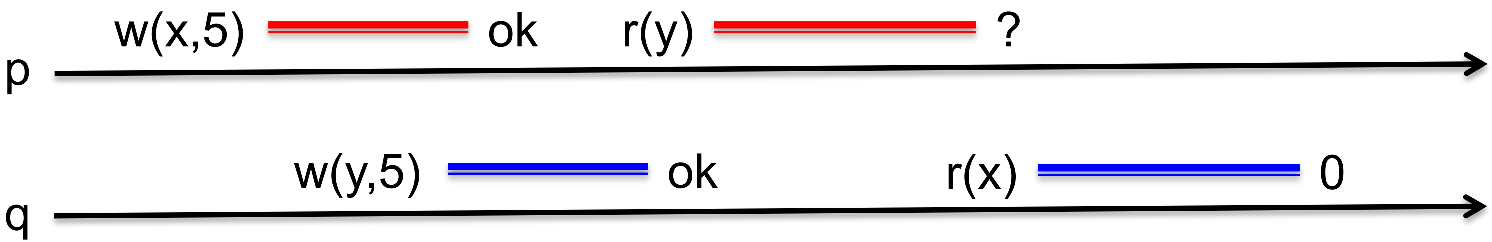
Still have to make sense

Assume x and y is initially set to 0



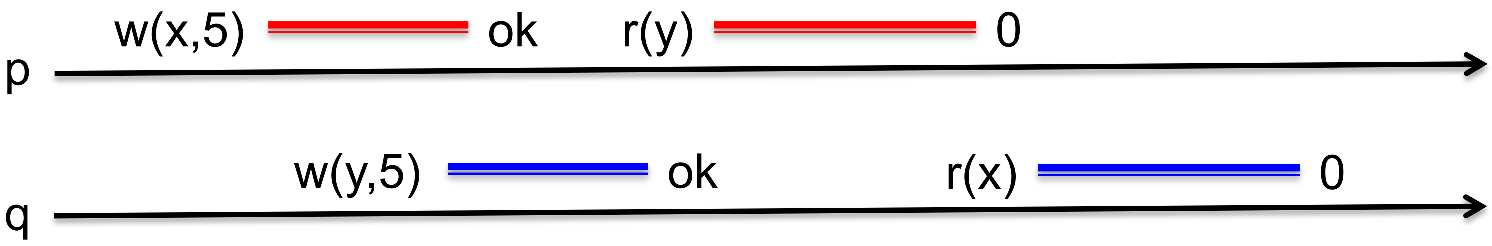
Still have to make sense

Assume x and y is initially set to 0



Still have to make sense

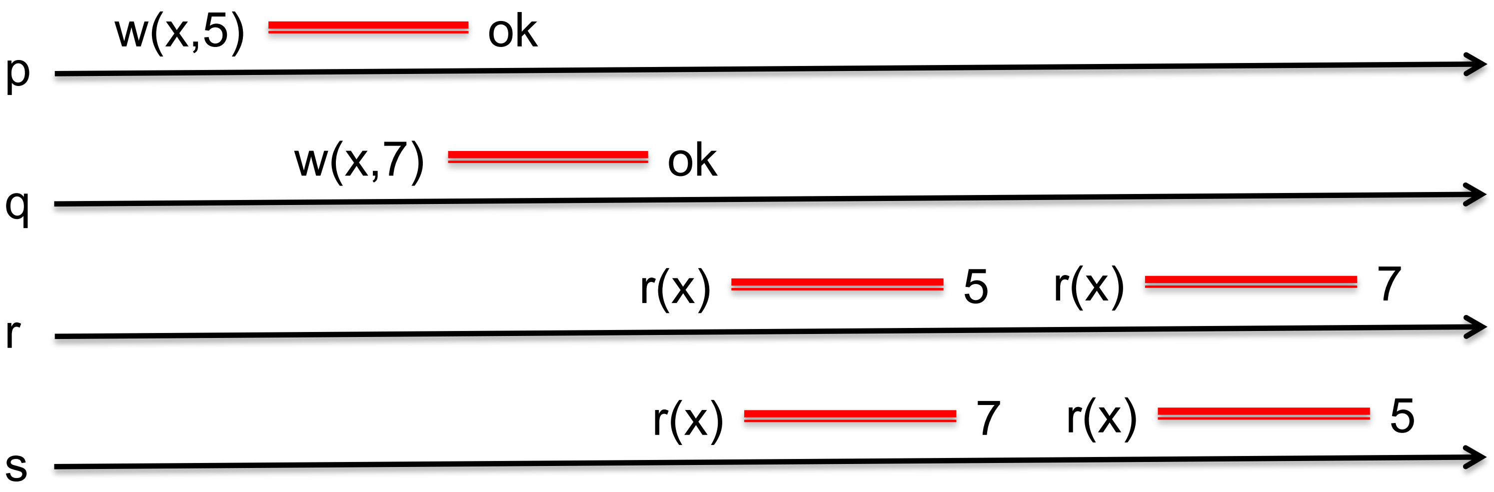
Assume x and y is initially set to 0



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

Total Order Store: this is still ok in X86 architecture (processor consistency).

Even more relaxed



As long as it makes sense for each process.

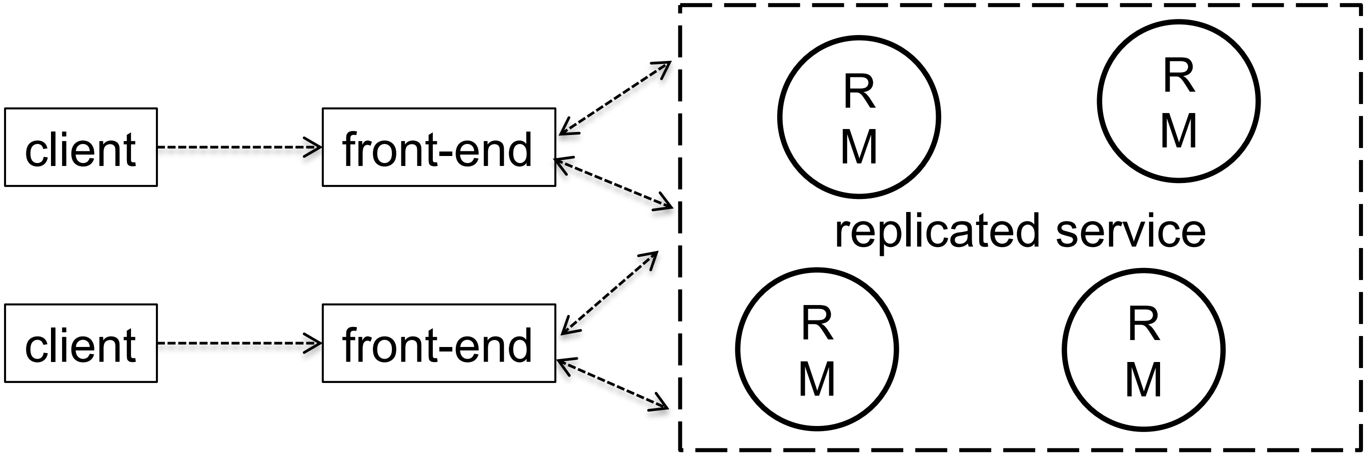
Causal consistency, unordered (causally unrelated) operations could be seen in a different order.

Eventual consistency

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

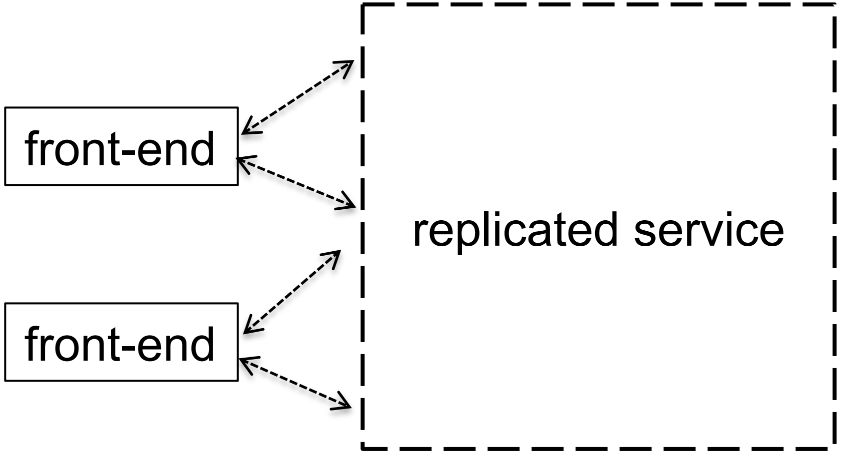
More on this later.

Replication system model



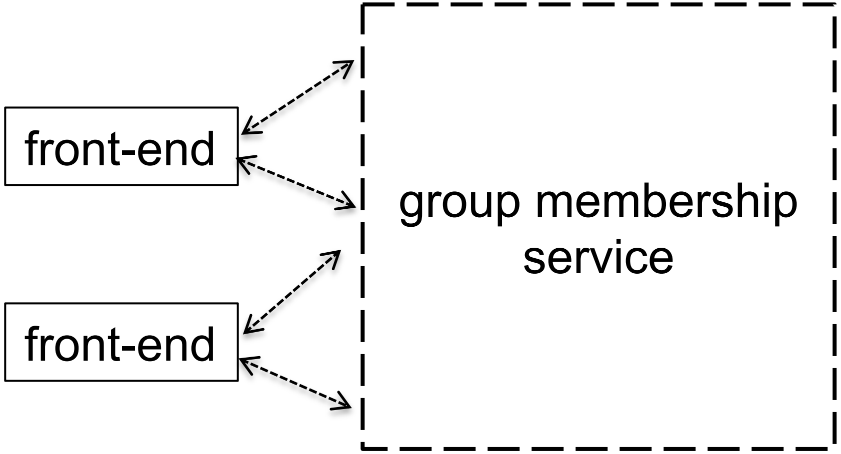
- Front-end knows about replication scheme
  - could be implemented on the client side
- Replica managers (RM) coordinate operations to guarantee consistency.

Replication system model



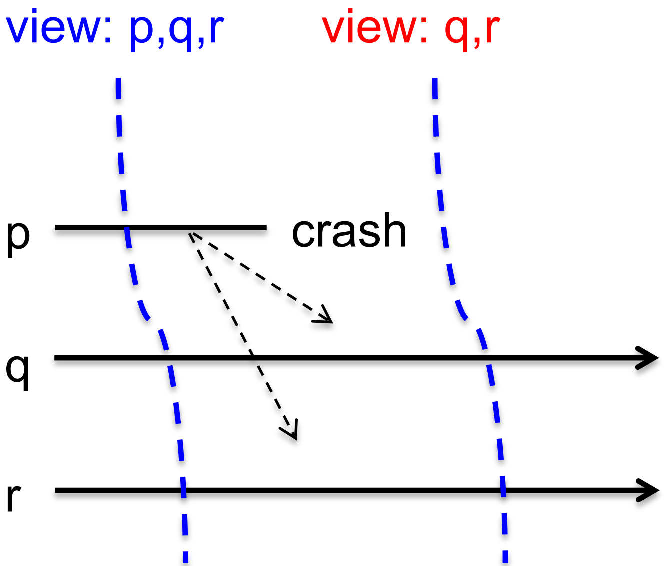
- **Request**: from front-end to one or more replicas
- **Coordination**: decide on order etc
- **Execution**: the actual execution of the request
- **Agreement**: agree on possible state change
- **Response**: reply received by front-end and delivered to the client

Group membership service



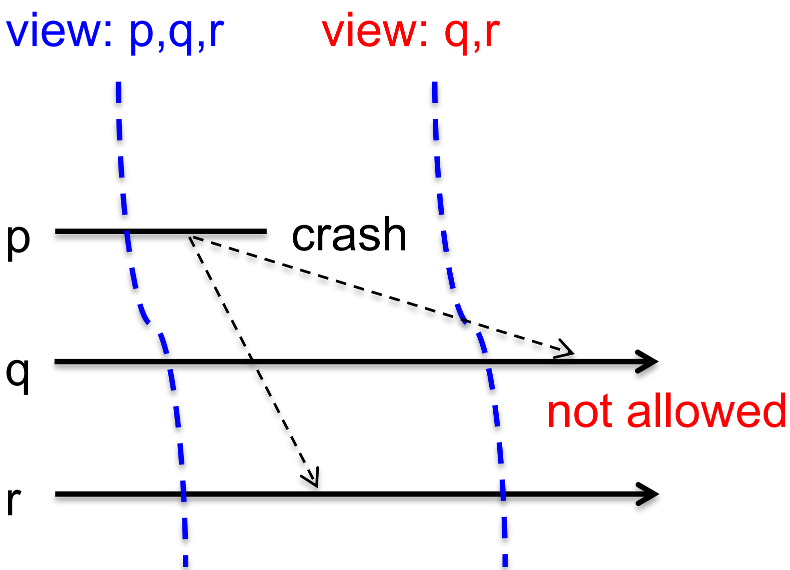
- adding and removing nodes
- ordered multicast
- leader election
- view delivery

View-synchronous group communication



- reliable multicast
- delivered in the same view

View-synchronous group communication

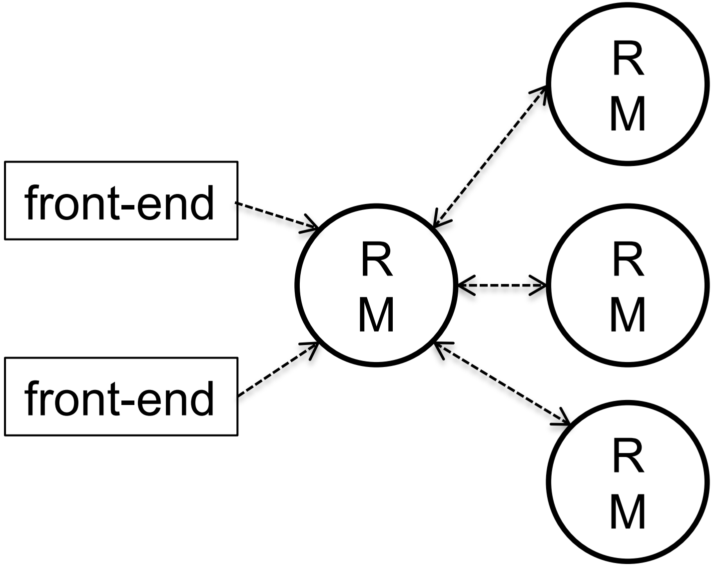


- reliable multicast
- delivered in same view
- never deliver from excluded node
- never deliver not yet included node

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

What about crashes

Primary crashes:

- backups will receive a new view with primary missing
- a 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*.

Passive replication – Pros and cons

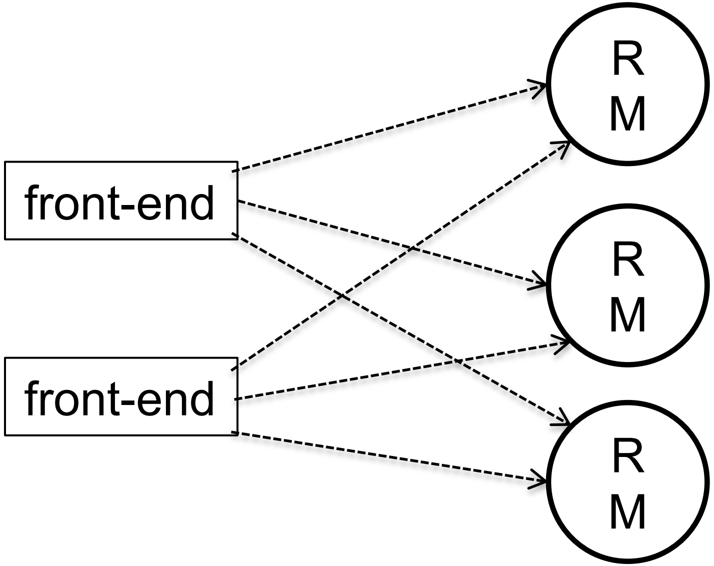
Pros

- All operations pass through a primary that linearizes operations.
- Works even if execution is non-deterministic

Cons

- Delivering state change can be costly.
- Replicas are under-utilized.
- View-synchrony and leader election could be expensive.

Active replication



- *Request*: front-end multicast to all
- *Coordination*: reliable total order delivery
- *Execution*: all replicas execute the request
- *Agreement*: no need
- *Response*: all replicas reply to the 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 is not guaranteed if the front-end acknowledges an operation before replicas have processed it.

Active replication – Pros and cons

Pros

- No need to send state changes.
- No need to change existing servers.
- Read requests could be sent directly to replicas.
- Could survive Byzantine failures.

Cons:

- Requires total order multicast.
- Requires deterministic execution.

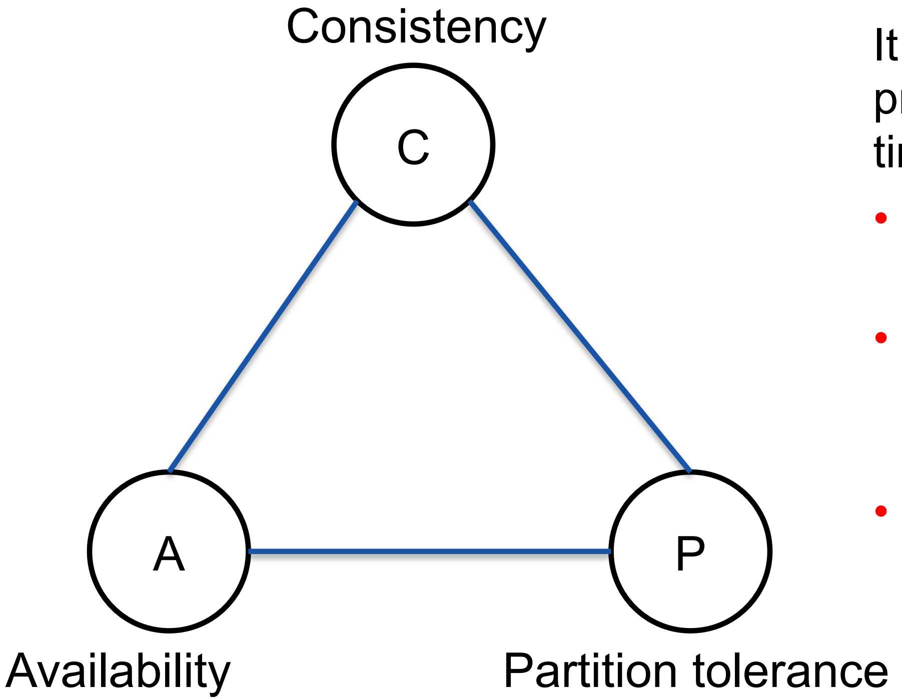
Availability

Both replication schemes require that servers are available.

If a server crashes, detecting and removing the faulty node will take some time.

Can we build a system that responds even if some nodes are unavailable?

The CAP theorem



It is impossible for a distributed system to provide all three guarantees at the same time:

- ***Consistency*** (all nodes see the same data at the same time)
- ***Availability*** (every request receives a response about whether it succeeded or failed)
- ***Partition tolerance*** (the system continues to operate despite arbitrary partitioning due to network failures)

The CAP theorem

You can not have a consistent and always available system if you’re in an environment where you face network partitions.

When there is a network partition:

- limit operations, i.e., some operations are not available,
- continue, but record all operations that could cause an inconsistency.

When the system re-connects: merge operations are performed in separate partitions.

The CAP theorem

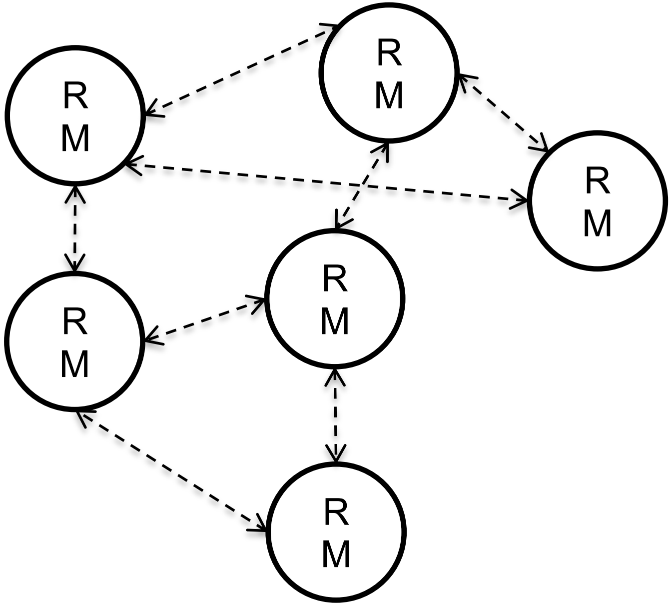
An alternative is to relax consistency.

- ***BASE***: Basic Availability, Soft-state, ***Eventual consistency***

Used by many large-scale key-value stores and replicated distributed services

Gossip architecture

What if we only need to provide causal consistency?



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

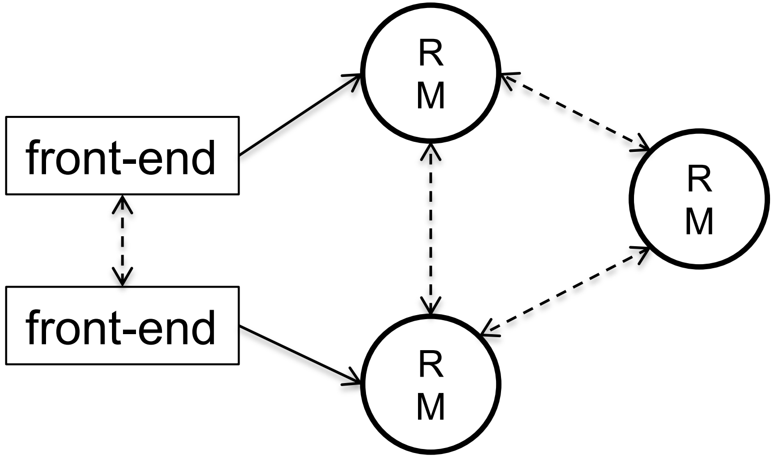
Vector clocks

A vector clock with one index per replica manager.

Each update will be tagged with a vector clock timestamp.

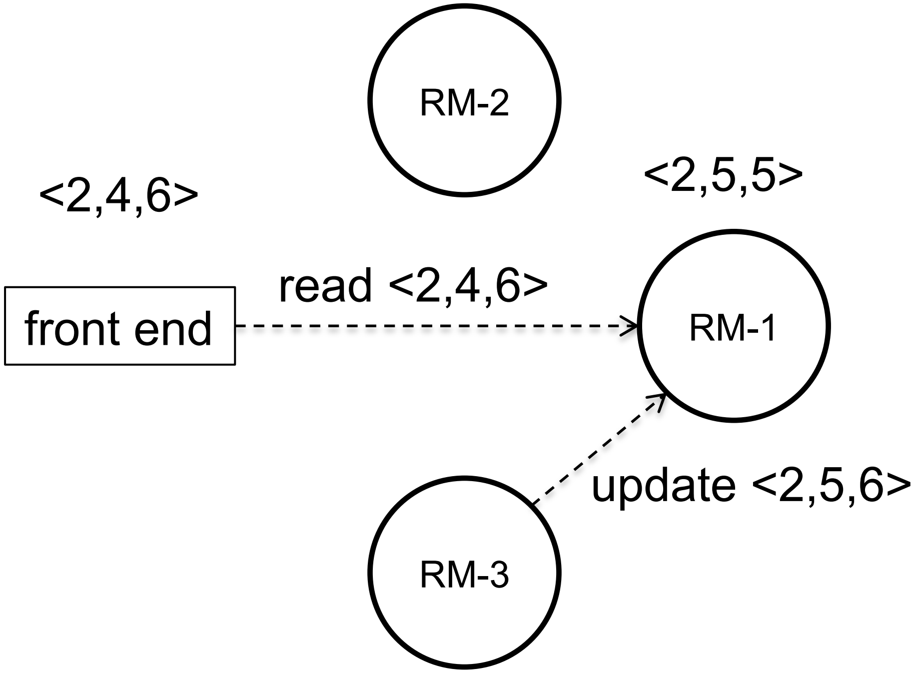
Some updates are concurrent!

The front end



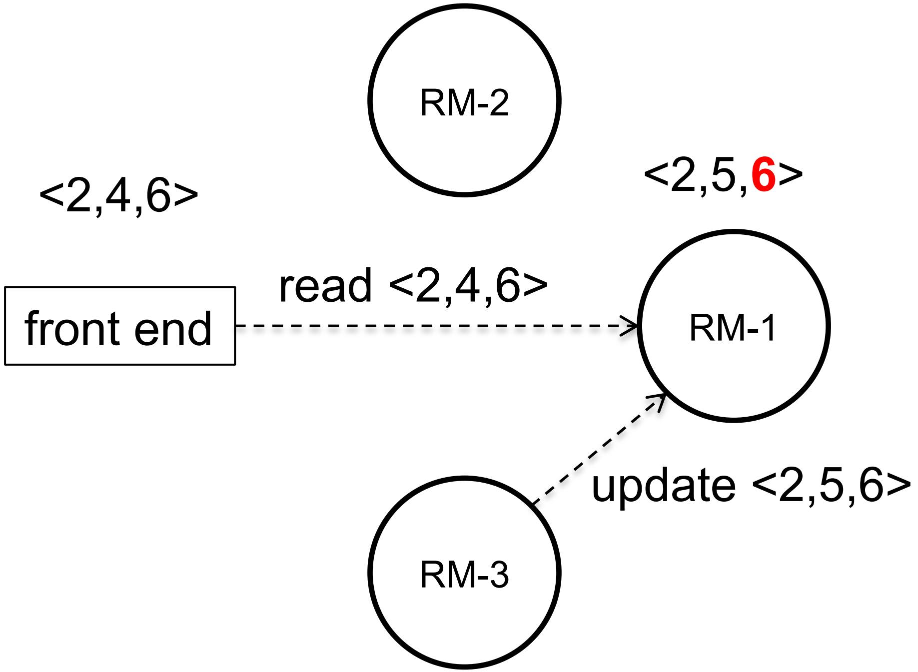
- one index per replica manager
- front-ends keep vector clocks
- replica managers apply updates in order
- causal consistency guaranteed

The front end



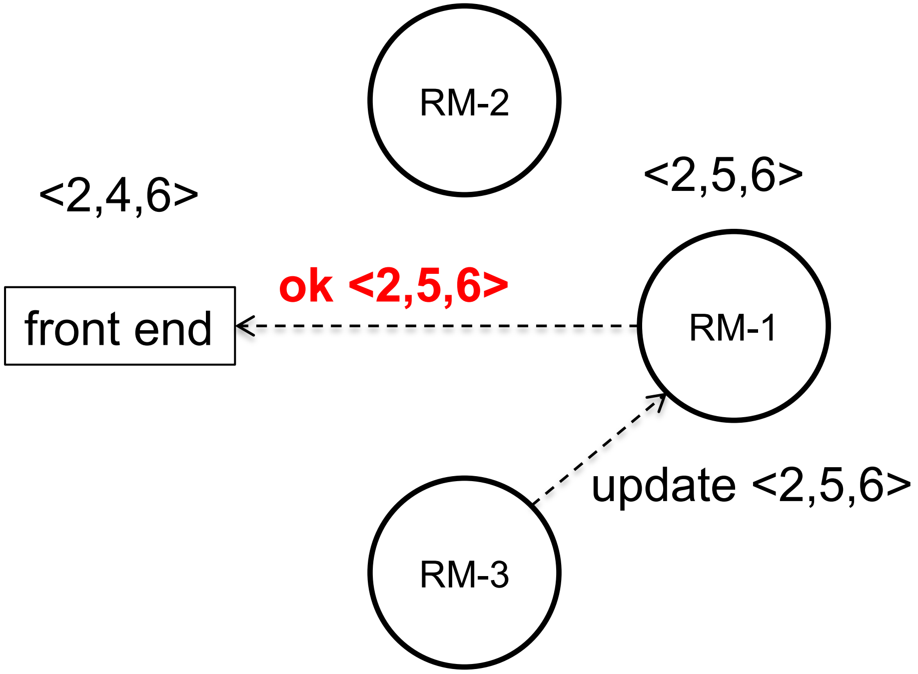
- send a query with a timestamp
- check the current time, wait for updates
- update will arrive

The front end



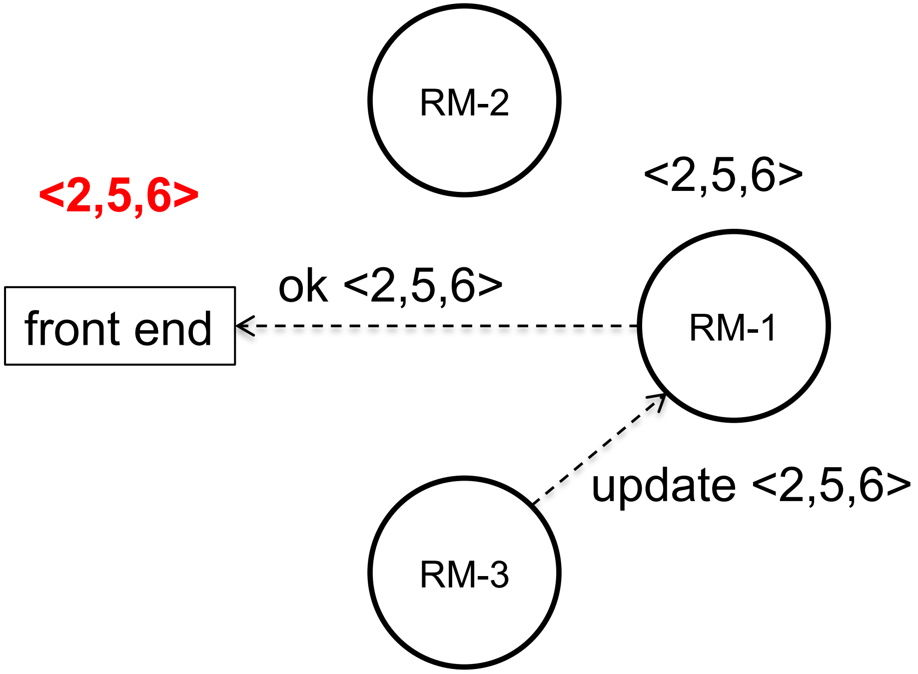
- send a query with a timestamp
- check the current time, wait for updates
- update will arrive
- **update state and clock**

The front end



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

The front end



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

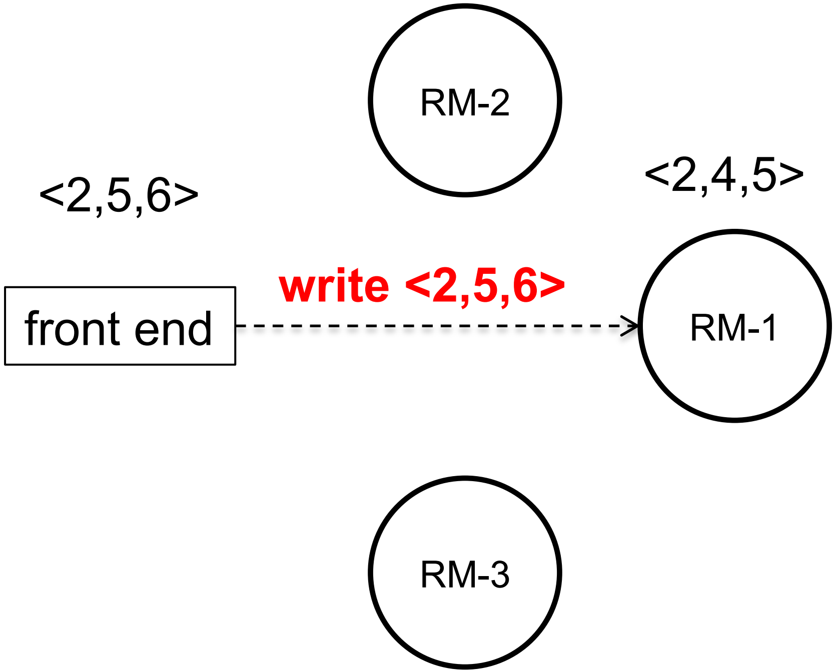
The replica manager

The replica manager has a ***hold-back queue*** for operations that are too early to execute.

As updates arrive, the replica will execute updates and pending read operations.

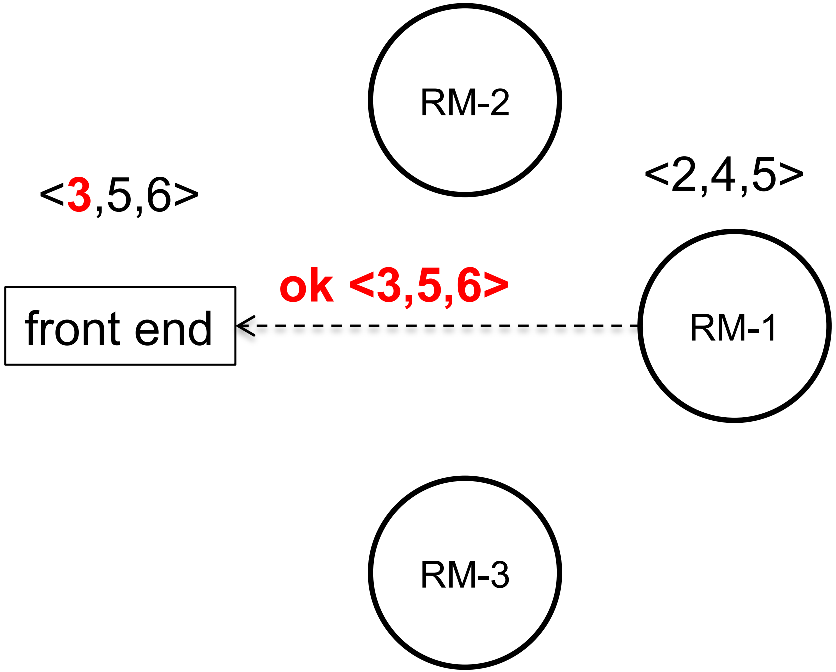
Updates are partially ordered.

Update operation



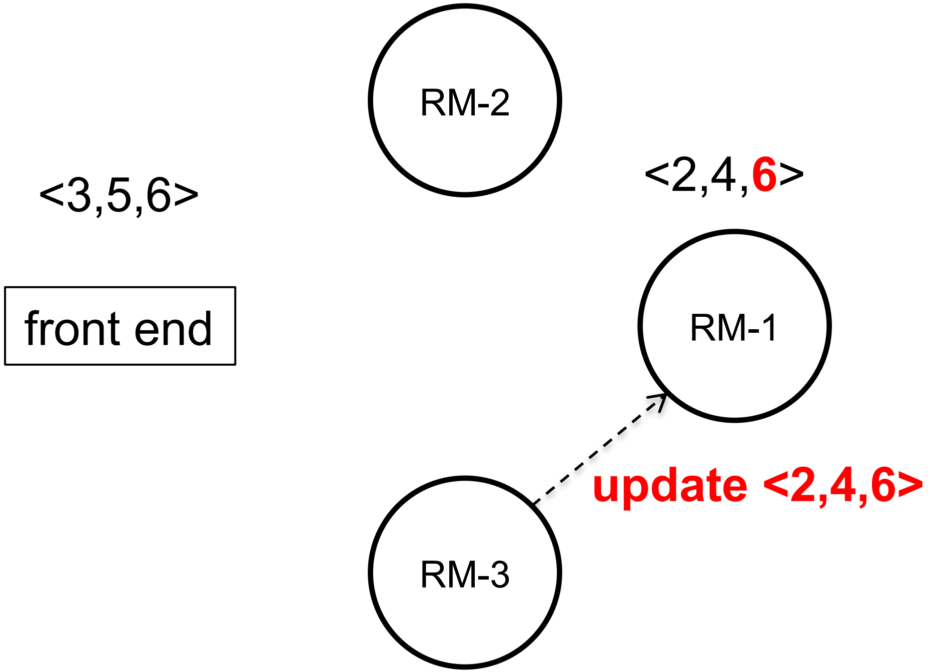
- **operation with timestamp**

Update operation



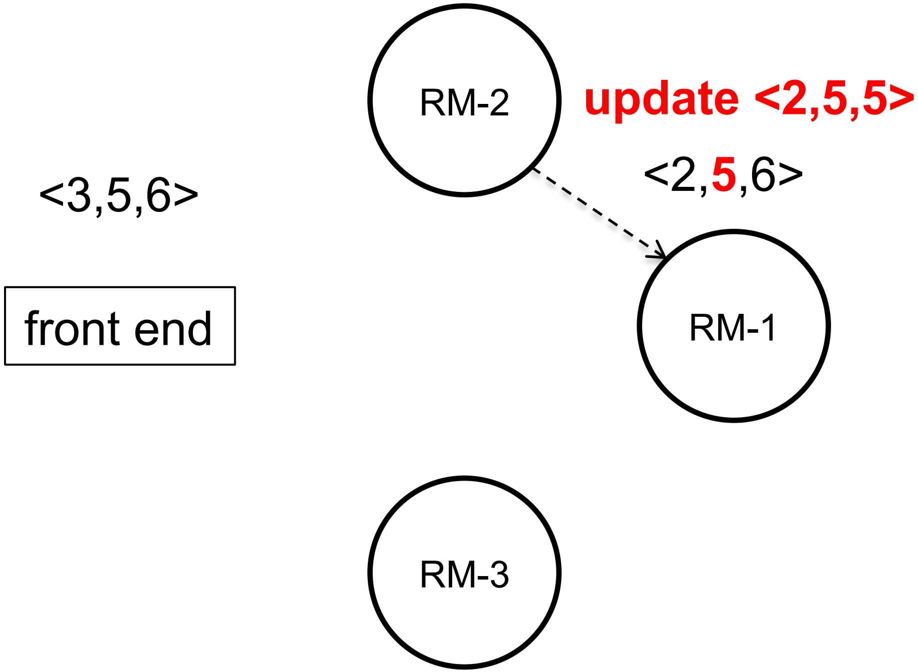
- operation with timestamp
- **reply with a unique timestamp**

Update operation



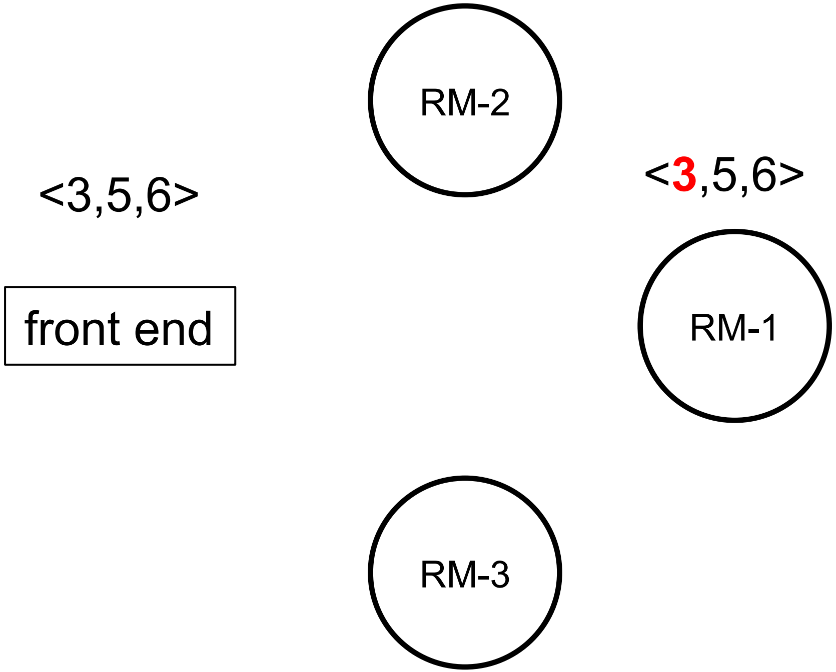
- operation with timestamp
- reply with a unique timestamp
- **wait for updates**

Update operation



- operation with timestamp
- reply with a unique timestamp
- **wait for updates**

Update operation



- operation with timestamp
- reply with a unique timestamp
- wait for updates
- **perform write when safe**

Implementation

Read operations: on hold until safe to answer.

Update operations from the front end.

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

Gossip operations

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

Execute operations

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

Stable operations and order of execution

- An operation in the log is ***stable*** if its timestamp, provided by *the front end*, is less than or equal to the value timestamp.

- Operations must be executed in the order described by the replica managers in their replies to the front-ends.

Causal, forced and immediate

Sometimes we would like to have stronger consistency guarantees:

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

Will, of course, require that we do some more bookkeeping.

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
- The CAP theorem
- Gossip architectures for causal consistency

Lecture resumes 14:15