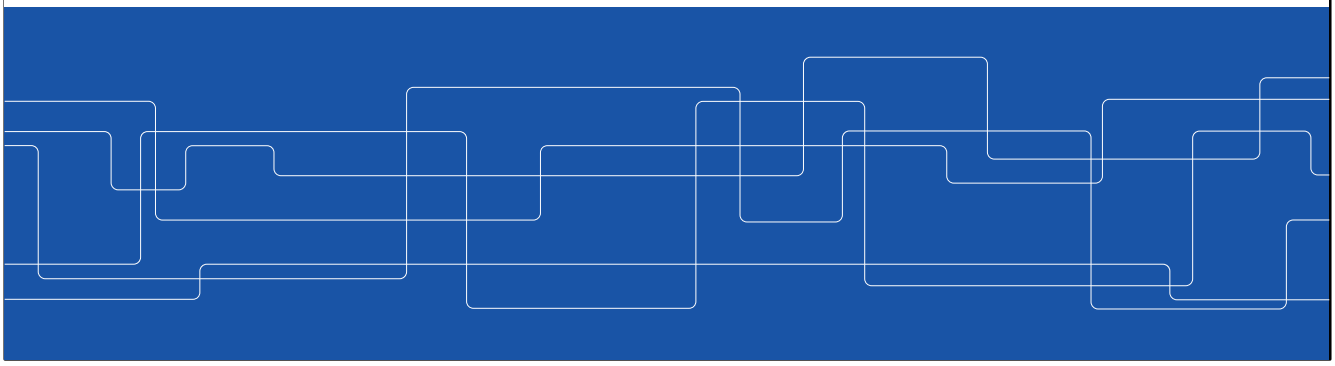
KTH ROYAL INSTITUTE
OF TECHNOLOGY

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

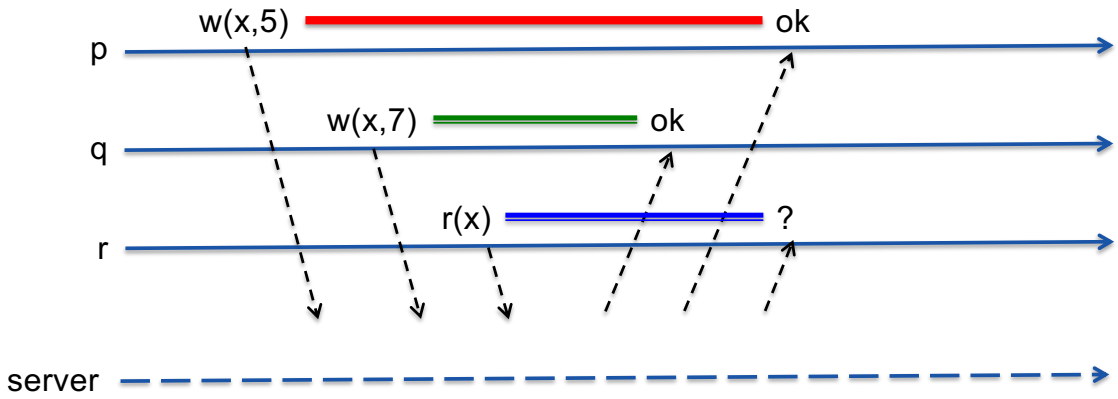
Regular register

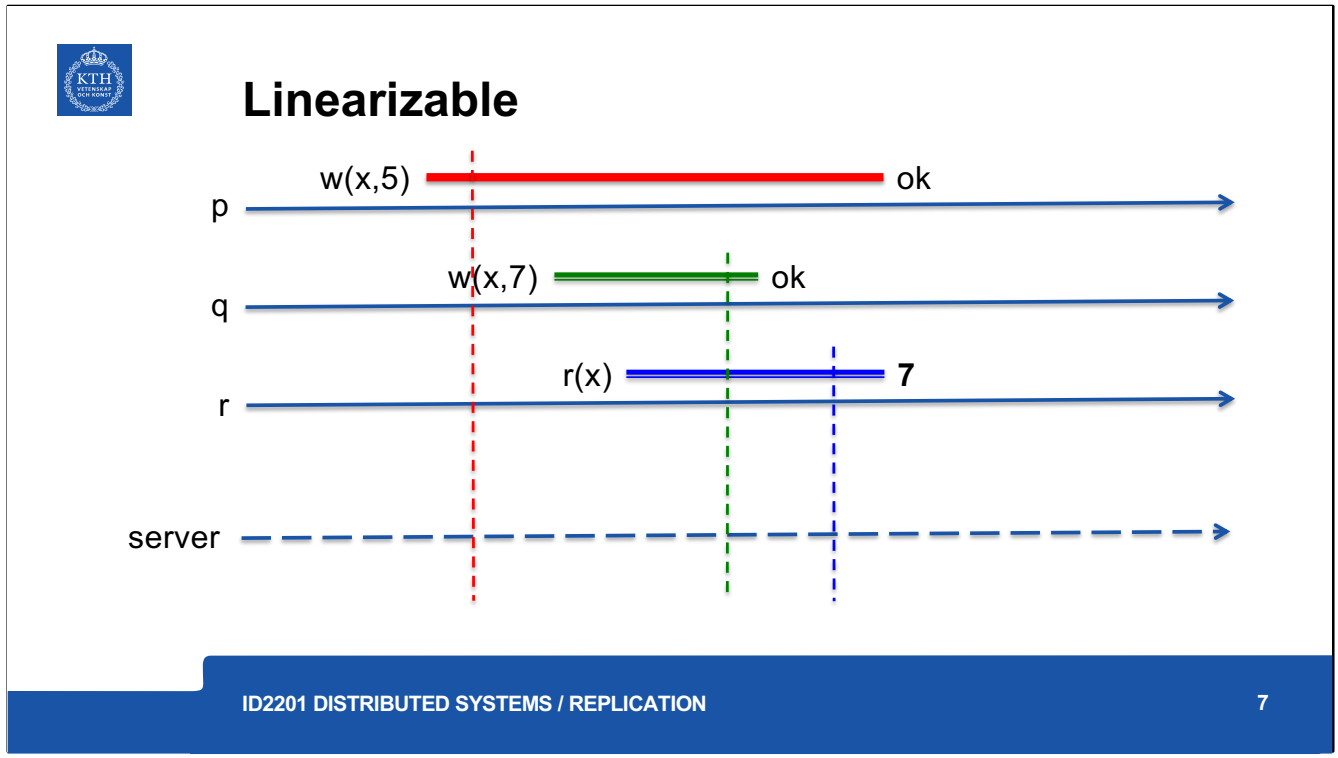
- If read does not overlap write, the register is safe
- If read overlaps write, it returns either the old or the new value

Atomic register (linearizable)

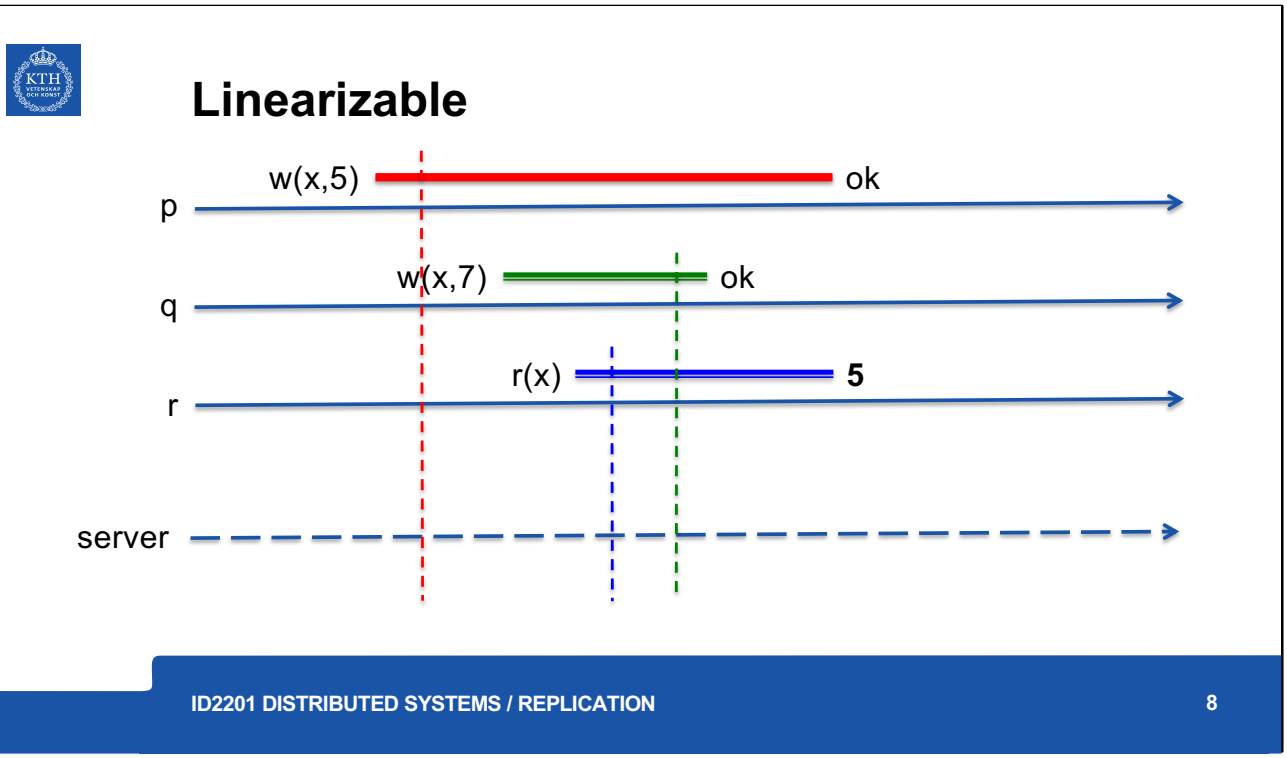
- If read does not overlap write, the register is safe
- If read overlaps with write, it returns either the old value or the new value but not newer than the next read

Linearizable



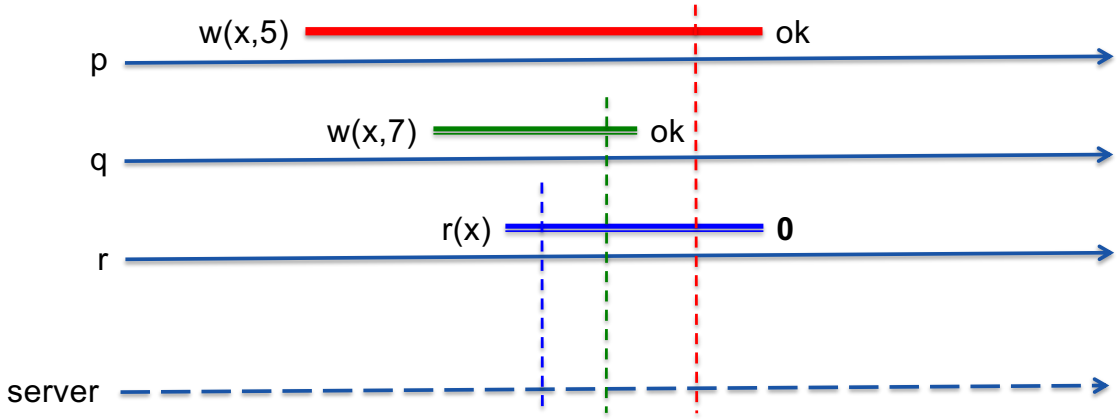


Linearizable



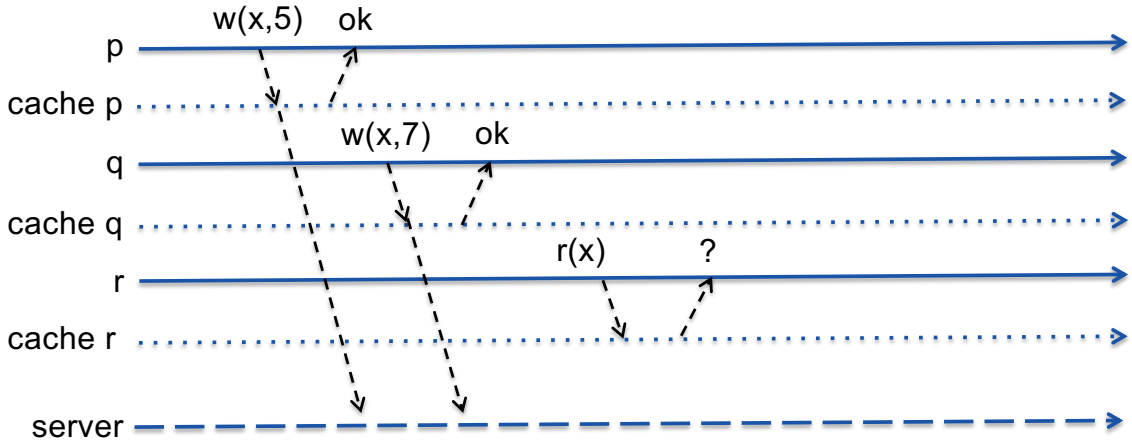
Linearizable

Linearizable

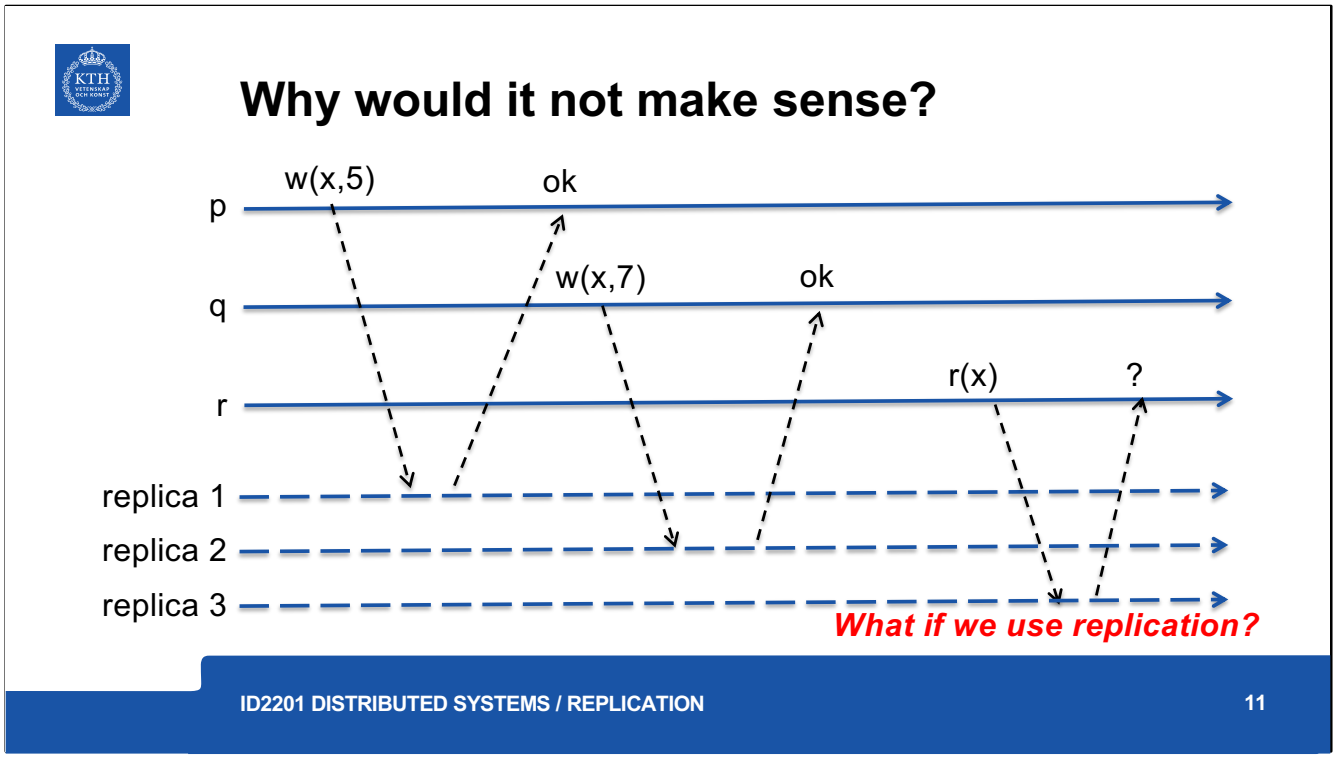


We guarantee that there is a sequence that makes sense.

Why would it not make sense?



What if we use caches?



In a naïve replication strategy, the result is unpredictable

Sequential consistency

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

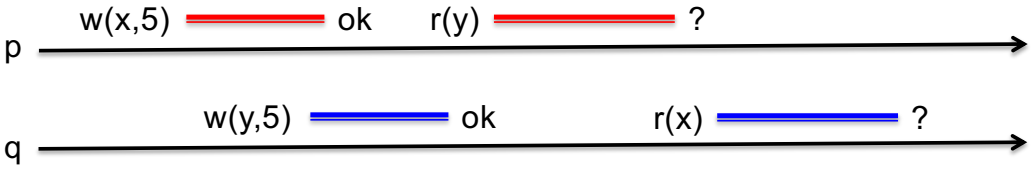
- meets the specification of a non-replicated service
- matches the *program order* of operations in the real execution

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

The real-time requirement in linearizability is desirable in an ideal world because it captures our notion that clients should receive up-to-date information. **But, equally, the presence of real time in the definition raises the issue of linearizability's practicality** because we cannot always synchronize clocks to the required degree of accuracy. **A weaker correctness condition is sequential consistency**, which captures an essential requirement concerning the order in which requests are processed without appealing to real time. We replace the real-time requirement with "**The order of operations in the interleaving is consistent with the program order in which each individual client executed them.**"
Note that absolute time does not appear in this definition of sequential consistency. Nor does any other total order on all operations. The only notion of ordering that is relevant is the order of events at each separate client – the program order.

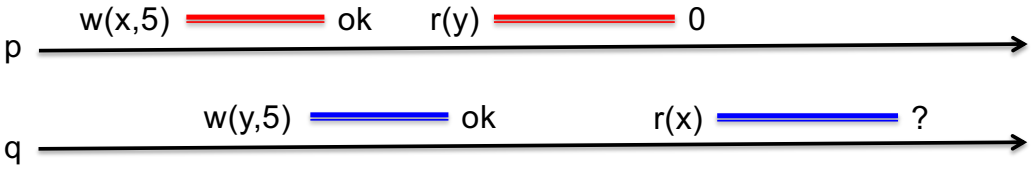
Still have to make sense

Assume x and y is initially set to 0



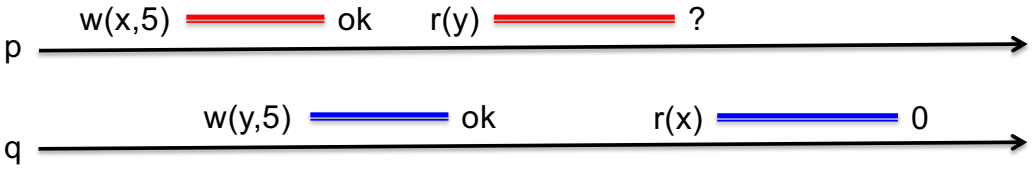
Still have to make sense

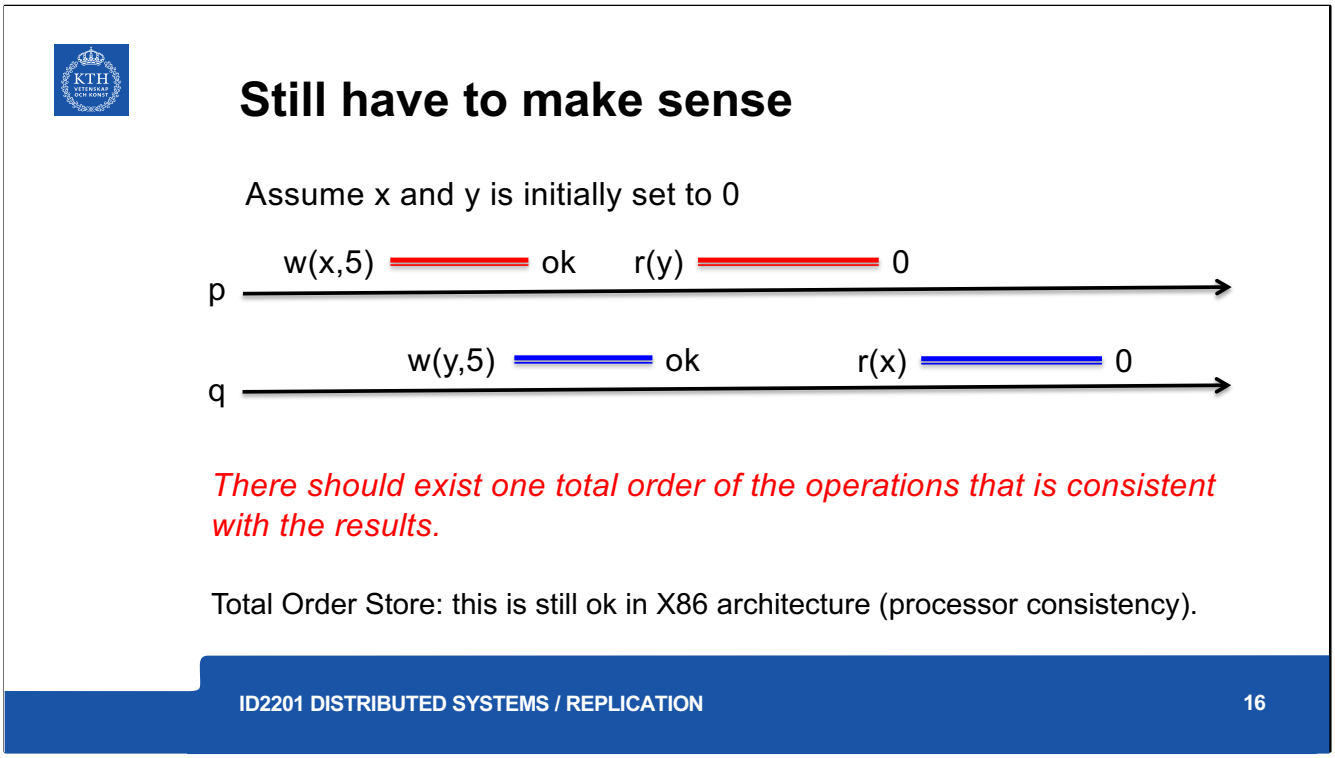
Assume x and y is initially set to 0



Still have to make sense

Assume x and y is initially set to 0

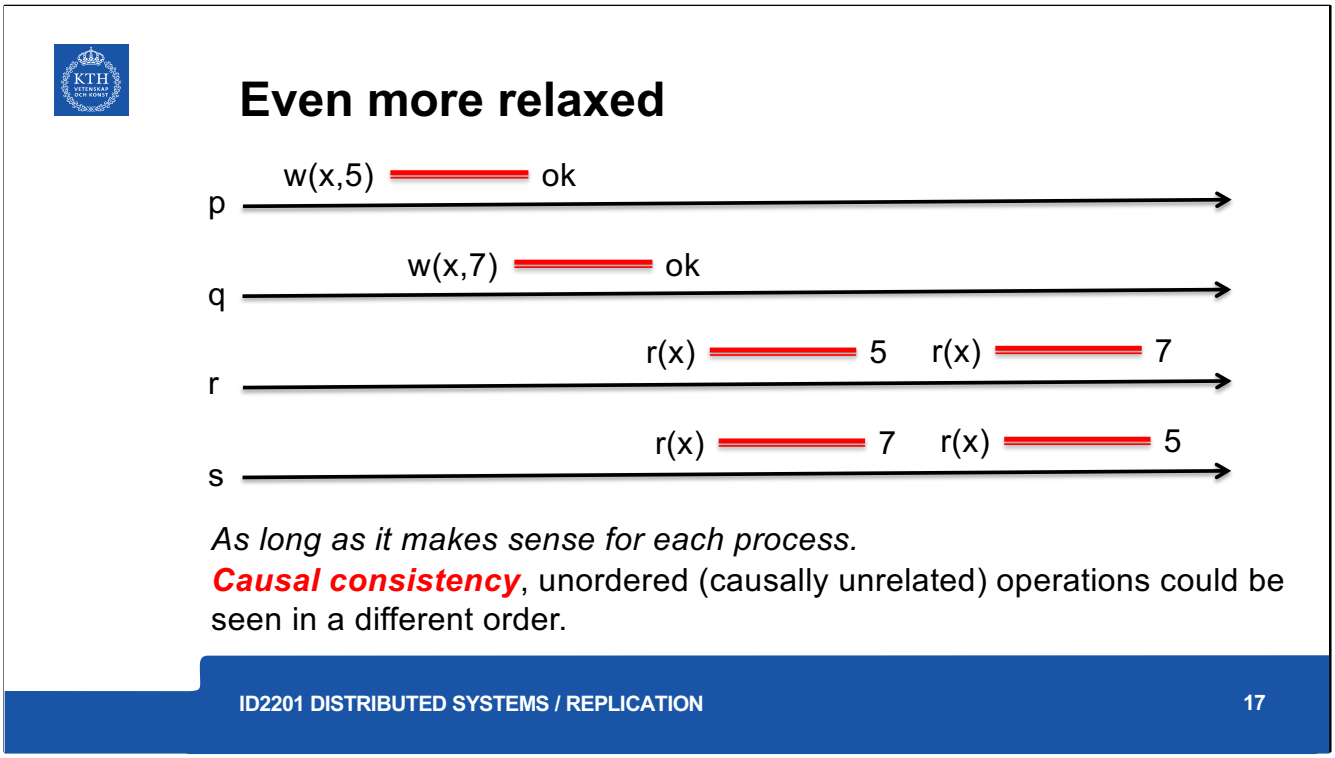




Operations are executed according to the programming order within a process but what about across processes?

Every linearizable service is also sequentially consistent, since real-time order reflects each client's program order. The converse does not hold.

Processor Consistency (in X86) Wikipedia: "One of the main components of processor consistency is that if a write followed by a read is allowed to execute out of program order. This essentially results in the hiding of write latency when loads are allowed to go ahead of stores. Since many applications function correctly with this structure, systems that implement this type of relaxed ordering typically appear sequentially consistent."



Make sense for each process, but what about across processes?

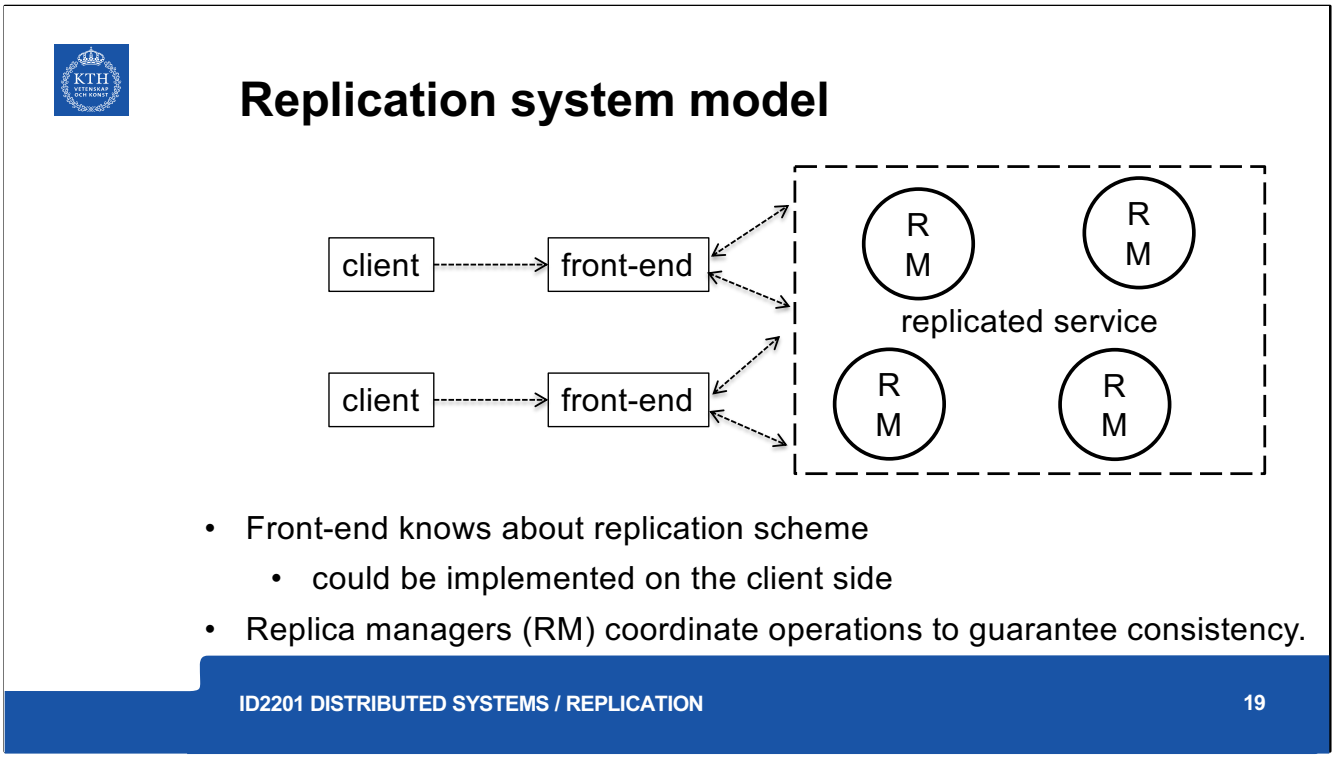
Wikipedia: **Causal consistency** captures the potential causal relationships between operations and guarantees that all processes observe causally-related operations in a common order. In other words, all processes in the system agree on the order of the causally-related operations. They may disagree on the order of causally unrelated operations. … The Causal Precedence relation in shared memory is related to the happened-before relation in message-based communication.

In this example, reading processes see writes in a different order as those writes are causally unrelated.

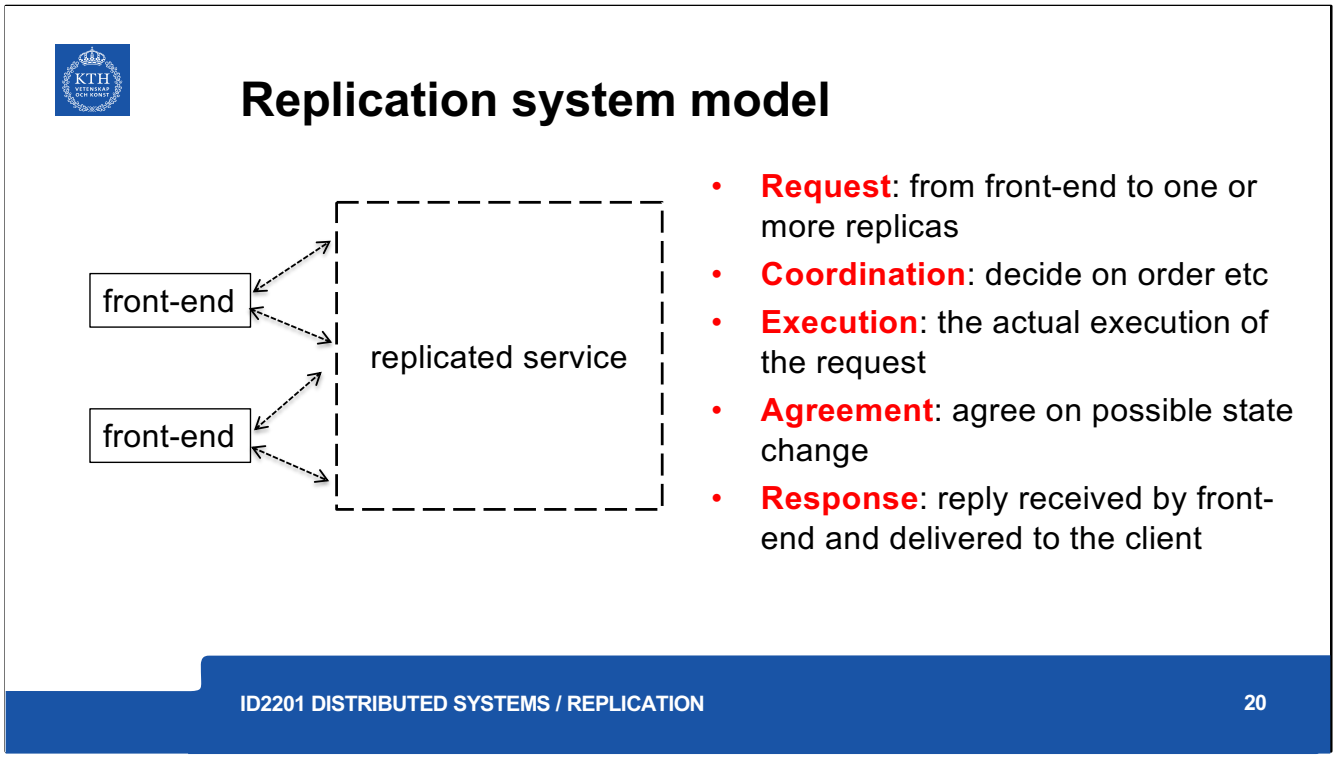
Eventual consistency

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

More on this later.



Each client requests a series of operations – invocations upon one or more objects: reads or updates. Each client's requests are first handled by ***a front end*** that communicates by message passing with one or more of the ***replica managers*** (RM). A front end may be implemented in the client's address space, or it may be a separate process.

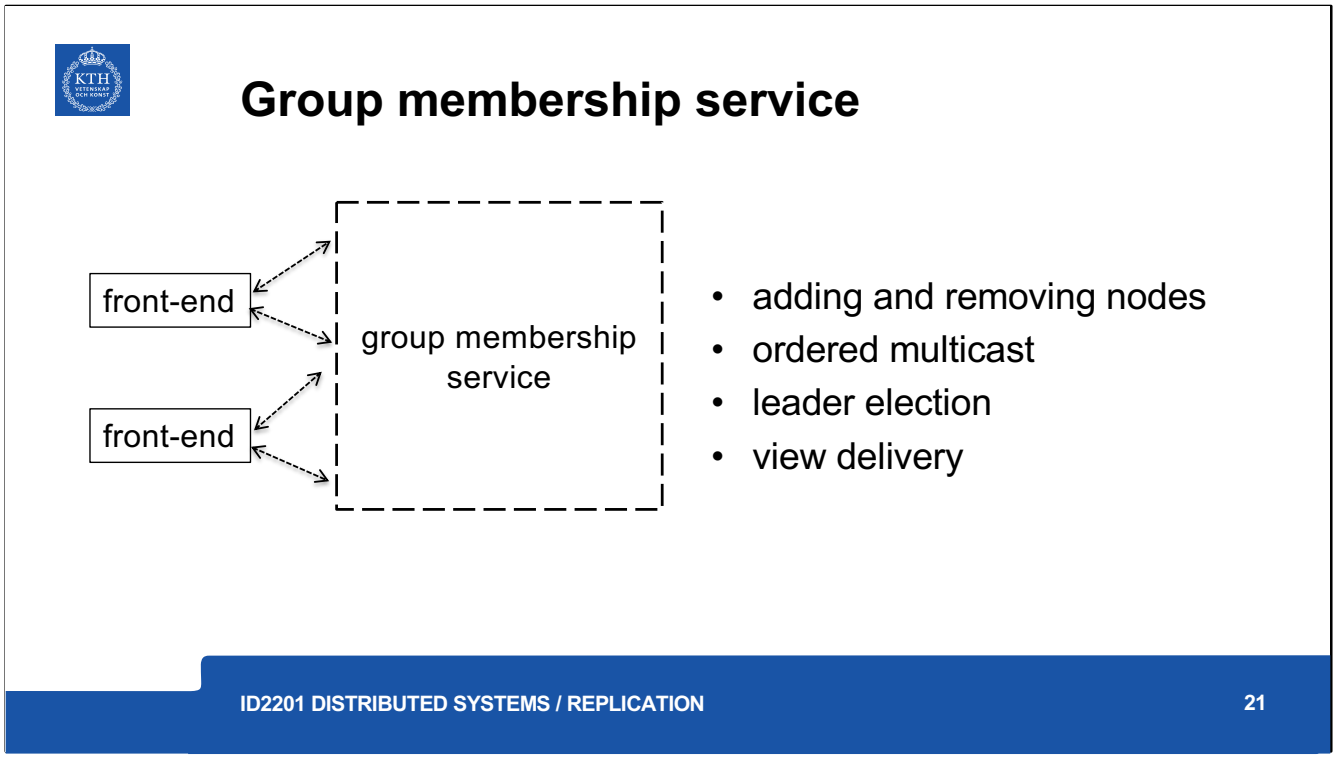


Five phases in performing a request: Request, Coordination, Execution, Agreement, and Response.

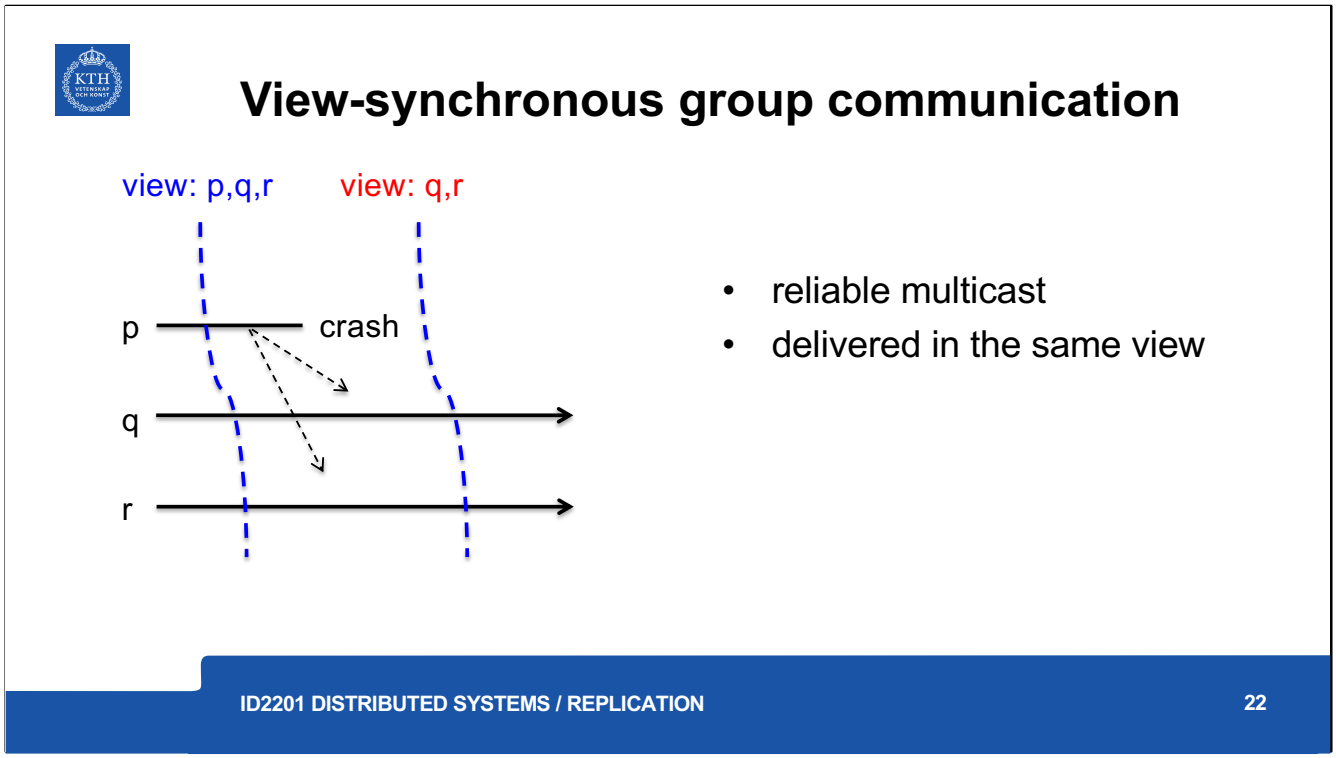
FIFO ordering (most used): If a front-end issues request r and then request r', any correct replica manager that handles r' handles r before it.

Causal ordering: If the issue of request r *happened before* request r', then any correct replica manager that handles r' handles r before it.

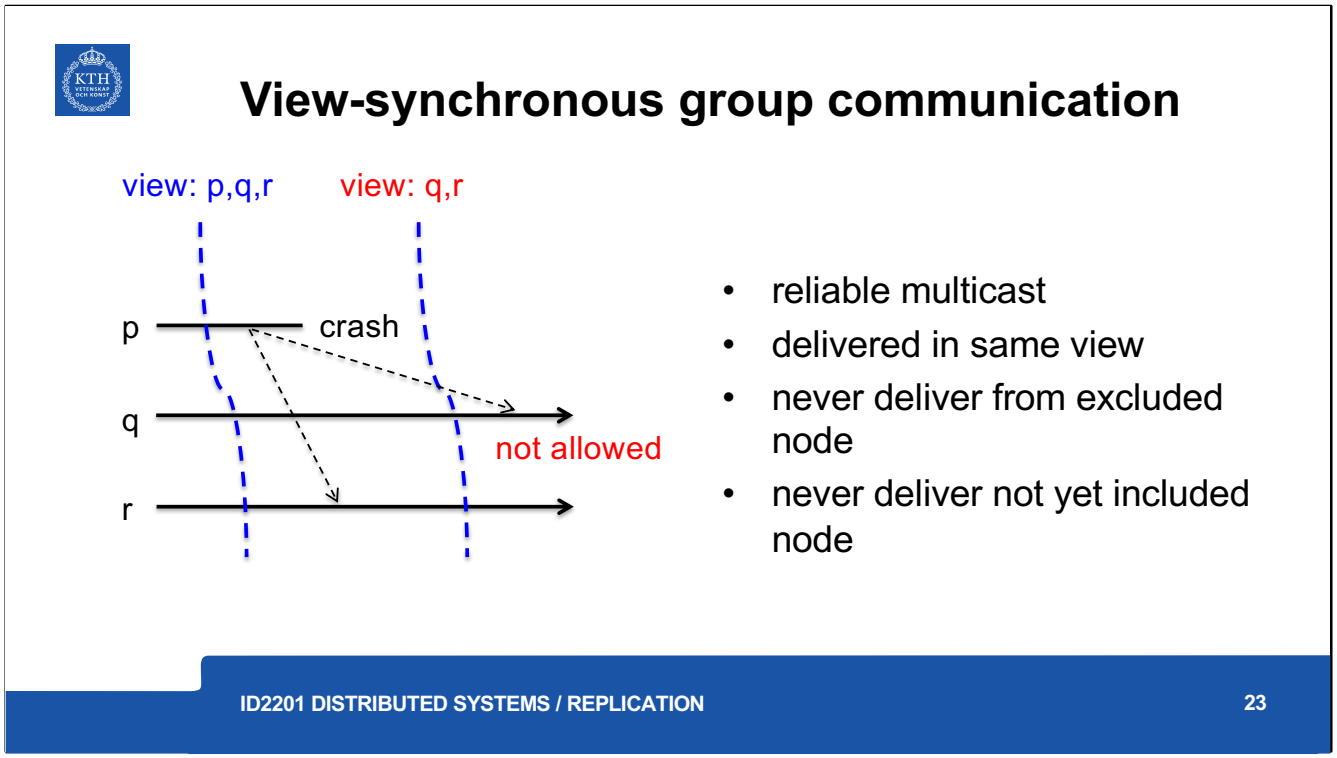
Total ordering: If a correct replica manager handles r before request r', then any correct replica manager that handles r' handles r before it.



Group membership management is essential for the service that manages replicated data.
A group membership service maintains ***group views***, which are lists of the current group members identified by their unique PIDs.
View delivery: For each group, the group management service delivers a new group view to any member whenever a membership change occurs. A new group view is generated each time a process is added or excluded. A group membership service may exclude a process from a group because it is ***Suspected*** to be crashed. That process will have to rejoin the group or abort its operations.



For each group, **the group management service delivers a new group view to any member** whenever a membership change occurs. A new group view is generated each time a process is added or excluded.
View-synchronous communication extends the reliable multicast semantics to account for changing group views.
Agreement: Correct processes deliver the same sequence of views and the same set of messages in any given view. That is, if a correct process delivers message m in view v(g), then all other correct processes that deliver m also do so in the view v(g).



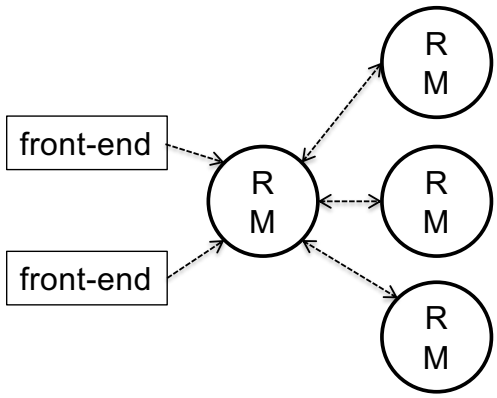
View-synchronous communication extends the reliable multicast semantics to account for changing group views.

Agreement: Correct processes deliver the same sequence of views and the same set of messages in any given view. That is, if a correct process delivers message m in view v(g), then all other correct processes that deliver m also do so in the view v(g).

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*.

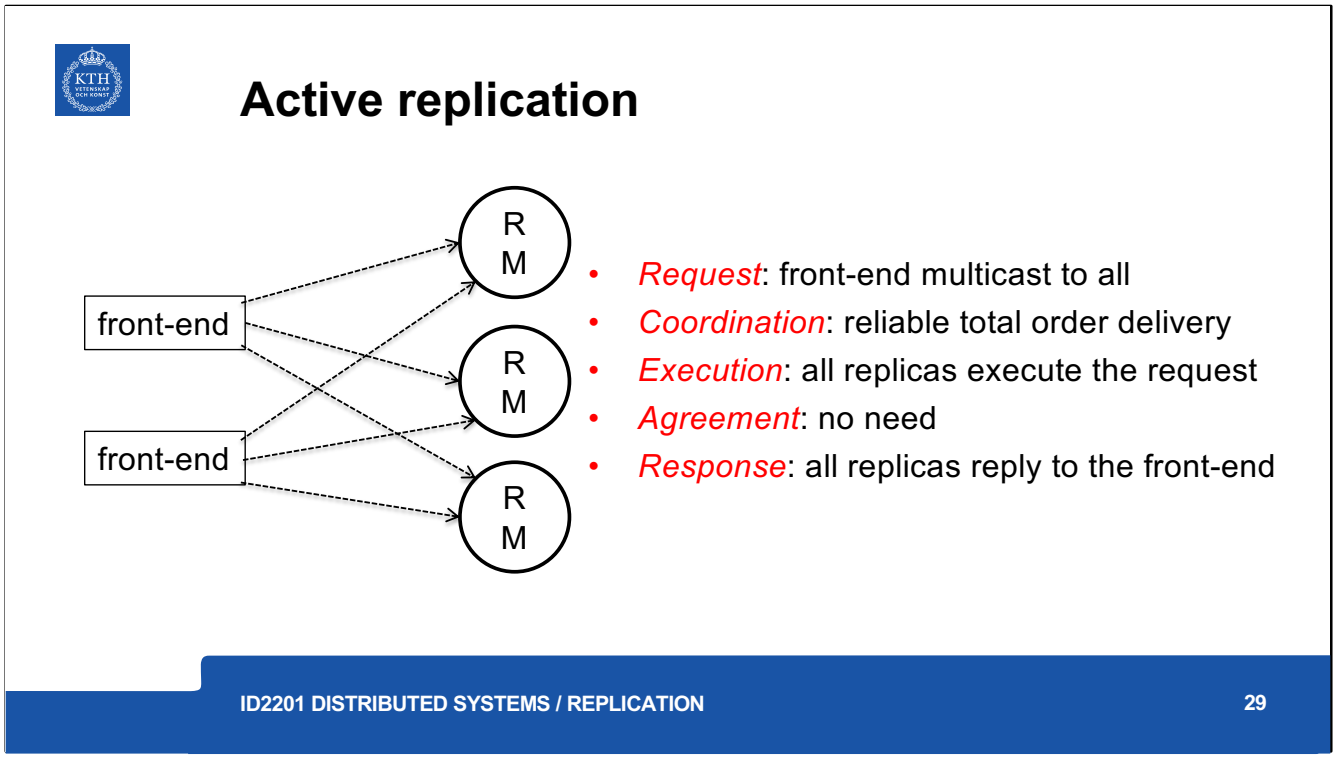
Passive replication – Pros and cons

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.



Response: Each replica manager sends its response to the front end. The number of replies that the front-end collects depends on failure assumptions and the multicast algorithm. If, for example, the goal is to tolerate only crash failures and the multicast satisfies uniform agreement and ordering properties, then the front end passes the first response to arrive back to the client and discards the rest (it can distinguish these from responses to other requests by examining the identifier in the response).

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.

The active replication system does not achieve linearizability. This is because the total order in which the replica managers process requests is not necessarily the same as the real-time order in which the clients made their requests.

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.

Byzantine failures are arbitrary deviations of a process from its assumed behavior based on the algorithm it is supposed to run and the inputs it receives. Such failures can occur, e.g., due to a software bug, a (transitional or permanent) hardware malfunction, or a malicious attack. **Active replication can tolerate Byzantine failures because the front end can collect and compare the replies it receives.**

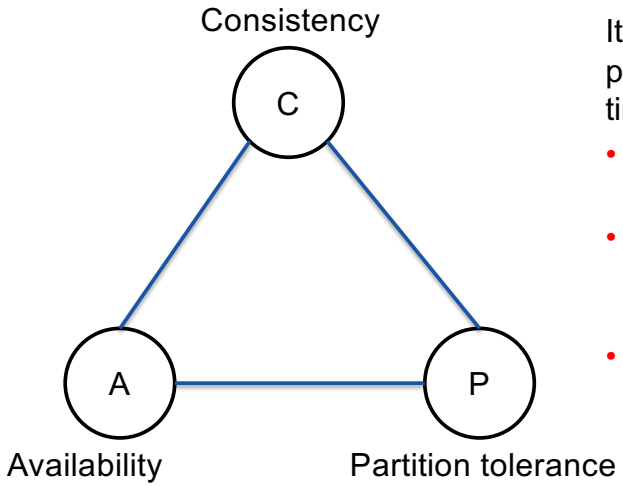
Availability

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.

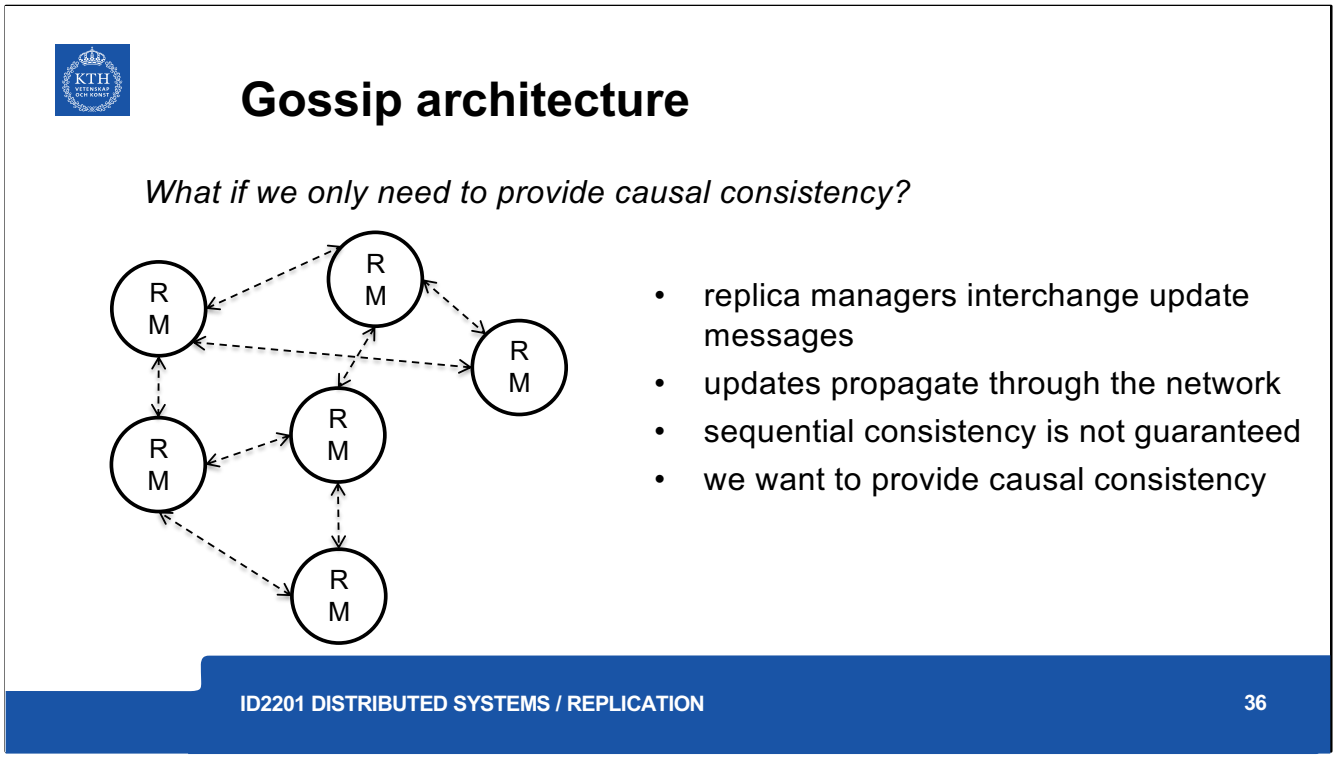
When the system re-connects: merge operations are performed in separate partitions.

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 guarantees: ***Each client obtains a consistent service over time***: In answer to a query, RMs only provide a client with data that reflects at least the updates the client has observed. ***Relaxed consistency between replicas***: All RMs eventually receive all updates and apply updates with ordering guarantees that make the replicas sufficiently similar to suit the application's needs. To support relaxed consistency, ***gossip supports causal update ordering***. It also ***supports stronger ordering guarantees in the form of forced (total and causal) and immediate ordering.***

Gossip allows clients to make updates to local replicas while partitioned. RMs exchange updates with one another when they become reconnected. Gossip provides its highest availability at the expense of relaxed, causal consistency.

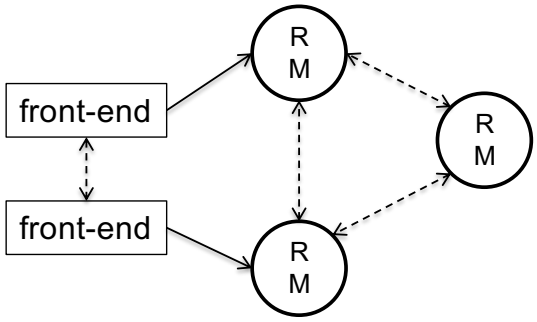
Vector clocks

A vector clock with one index per replica manager.

Each update will be tagged with a vector clock timestamp.

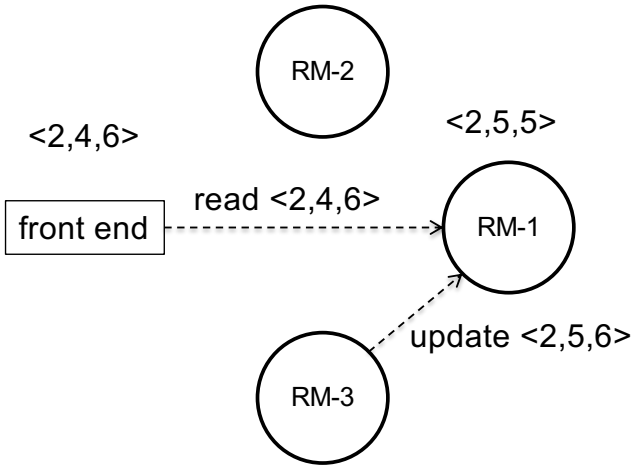
Some updates are concurrent!

The front end



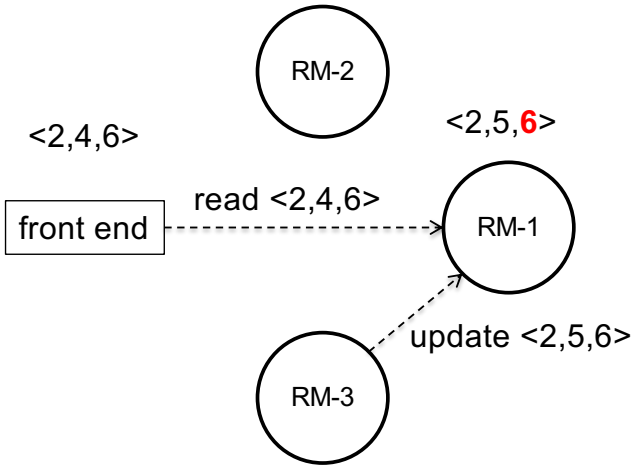
- one index per replica manager
- front-ends keep vector clocks
- replica managers apply updates in order
- causal consistency guaranteed

The front end



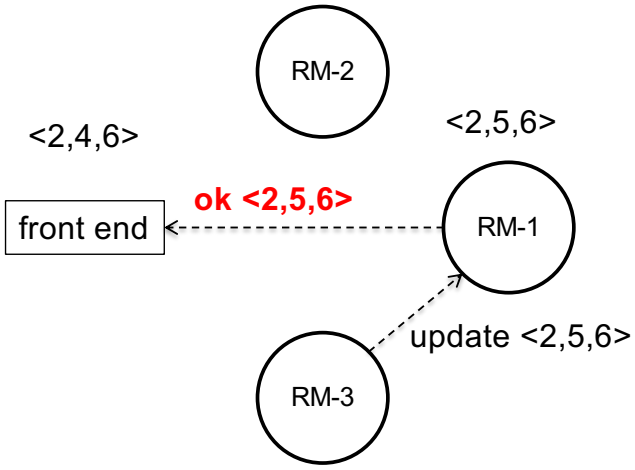
- send a query with a timestamp
- check the current time, wait for updates
- update will arrive

The front end



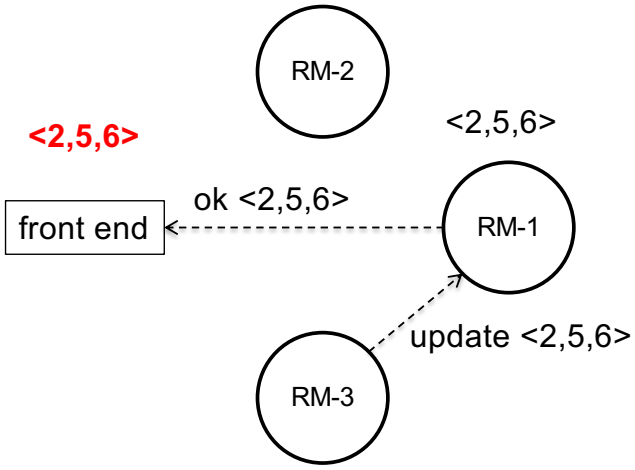
- send a query with a timestamp
- check the current time, wait for updates
- update will arrive
- **update state and clock**

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



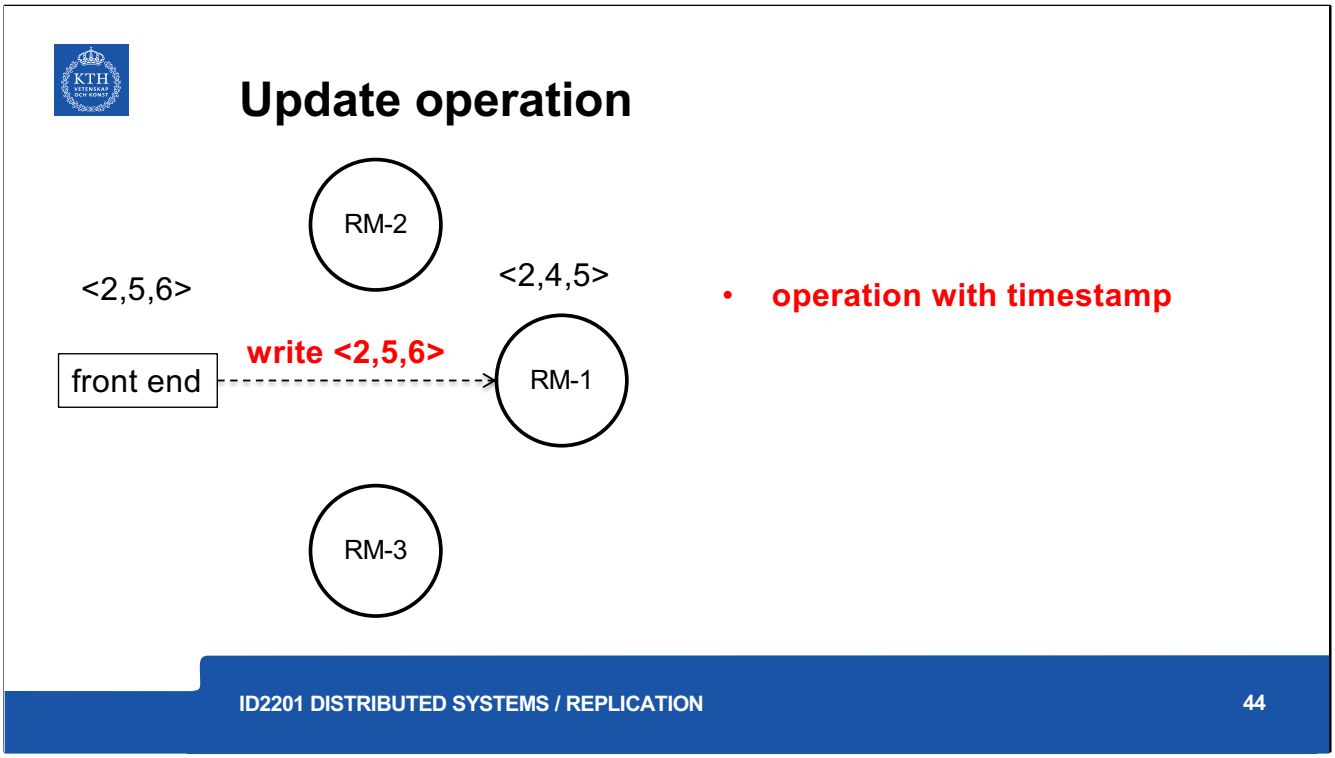
- send a query with a timestamp
- check the current time, wait for updates
- update will arrive
- update state and clock
- reply
- **update state and clock**

The replica manager

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

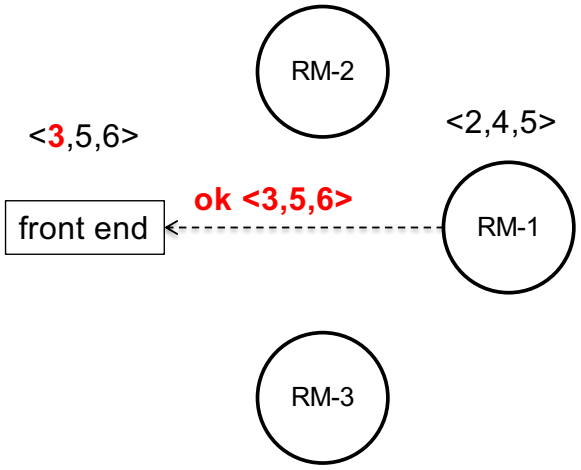
As updates arrive, the replica will execute updates and pending read operations.

Updates are partially ordered.



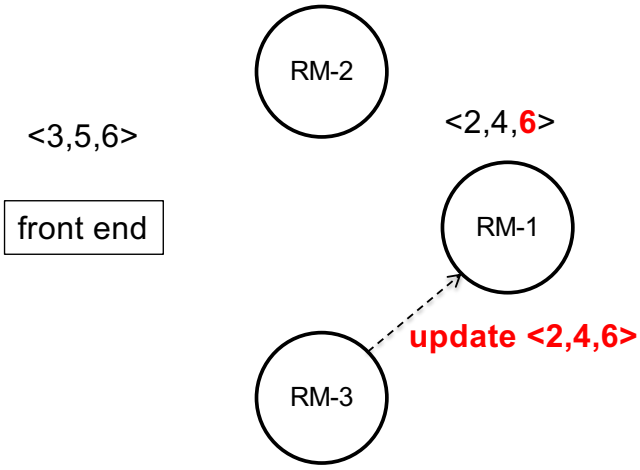
Update <2,5,6> is not stable yet because the value time stamp <2,4,5> is smaller than the timestamp of the requested update, i.e., the previous updates done at replicas 2 and 3 have not been applied yet.

Update operation



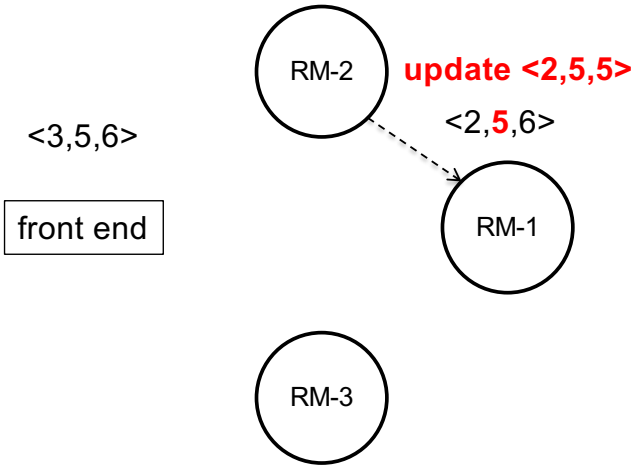
- operation with timestamp
- **reply with a unique timestamp**

Update operation

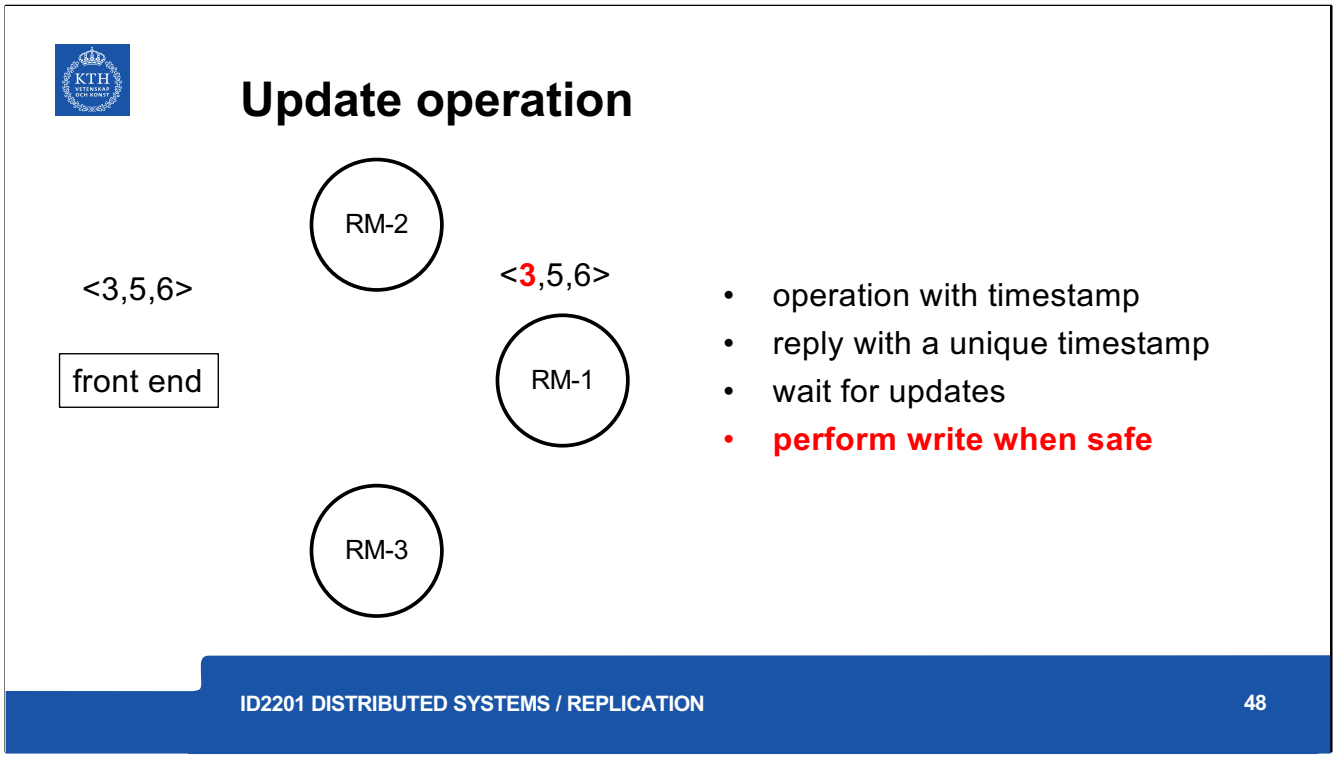


- operation with timestamp
- reply with a unique timestamp
- **wait for updates**

Update operation



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



Now the held (waiting) update request <2,5,6> is stable and can be performed.

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.

This condition states that all the updates on which this update depends – that is, all the updates observed by the front end that issued the update – have already been applied to the value. If this condition is not met when the update is submitted, it will be rechecked when gossip messages arrive. When the stability condition has been met for an update record r, the replica manager applies the update to the value and updates the value timestamp and the executed operation table, executed

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.

Forced updates are totally and causally ordered using unique sequence numbers for updates. One option is to use a single sequencer process. The better solution is to designate a so-called primary replica manager as the sequencer.
Immediate updates are ordered with respect to forced updates by the primary replica, which also determines which causal updates are deemed to have preceded an immediate update. It does this by communicating and synchronizing with the other replica managers to reach an agreement.

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

ID2201 Distributed Systems

Lecture resumes 14:15