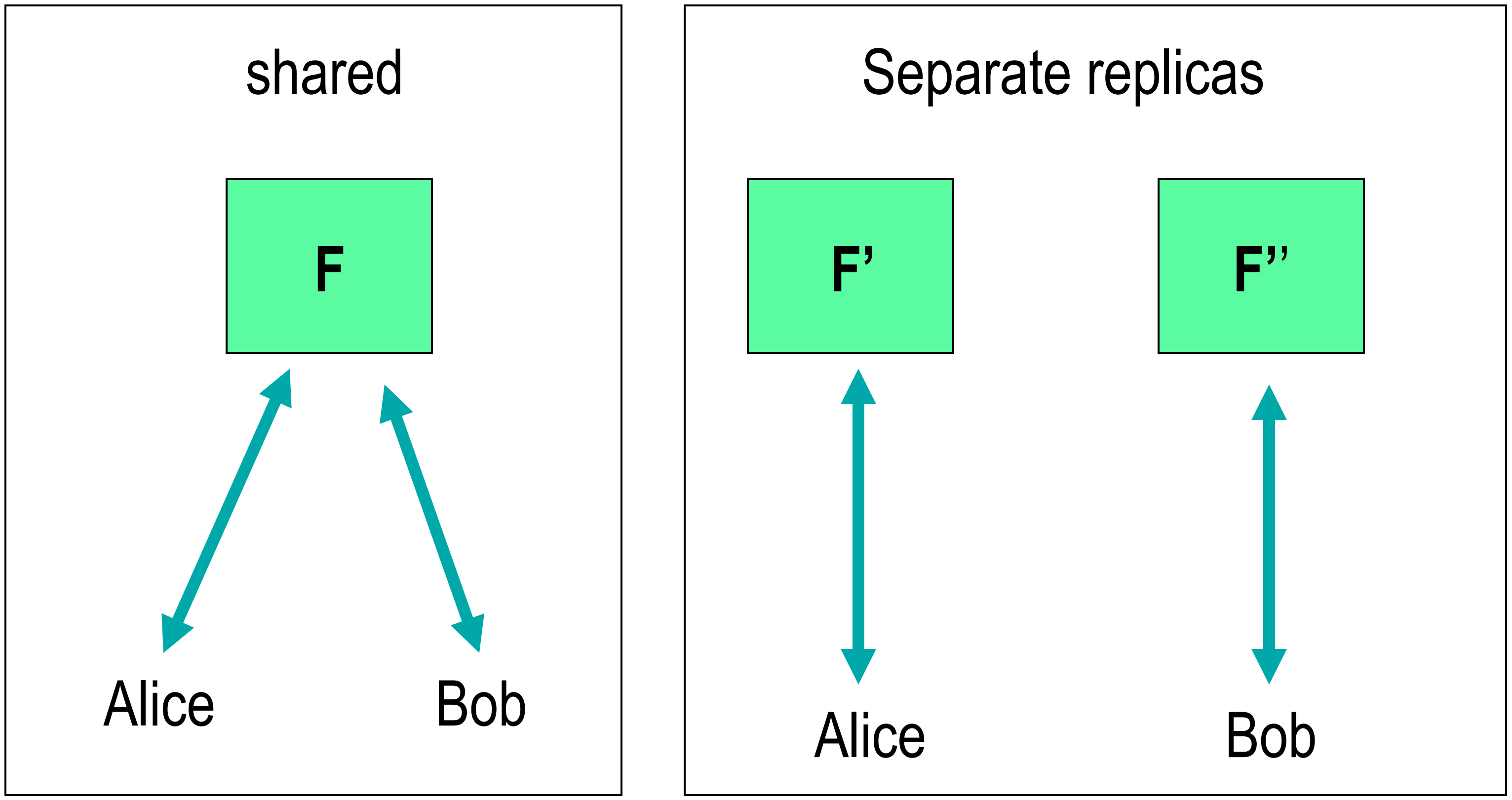
Replication

- **Improves reliability**
- **Improves availability**

  (*What good is a reliable system if it is not available?)*

- Replication must be **transparent** and create the illusion of a single copy.

Updating replicated data



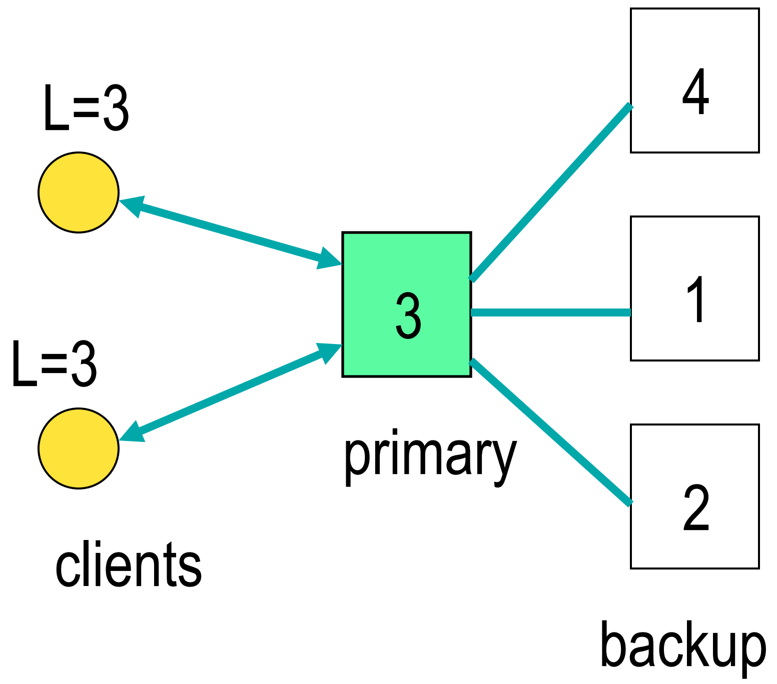
Update and consistency are primary issues.

Passive replication

At most one replica can be the primary server

Each client maintains a variable **L** (leader) that specifies the replica to which it will send requests. Requests are queued at the primary server.

Backup servers ignore client requests.

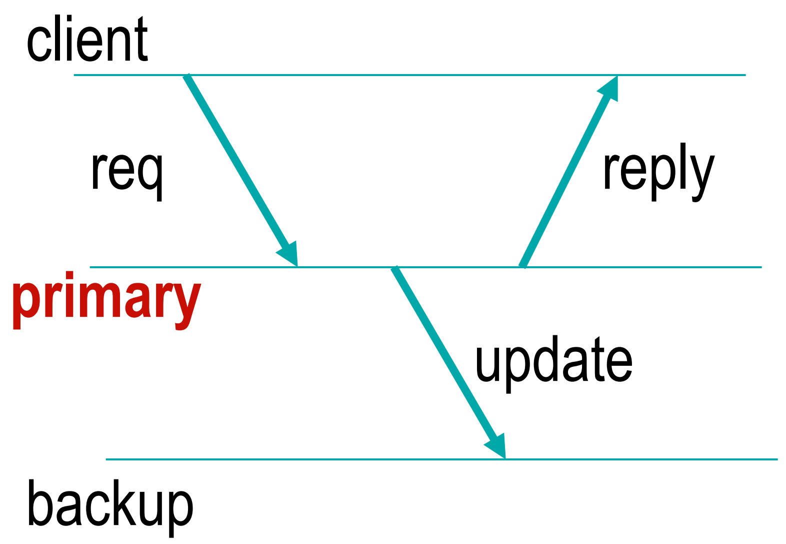


Primary-backup protocol

Receive. Receive the request from the client and update the state if appropriate.

Broadcast. Broadcast an update of the state to all other replicas.

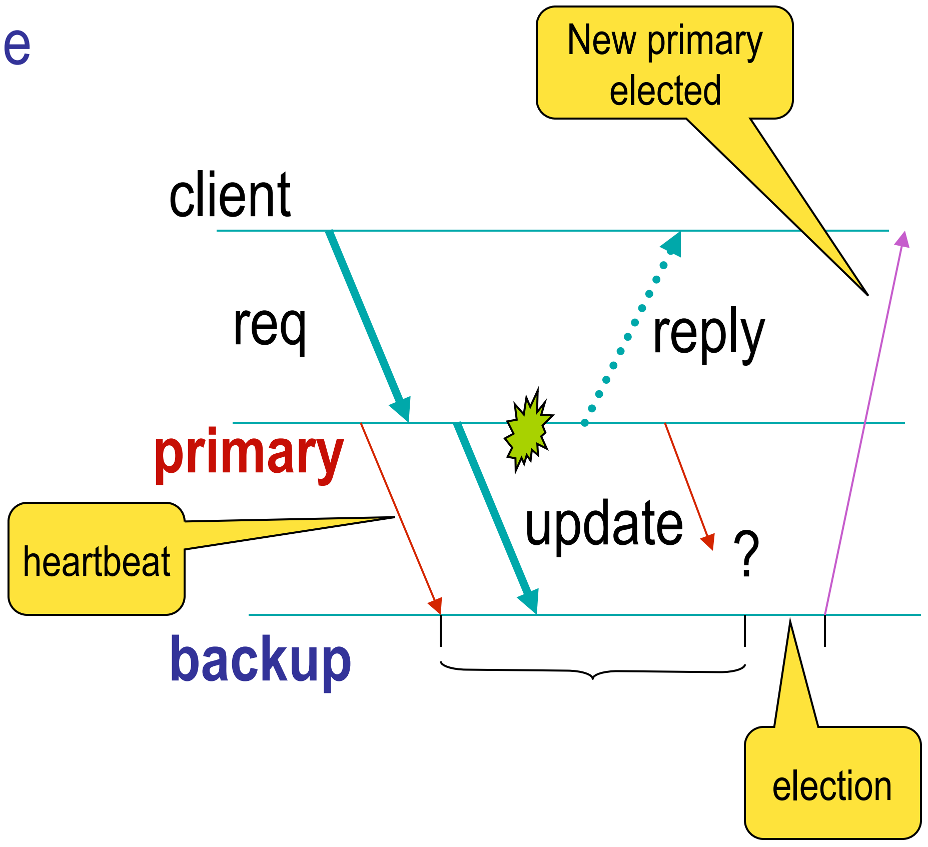
Reply. Send a response to the client.



Primary-backup protocol

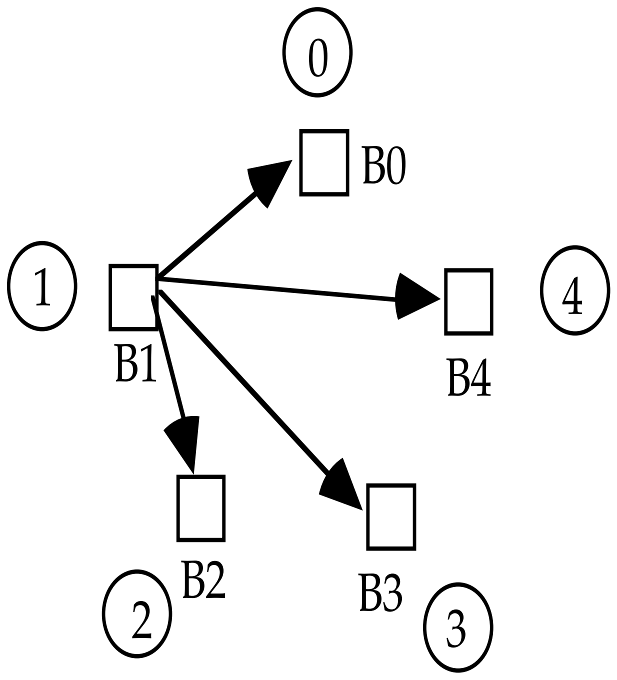
If the client fails to get a response due to the crash of the primary, then the request is retransmitted until a backup is promoted as the primary.

Failover time is the duration when there is no primary server.



Active replication

Each server receives client requests, and broadcasts them to the other servers. They collectively implement a ***fault-tolerant state machine***. In presence of crash, all the correct processes reach the same next state.



$$\text{State} \xrightarrow{\text{input}} \text{Next state}$$

Fault-tolerant state machine

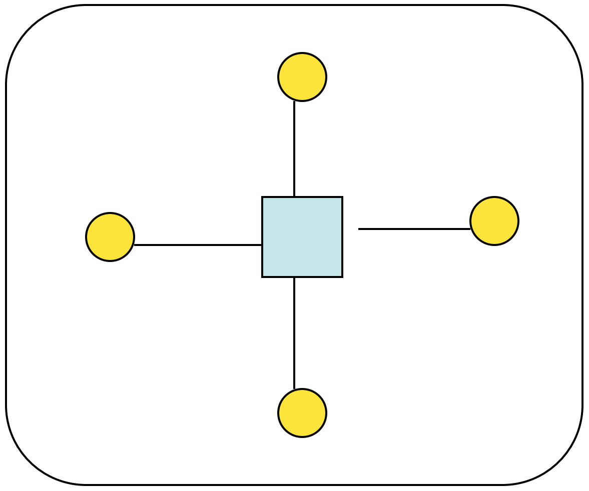
This formalism is based on a survey by Fred Schneider.

The clients must receive correct response **even if up to m replica servers fail (either fail-stop or byzantine).**

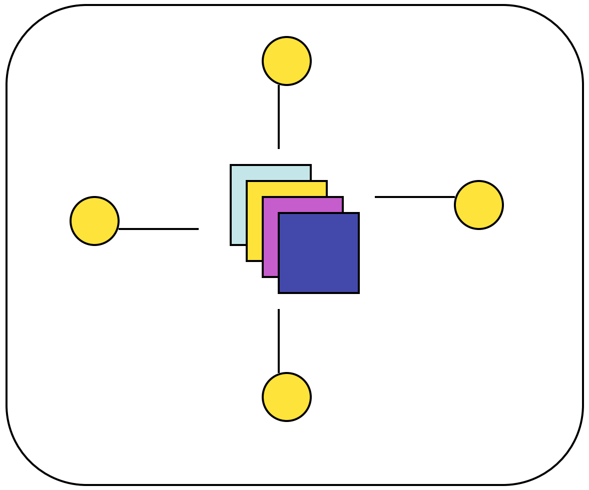
For **fail-stop**, **≥ (m+1) replicas** are needed. If a client queries the replicas, the first one that responds gives a correct value.

For **byzantine** failure **≥ (2m+1) replicas** are needed. **m** bad responses can be voted out by the **(m+1)** good responses.

But the states of the good processes must be correctly updated



Fault intolerant



Fault tolerant

Replica coordination

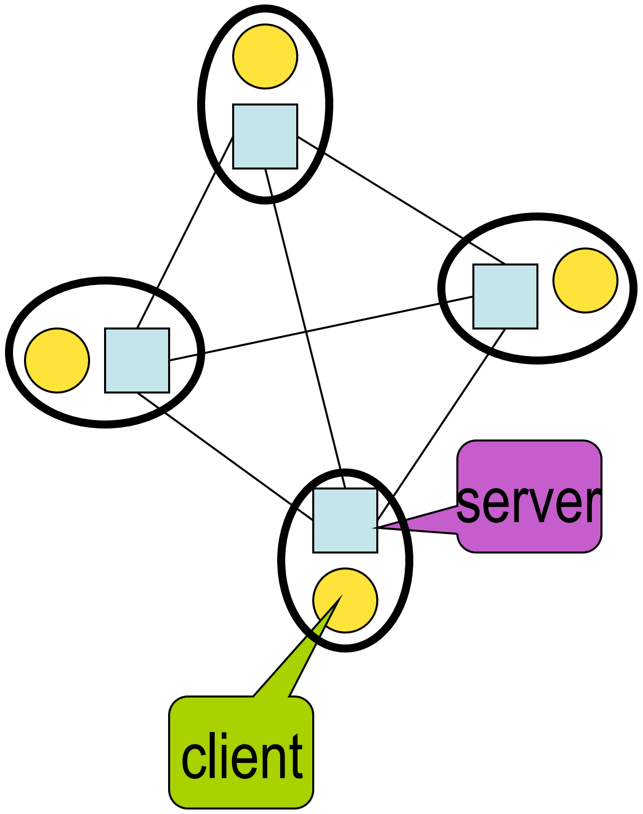
Agreement. Every correct replica receives all the requests.

Order. Every correct replica receives the requests in the same order.

Agreement part is solved by atomic multicast.

Order part is solved by total order multicast.

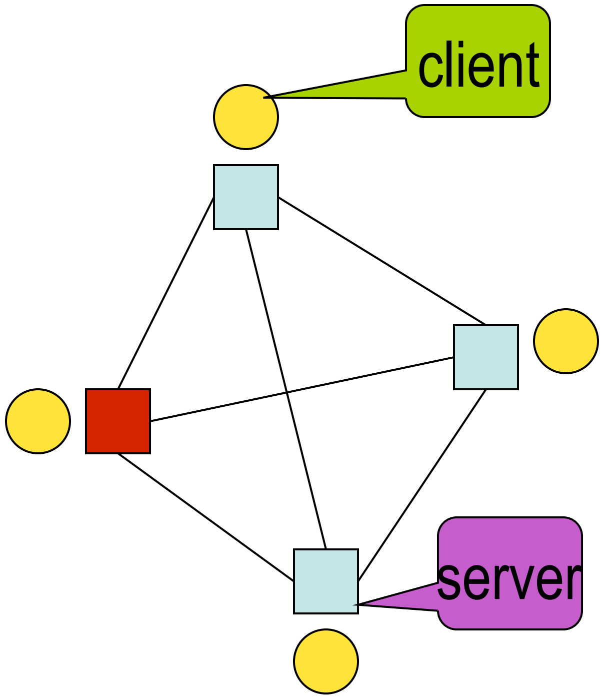
The order part solves the **consensus problem** where servers will agree about the ***next update***. It requires a synchronous model. Why?



Agreement

With fail-stop processors, the agreement part is solved by *reliable* atomic multicast.

To deal with byzantine failures, an interactive consistency protocol needs to be implemented. Thus, with an oral message protocol, $n \geq 3m+1$ processors will be required.

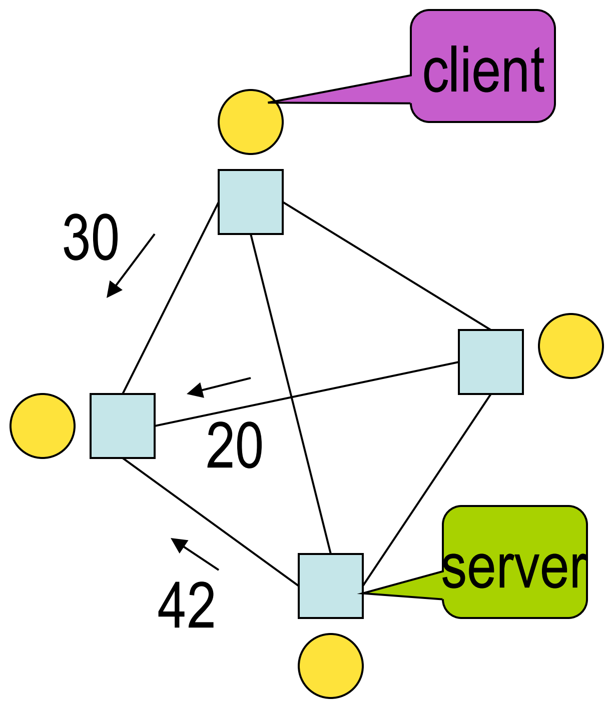


Order

Let **timestamps** determine the message order.

A request is stable at a server, when the it does not expect to receive any other client request with a lower timestamp.

Assume three clients are trying to send an update, the channels are FIFO, and their timestamps are 20, 30, 42. Each server will first update its copy with the value that has the timestamp 20.

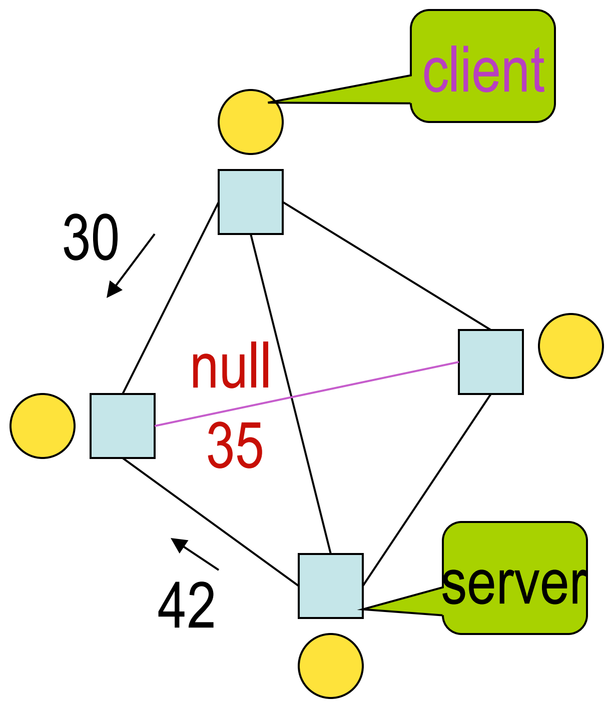


Order

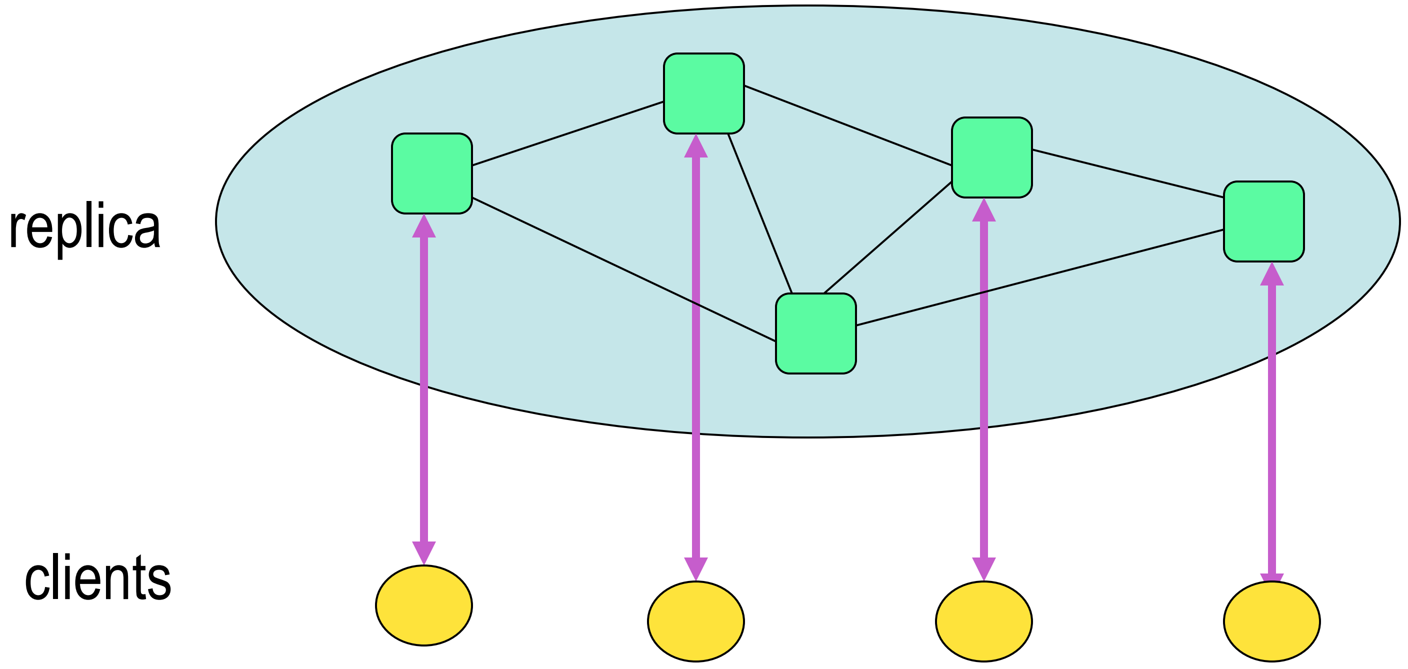
But some clients may not have any update.
How long should the server wait?

Require clients to send null messages (as heartbeat signals) with some timestamp ts. A message (null, 35) means that the client **will not** send any update till ts=35. These can be part of *periodic heartbeat messages*.

An alternative is to use **virtual time**, where processes are able to undo actions.



What is replica consistency?



Consistency models define a contract between the data manager and the clients regarding the responses to read and write operations.

Replica Consistency

- Data Centric

  Client communicates with the same replica

- Client centric

  Client communicates with different replica at different times. This may be the case with mobile clients.

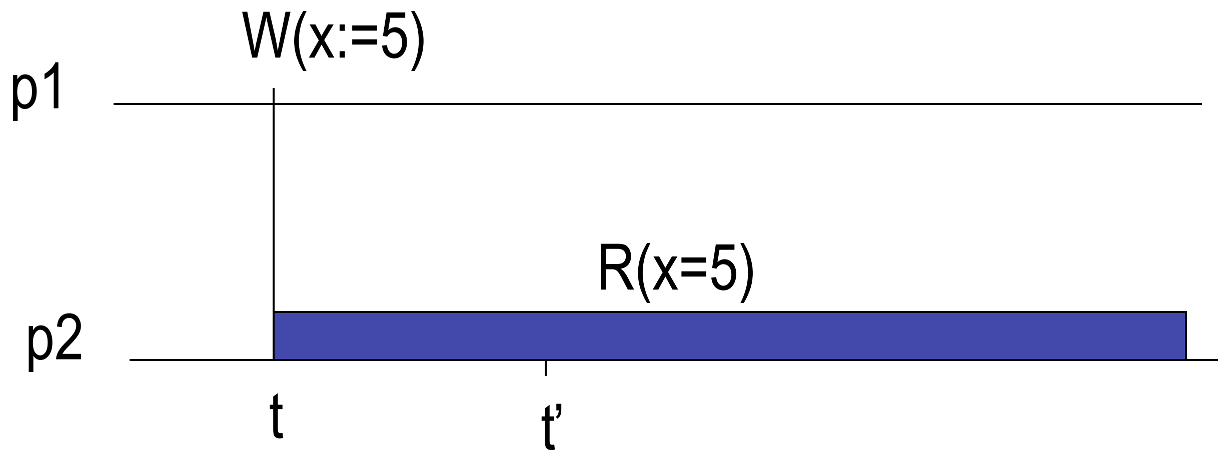
Data-centric Consistency Models

1. **Strict consistency**
2. **Linearizability**
3. **Sequential consistency**
4. **Causal consistency**
5. **Eventual consistency (as in DNS)**
6. **Weak consistency**

There are many other models

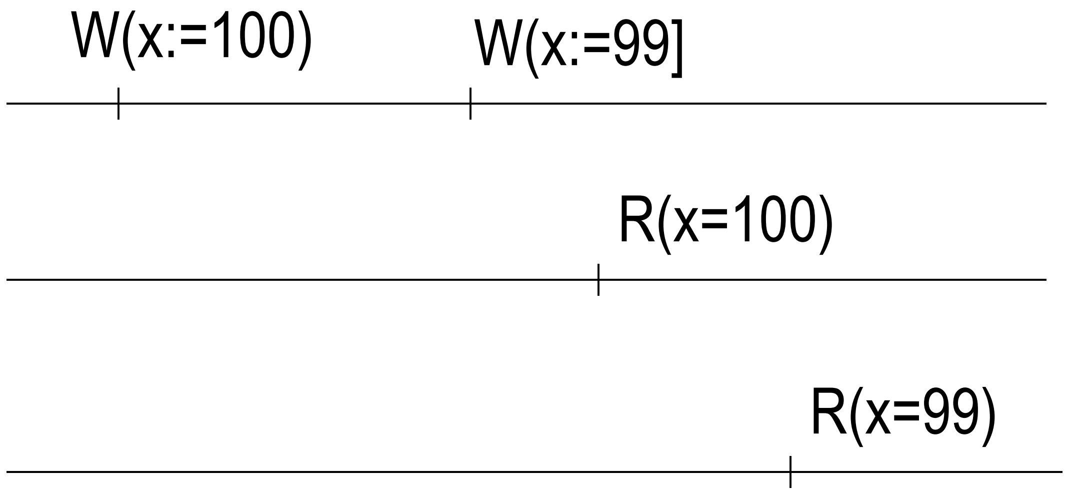
Strict consistency

Strict consistency corresponds to true replication transparency. If one of the processes executes x:= 5 at real time t and this is the latest write operation, then at a real time t' > t, every process trying to read x will receive the value 5. Too strict! Why?



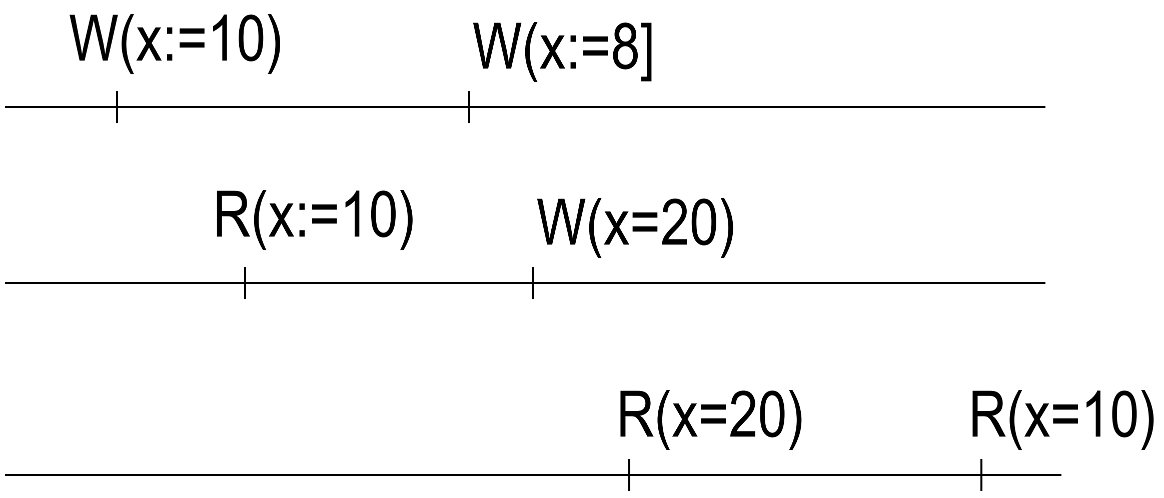
Sequential consistency

Some interleaving of the local temporal order of events at the different replicas is a consistent trace.



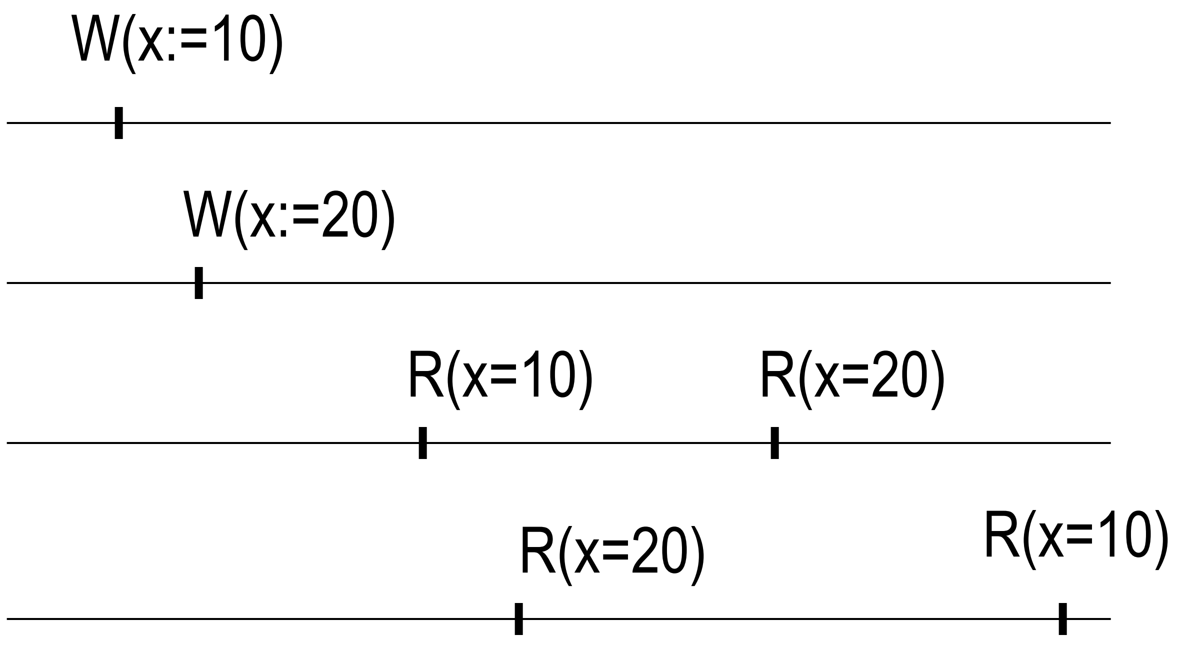
Sequential consistency

Is sequential consistency satisfied here? Initially x = y = 0



Causal consistency

All writes that are *causally related* must be seen by every process in the same order.

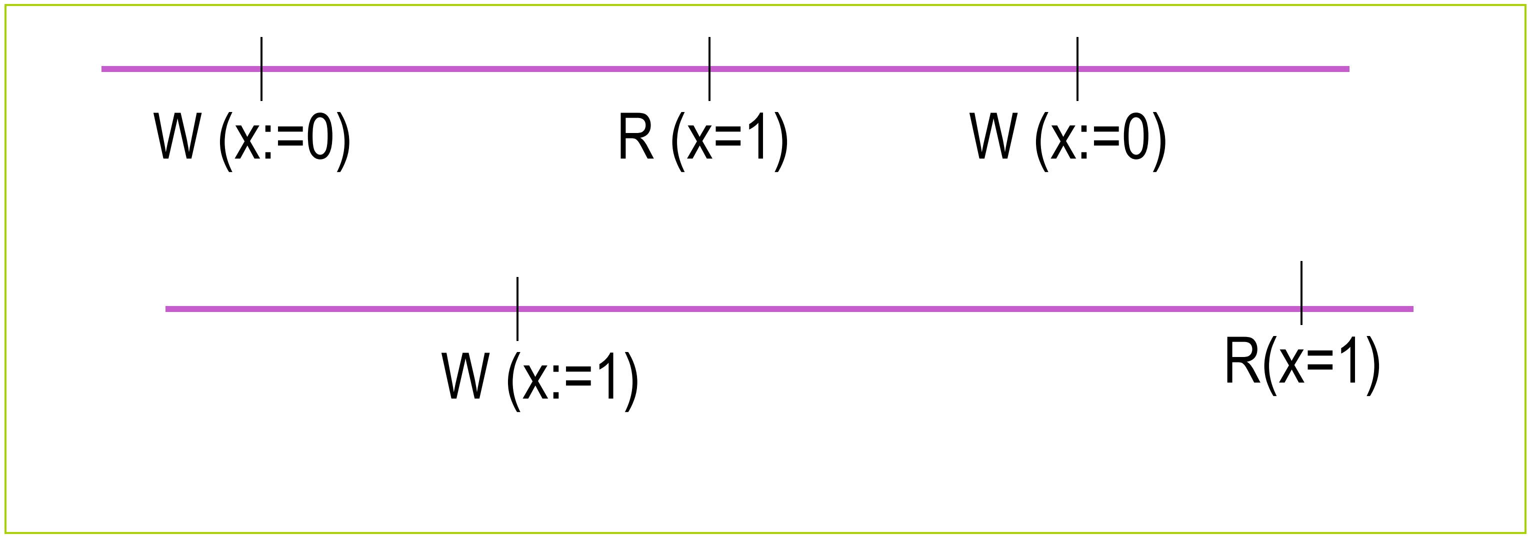


Linearizability

Linearizability is a correctness criterion for *concurrent object* (Herlihy & Wing ACM TOPLAS 1990). It provides the *illusion* that each operation on the object takes effect in zero time, and the results are “equivalent to” some legal sequential computation.

Linearizability

A trace is in a read-write system is *consistent*, when every read returns the *latest* value written into the shared variable preceding that read operation. A trace is **linearizable**, when (1) it is consistent, and (2) the **temporal ordering** among the reads and writes is respected (may be based on real time or logical time).

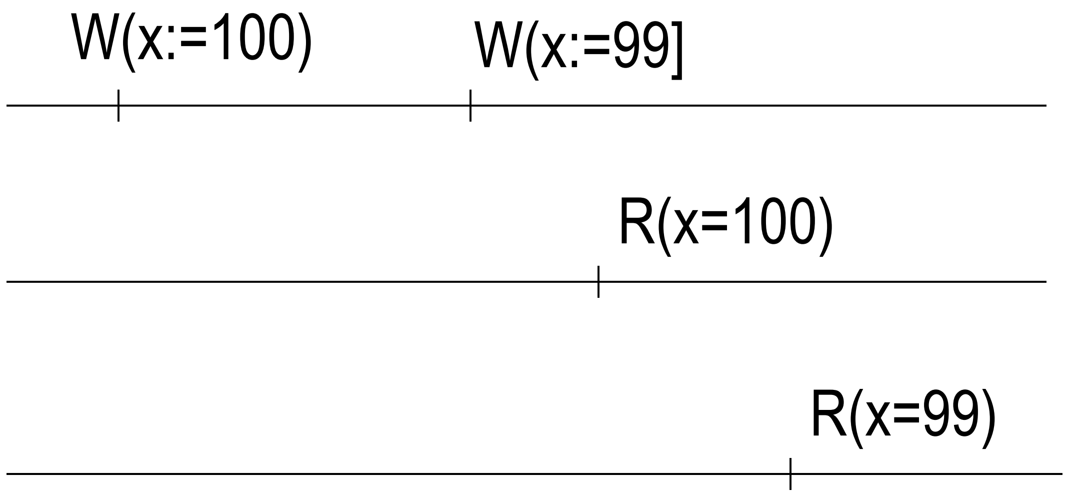


(Initially x=y=0)

Is it a linearizable trace?

Sequential consistency

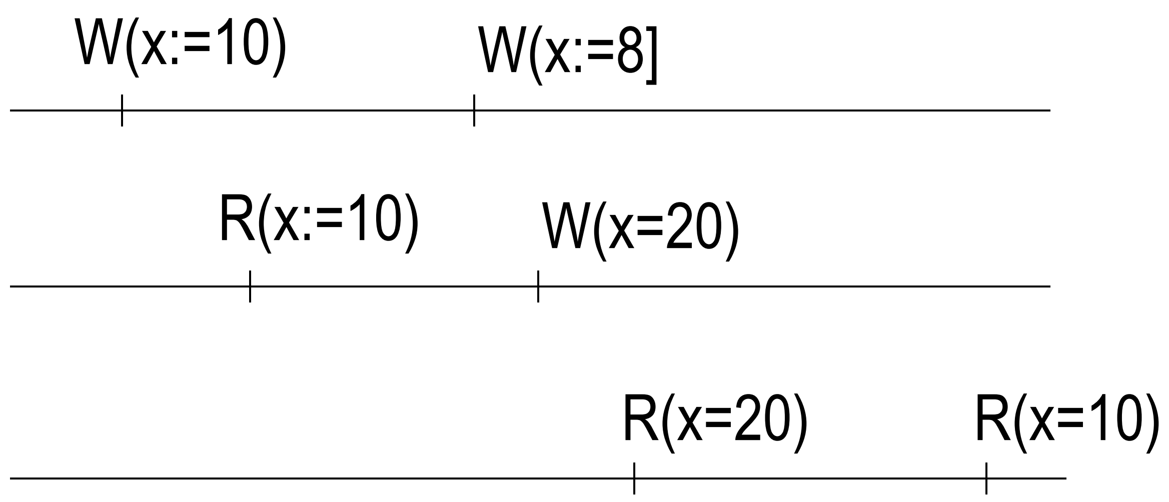
Some interleaving of the local temporal order of events at the different replicas is a consistent trace.



Is this a linearizable trace?

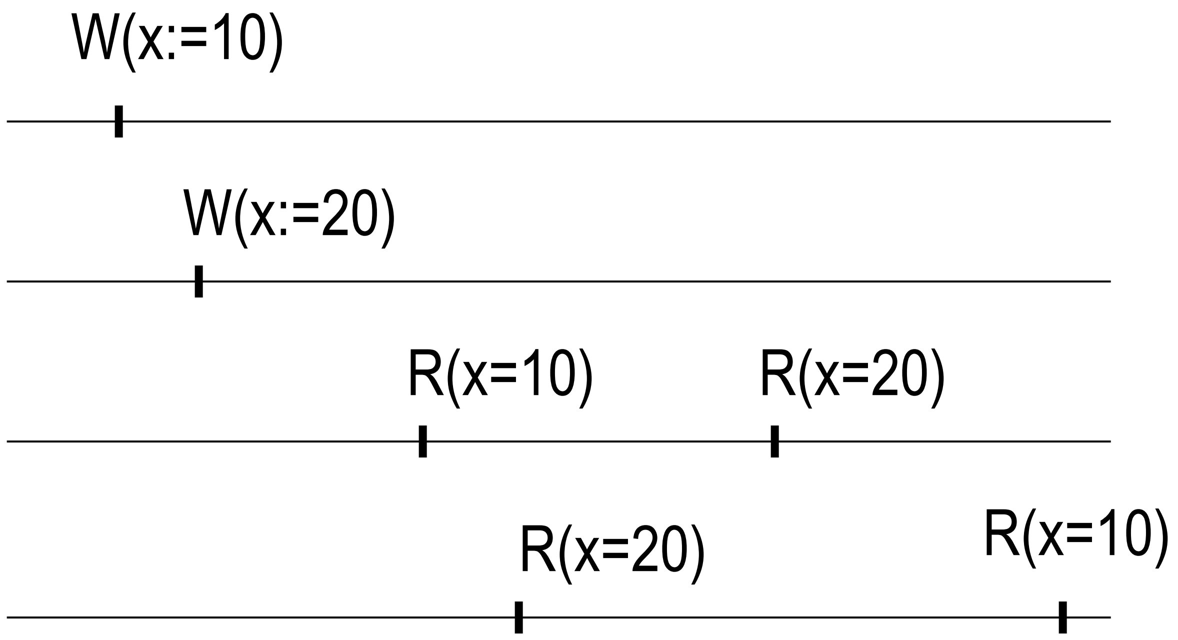
Sequential consistency

Is sequential consistency satisfied here? Assume that initially x=y=0.



Causal consistency

All writes that are *causally related* must be seen by every process in the same order.



Does this satisfy sequential consistency of linearizability?

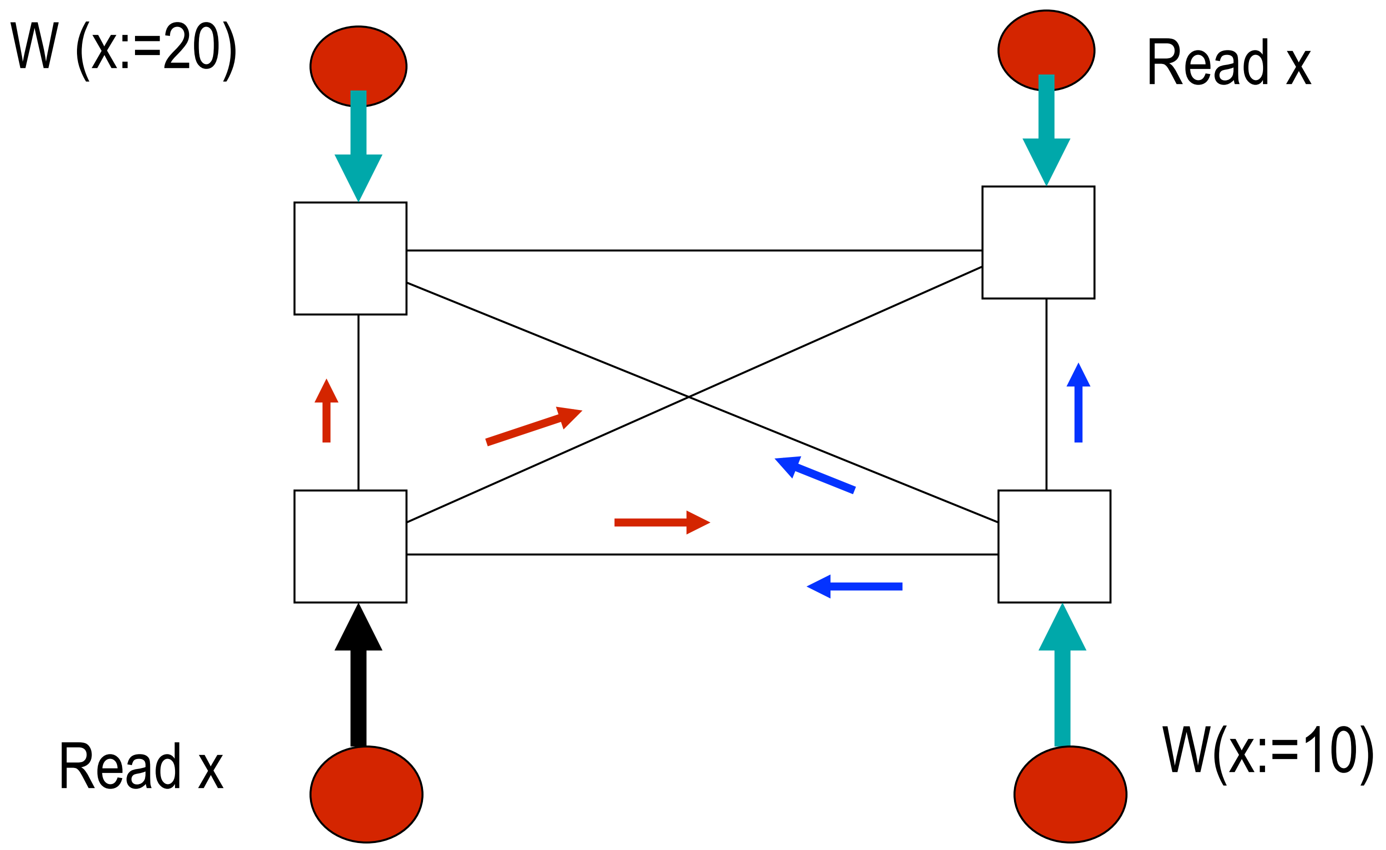
Implementing consistency models

Why are there so many consistency models?

Each model has a use in some application.

The *cost of implementation* (as measured by message complexity) decreases as the models become “weaker”.

Implementing linearizability



Needs total order multicast of all reads and writes

Implementing linearizability

- The **total order multicast** forces every process to accept and handle all reads and writes in the **same temporal order**.

- The peers update their copies in response to a write, but only send acknowledgments for reads. After all updates and acknowledgments are received, the local copy is returned to the client.

Implementing sequential consistency

Use total order broadcast all writes only,

but for reads, immediately return local copies.

Eventual consistency

Only guarantees that all replicas eventually receive all updates, regardless of the order.

The system does not provide replication transparency but some large scale systems like Bayou allows this. Conflicting updates are resolved using occasional anti-entropy sessions that incrementally steer the system towards a consistent configuration.

Implementing eventual consistency

Updates are propagated via epidemic protocols. Server S1 *randomly picks* a neighboring server S2, and passes on the update.

Case 1. S2 did not receive the update before. In this case, S2 accepts the update, and both S1 and S2 continue the process.

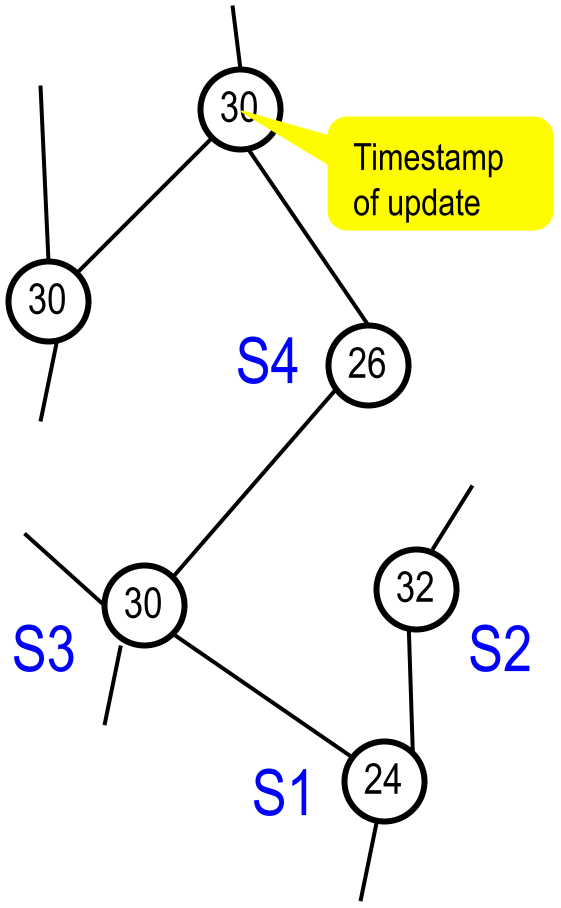
Case 2. S2 already received the update from someone else. In that case, S1 *loses interest in sending updates to S2* (reduces the probability of transmission to S2 to 1/p (p is a tunable parameter)

There is always a finite probability that some servers do not receive all updates. The number can be controlled by changing p.

Anti-entropy sessions

These sessions minimize the “degree of chaos” in the states of the replicas.

During such a session, server S1 will “pull” the update from S2, and server S3 can “push” the update to S4



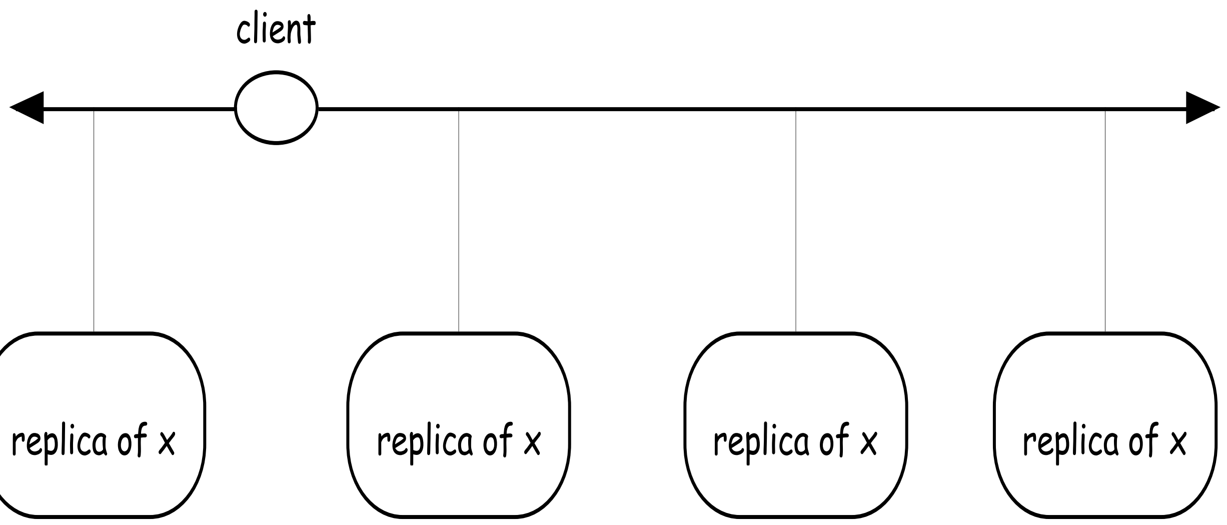
Exercise

Let x, y be two shared variables

```
Process P                      Process Q
{initially x=0}                {initially y=0}
x :=1;                         y:=1;
if y=0 → x:=2 fi;              if x=0 → y:=2 fi;
Print x                        Print y
```

If sequential consistency is preserved, then what are the possible values of the printouts? List all of them.

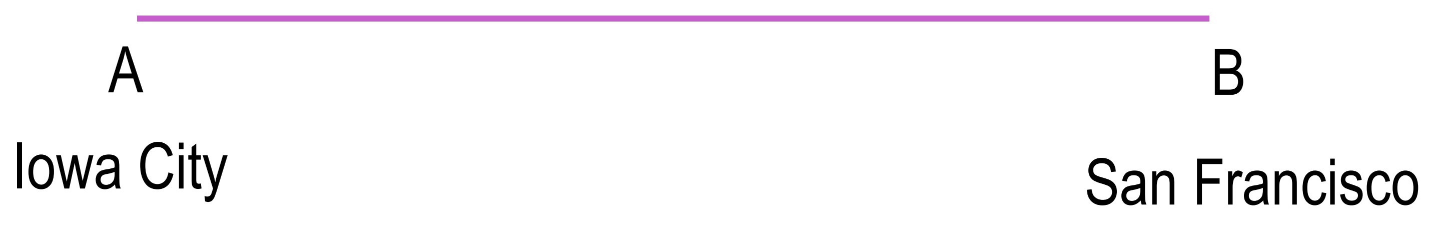
Client centric consistency model



Client-centric consistency model

Read-after-read

If read from A is followed by read from B then the second read should return a data that is as least as old the previous read.

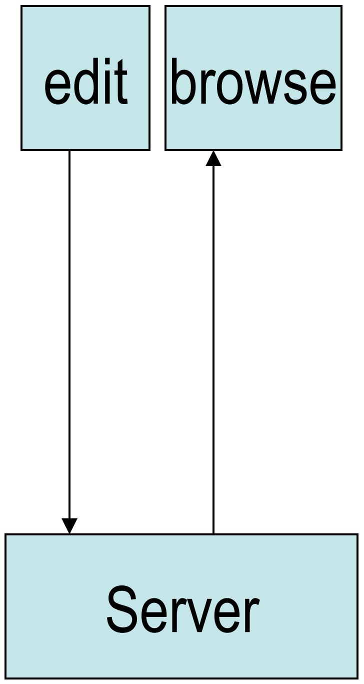


Client-centric consistency model

Read-after-write

Each process must be able to see its own updates.

Consider updating a webpage. If the editor and the browser are not integrated, the editor will send the updated HTML page to the server, but the browser may return an old copy of the page when you view it

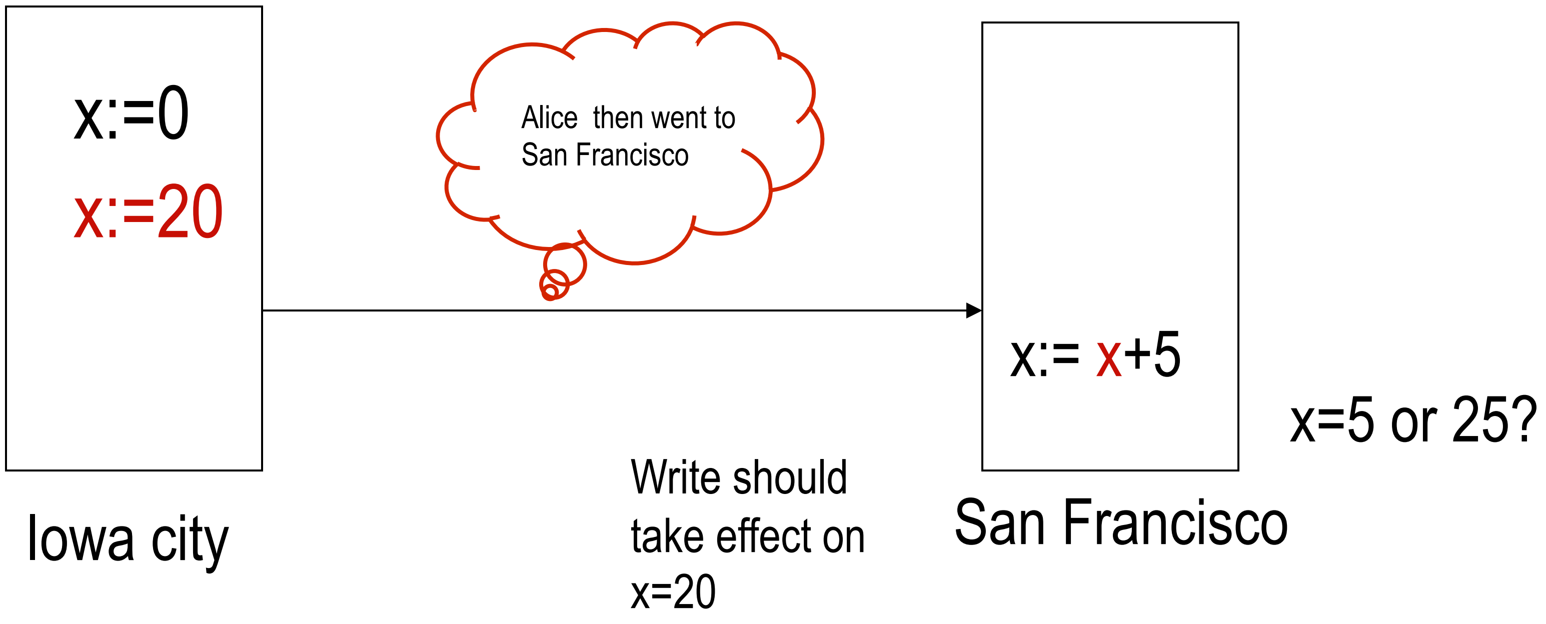


To implement this consistency model, the editor must invalidate the cached copy, forcing the browser to fetch the recently uploaded version from the server.

Client-centric consistency model

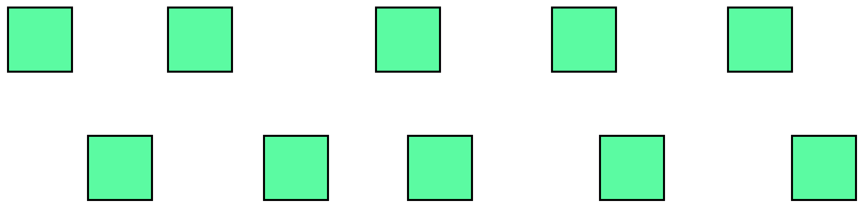
Write-after-read

Each write operation following a read should take effect on the previously read copy, or a more recent version of it.

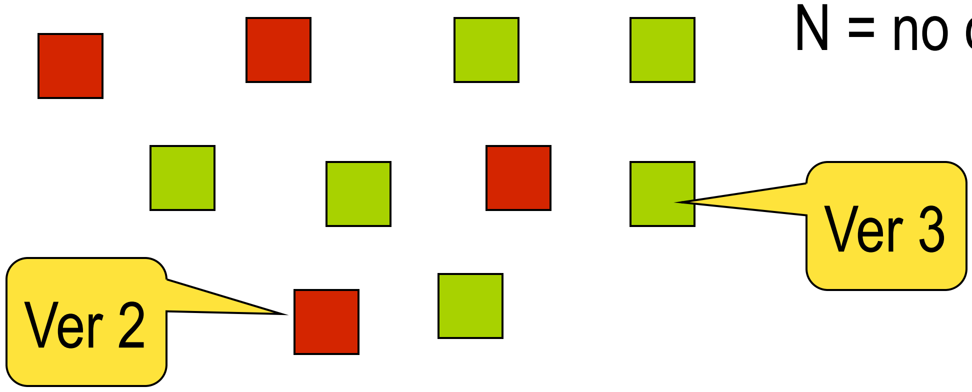

x:=0
x:=20
Alice then went to San Francisco
x:= x+5
x=5 or 25?
Iowa city
Write should take effect on x=20
San Francisco

Quorum-based protocols

A **quorum system** engages only a **designated minimum number of the replicas** for every read or write operation – this number is called the **read or write quorum**. When the quorum is not met, the operation (read or write) is not performed.



Quorum-based protocols



N = no of replicas.

Thomas rule

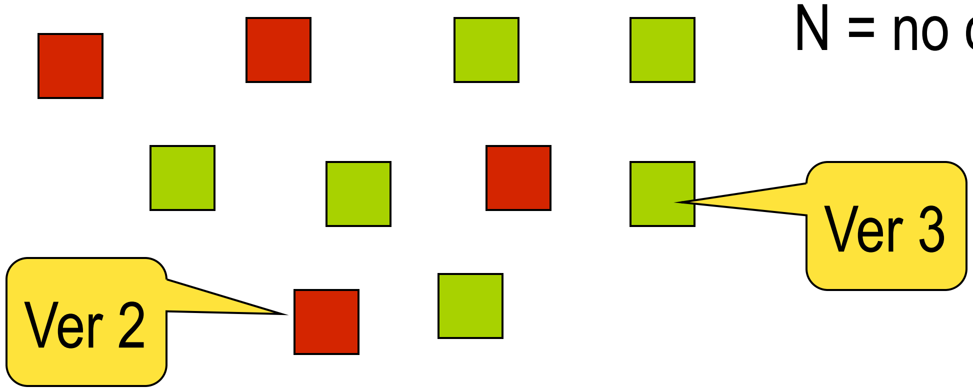
Write quorum

To write, update > N/2 of them, and tag it with new version number.

To read, access > N/2 replicas with version numbers. Otherwise abandon the read

Read quorum

Rationale

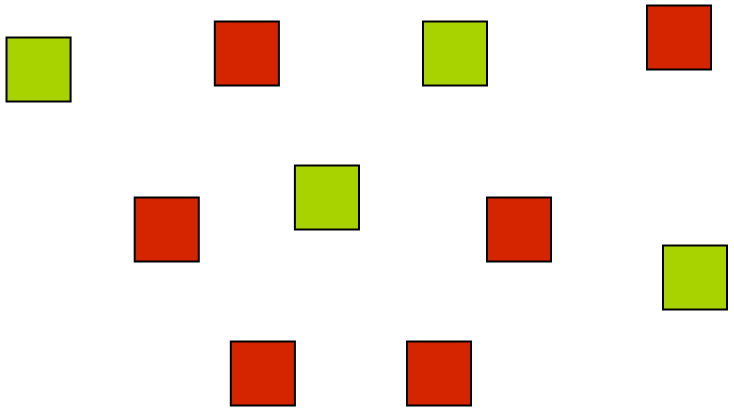


N = no of replicas.

If different replicas store different version numbers for an item, the state associated with a larger version number is more recent than the state associated with a smaller version number.

We require that R+W > N, i.e., read quorums always intersect with write quorums. This will ensure that read results always reflect the result of the most recent write (because the read quorum will include at least one replica that was involved in the most recent write).

How it works

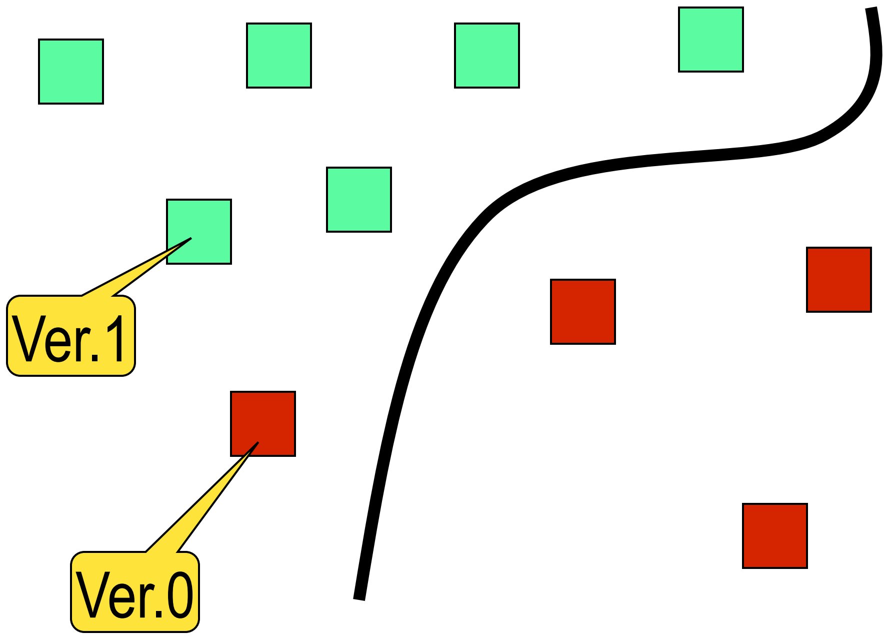


N = no of replicas.

1. Send a *write request* containing the state and new version number to all the replicas and waits to receive acknowledgements from a write quorum. At that point the write operation is complete. The replicas are locked when the write is in progress.

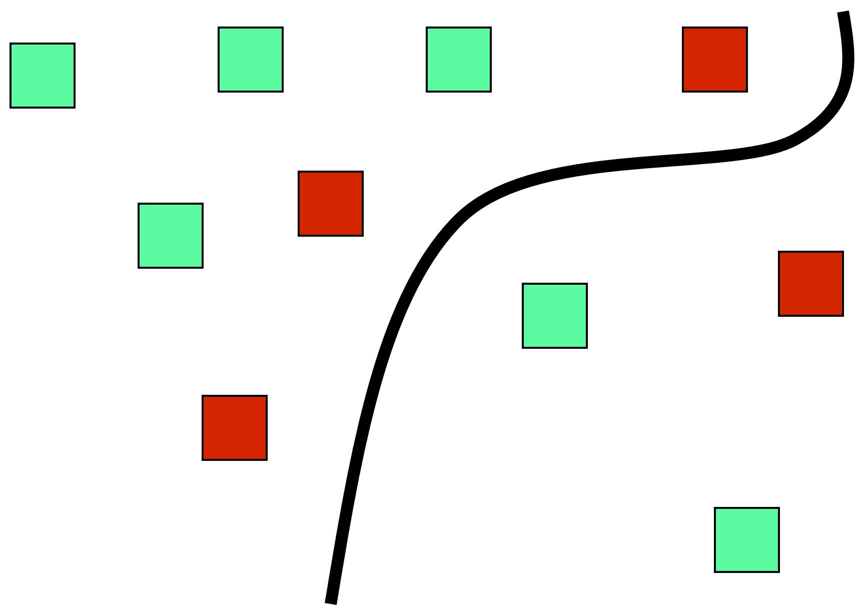
2. Send a *read request* for the version number to all the replicas, and wait for replies from a read quorum.

Quorum-based protocols



After a partition, only the larger segment runs the consensus protocol. The smaller segment contains stale data, until the network is repaired.

Quorum-based protocols



No partition satisfies the read or write quorum

Quorum-based protocols

Asymmetric quorum:

$W + R > N$

$W > N/2$

No two writes overlap

No read overlaps with a write.

R = read quorum

W = write quorum