CS422
Database systems

Distributed Transactions

Data-Intensive Applications and Systems (DIAS) Laboratory
École Polytechnique Fédérale de Lausanne

"Too much agreement kills the chat."
– John Jay Chapman

Some slides adapted from:

- Andy Pavlo
- CS-322

Outline

- **Parallel architectures**
- Distributed transactions
  - Concurrency control
  - Commit
- Replication
- Eventual Consistency

Architectures for (Parallel) Databases

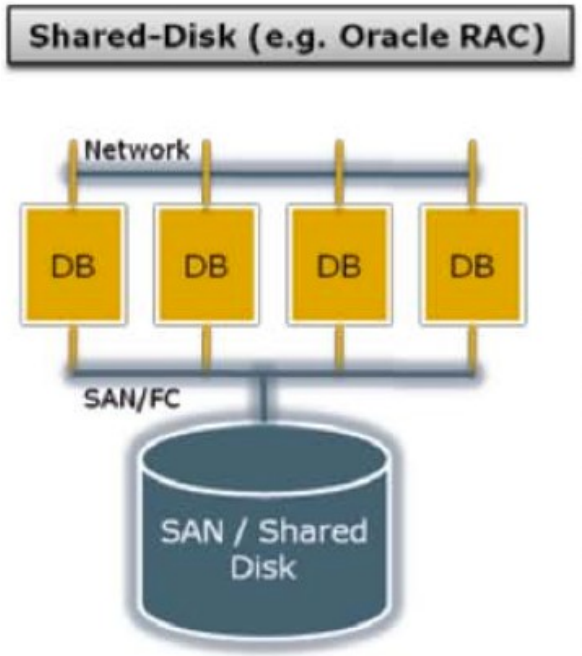
- Shared Everything
- Shared Memory
- Shared Disk
- Shared Nothing

Shared Memory

- Nodes share both RAM and disk
- Dozens to hundreds of processors
- Easy to use and program
- But very expensive to scale
  - (Cheaper) RDMA may change the picture

Shared Disk

- All nodes access the same disks
  - Found in the largest "single-box" (non-cluster) multiprocessors



(SAN: "Storage Area Network")

- Traditionally: Hard to scale past a certain point
  - Contention on storage bandwidth
  - Existing deployments typically have < 10 machines

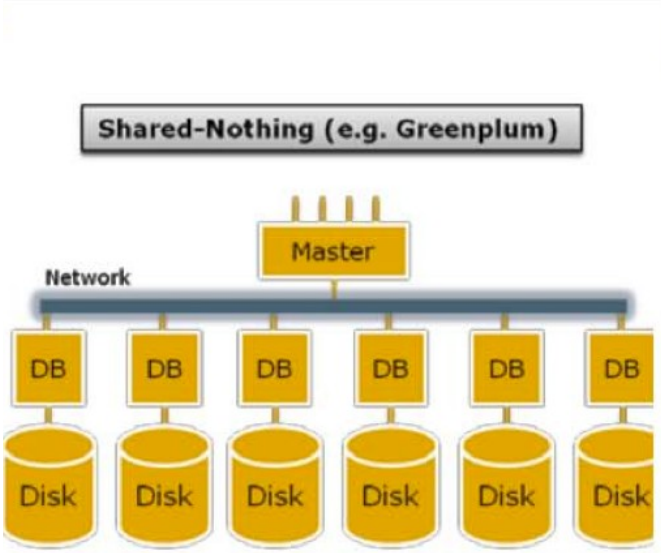
- **New incarnation:** Running over Amazon S3 / your_cloud_service
  - Arbitrary scaleout (Reasons: **S3 has A LOT of disks**, **Executors are stateless**)
  - Single-node performance low

Source:
Greenplum Database Whitepaper

Shared Nothing

- Cluster of machines on high-speed network
  - Called "clusters" or "blade servers”

- Each machine has its own memory and disk
  - Note: All machines today have many cores and many disks. SN systems run many "nodes” in each machine.



Source:
Greenplum Database Whitepaper

- Characteristics
  - Today, this is the most scalable architecture (lowest contention)
  - Most difficult to administer and tune.
    - (Think of rebalancing data when adding a new node!)

Outline

- Parallel architectures
- **Distributed transactions**
  - Concurrency control
  - Commit
- Replication
- Eventual Consistency

Single-node vs. Distributed transactions

- A single-node transaction accesses data that is contained on one data partition.
  - The DBMS does not need to coordinate the behavior of concurrent transactions running on other nodes

- A distributed transaction accesses data at one or more partitions.
  - Requires expensive coordination

Transaction Coordination

- If our DBMS supports multi-operation and distributed transactions, we need a way to coordinate their execution in the system

Two different approaches:

- **Centralized:** Global “traffic cop”
- **Decentralized:** Nodes organize themselves

Distributed Concurrency Control

Need to allow multiple transactions to execute simultaneously across multiple nodes.

- Many of the protocols from single-node DBMS can be adapted

It is harder though! Reasons:

- Network Communication Overhead
- Clock Skew
- Node Failures
- Replication

Distributed 2PL:

- Increased lock duration
- Who detects deadlocks?

Timestamp-based:

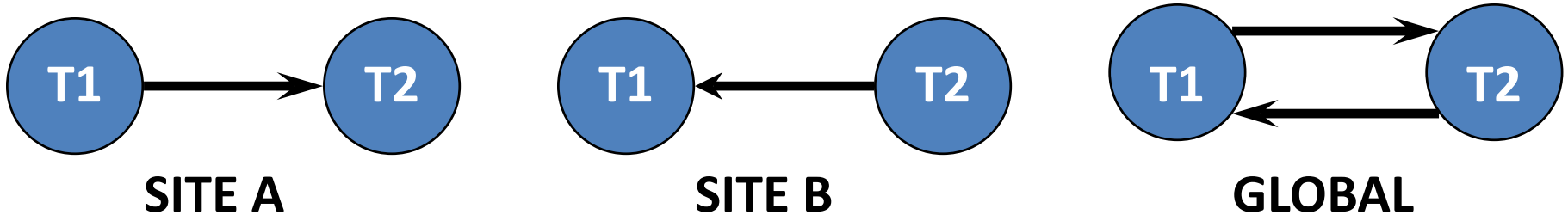
- Whose clock is correct?
- Can we have 1 global clock?

Distributed Locking

- How do we manage locks for objects across many sites?
  - Centralized: One site does all locking.
    - Vulnerable to single site failure.
  - Primary Copy: All locking for an object done at the primary copy site for this object.
    - Reading requires access to locking site as well as site where the object is stored.
  - Fully Distributed: Locking for a copy done at site where the copy is stored.
    - Locks at all sites while writing an object.

Distributed Deadlock Detection

- Each site maintains a local waits-for graph.
- A global deadlock might exist even with acyclic local graphs:



- Three solutions: Centralized (send all local graphs to one site); Hierarchical (organize sites into a hierarchy and send local graphs to parent in the hierarchy); Timeout (abort Xact if it waits too long).
- (Fourth): Abort instead of waiting for lock!

Is a distributed transaction OK to commit?

When a multi-node transaction finishes, the DBMS needs to ask all of the nodes involved whether it is safe to commit.

- Nodes must use an **atomic commit protocol**

Examples:

- Two-phase commit
- Three-phase commit (many assumptions; not used)
- Paxos
- Raft
- ZAB (Apache Zookeeper)

All nodes must agree!

Two-Phase Commit (2PC)

- Site at which Xact originates is coordinator; other sites at which it executes are subordinates.
- When an Xact wants to commit:
  - ✶ Coordinator sends **prepare** msg to each subordinate.
  - ✸ Subordinate force-writes an **abort** or **prepare** log record and then sends a **no** or **yes** msg to coordinator.
  - ✹ If coordinator gets unanimous yes votes, force-writes a **commit** log record and sends **commit** msg to all subs. Else, force-writes **abort** log rec, and sends **abort** msg.
  - ✵ Subordinates force-write **abort/commit** log rec based on msg they get, then send **ack** msg to coordinator.
  - ⌗ Coordinator writes **end** log rec after getting all acks.

Comments on 2PC

- Two rounds of communication: first, voting; then, termination. Both initiated by coordinator.
- Any site can decide to abort an Xact. All sites much agree to commit
- Every msg reflects a decision by the sender; to ensure that this decision survives failures, it is first recorded in the local log.
  - All commit protocol log recs for an Xact contain Xactid and Coordinatorid.
  - The coordinator's abort/commit record also includes ids of all subordinates.

Restart After a Failure at a Site

- If we have a commit or abort log rec for Xact T, but not an end rec, must redo/undo T.
  - If this site is the coordinator for T, keep sending **commit/abort** msgs to subs until **acks** received.
- If we have a prepare log rec for Xact T, but not commit/abort, this site is a subordinate for T.
  - Repeatedly contact the coordinator to find status of T, then write **commit/abort** log rec; redo/undo T; and write **end** log rec.
- If we don't have even a prepare log rec for T, unilaterally abort and undo T.
  - Site may be coordinator. If so, Subordinates may send messages later.

Link and Remote Site Failures

- If a remote site does not respond during the commit protocol for Xact T, either because the site failed or the link failed:
  - If the current site is the coordinator for T, should abort T.
  - If the current site is a subordinate, and has not yet voted yes, it should abort T.
  - If the current site is a subordinate and has voted yes, it is blocked until the coordinator responds.

Observations on 2PC

- Ack msgs used to let coordinator know when it can “forget” a Xact; until it receives all acks, it must keep T in the Xact Table.
- If coordinator fails after sending prepare msgs but before writing commit/abort log recs, when it comes back up it aborts the Xact.
- If a subtransaction does no updates, its commit or abort status is irrelevant.

2PC with Presumed Abort

- When coordinator aborts T, it undoes T and removes it from the Xact Table immediately.
  - Doesn't wait for **acks**; "presumes abort" if Xact not in Xact Table. Names of subs not recorded in **abort** log rec.
- Subordinates do not send acks on abort.
- If subxact does not do updates, it responds to prepare msg with reader instead of yes/no.
- Coordinator subsequently ignores readers.
- If all subxacts are readers, 2nd phase not needed.

2PC: Coordinator failures & **Blocking**

- If coordinator for Xact T fails, subordinates who have voted yes cannot decide whether to commit or abort T until coordinator recovers.
  - T is blocked.
  - Even if all subordinates know each other (extra overhead in prepare msg) they are blocked unless one of them voted no.

The general problem: server failures

- What to do when a server crashes?
  - Servers continuously fail in large-scale systems
  - Wait for recovery: too long…
- **Use replication!**
  - Keep several copies the data, i.e., *replica,* in other servers
  - If a server fails, another server takes over
  - Also use for load balancing
- Up to now: strong consistency
  - All copies have to have the same value

"Customer of online services will be confronted with the consequences of using replication techniques"

Replication & Updating Distributed Data

- Synchronous Replication: All copies of a modified relation (fragment) must be updated before the modifying Xact commits.
  - Data distribution is made transparent to users.

- Asynchronous Replication: Copies of a modified relation are only periodically updated; different copies may get out of sync in the meantime.
  - Users must be aware of data distribution.

Synchronous Replication

- Voting: Xact must write a majority of copies to modify an object; must read enough copies to be sure of seeing at least one most recent copy.
  - E.g., 10 copies; 7 written for update; 4 copies read.
  - Each copy has version number.
  - Not attractive usually because reads are common.
- Read-any Write-all: Writes are slower and reads are faster, relative to Voting.
  - Most common approach to synchronous replication.
- Choice of technique determines *which* locks to set.

Cost of Synchronous Replication

- Before an update Xact can commit, it must obtain locks on all modified copies.
  - Sends lock requests to remote sites, and while waiting for the response, holds on to other locks!
  - If sites or links fail, Xact cannot commit until they are back up.
  - Even if there is no failure, committing must follow an expensive ***commit protocol*** with many msgs.
- So the alternative of *asynchronous replication* is widely used in NoSQL systems (a.k.a eventual consistency)

Asynchronous Replication

- Allows modifying Xact to commit before all copies have been changed (and readers nonetheless look at just one copy).
  - Users must be aware of which copy they are reading, and that copies may be out-of-sync for short periods of time.
- Two approaches: Primary Site and Peer-to-Peer replication.
  - Difference lies in how many copies are ``updatable'' or ``master copies''.

Peer-to-Peer (multi-leader) Replication

- More than one of the copies of an object can be a master in this approach.
- Changes to a master copy must be propagated to other copies somehow.
- If two master copies are changed in a conflicting manner, this must be resolved. (e.g., Site 1: Joe's age changed to 35; Site 2: to 36)
- Best used when conflicts do not arise:
  - E.g., Each master site owns a disjoint fragment.
  - E.g., Updating rights owned by one master at a time.

Primary Site Replication

- Exactly one copy of a relation is designated the primary or master copy. Replicas at other sites cannot be directly updated.
  - The primary copy is published.
  - Other sites subscribe to (fragments of) this relation; these are secondary copies.
- Main issue: How are changes to the primary copy propagated to the secondary copies?
  - Done in two steps. First, capture changes made by committed Xacts; then apply these changes.

Implementing the Capture Step

- (Physical) Log-Based Capture: The log (kept for recovery) is used to generate a Change Data Table (CDT).
  - If this is done when the log tail is written to disk, must somehow remove changes due to subsequently aborted Xacts.

- Procedural Capture: A procedure that is automatically invoked (ex: trigger) does the capture; typically, just takes a snapshot.

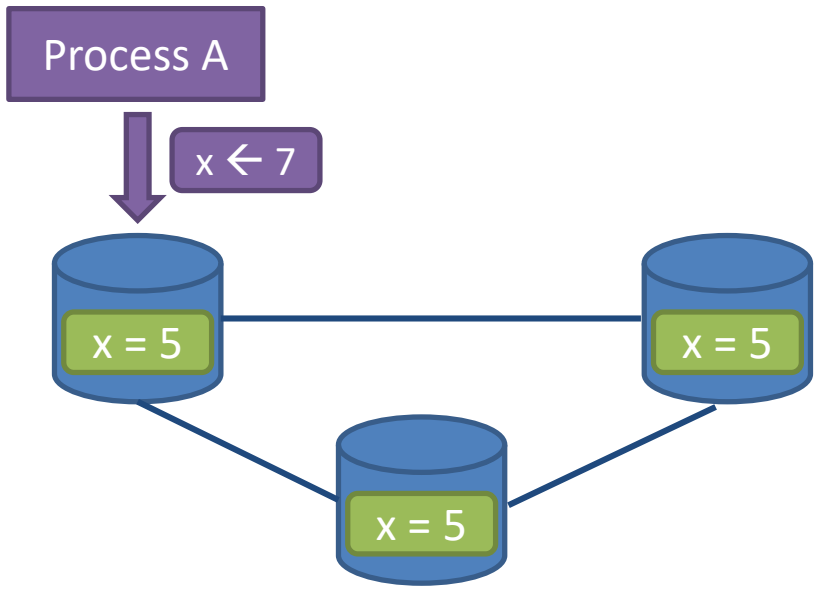
- Log-Based Capture is cheaper & faster, but relies on proprietary log details.
- Middle-ground: Logical log-based capture
  - **[MySQL]** Row-based replication: Describe edits at row granularity

Implementing the Apply Step

- The Apply process at the secondary site periodically obtains (a snapshot or) changes to the CDT table from the primary site, and updates the copy.
  - Period can be timer-based or user/application defined.
- Replica can be a *view* over the modified relation!
  - If so, the replication consists of incrementally updating the materialized view as the relation changes.
- Log-Based Capture plus continuous Apply minimizes delay in propagating changes.
- Procedural Capture plus application-driven Apply is the most flexible way to process changes.

NoSQL wars: How does a system handle failures (and replication)?

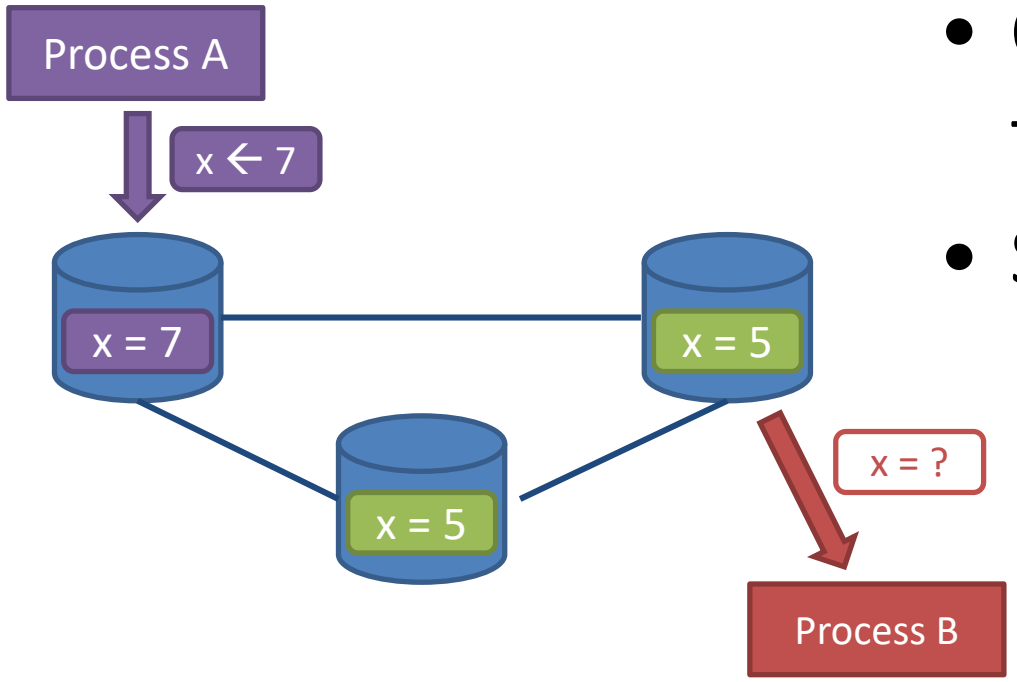
Tracking mutable, replicated state



Replication:

- Synchronous or asynchronous?
- Can processes read from / write to any replica?

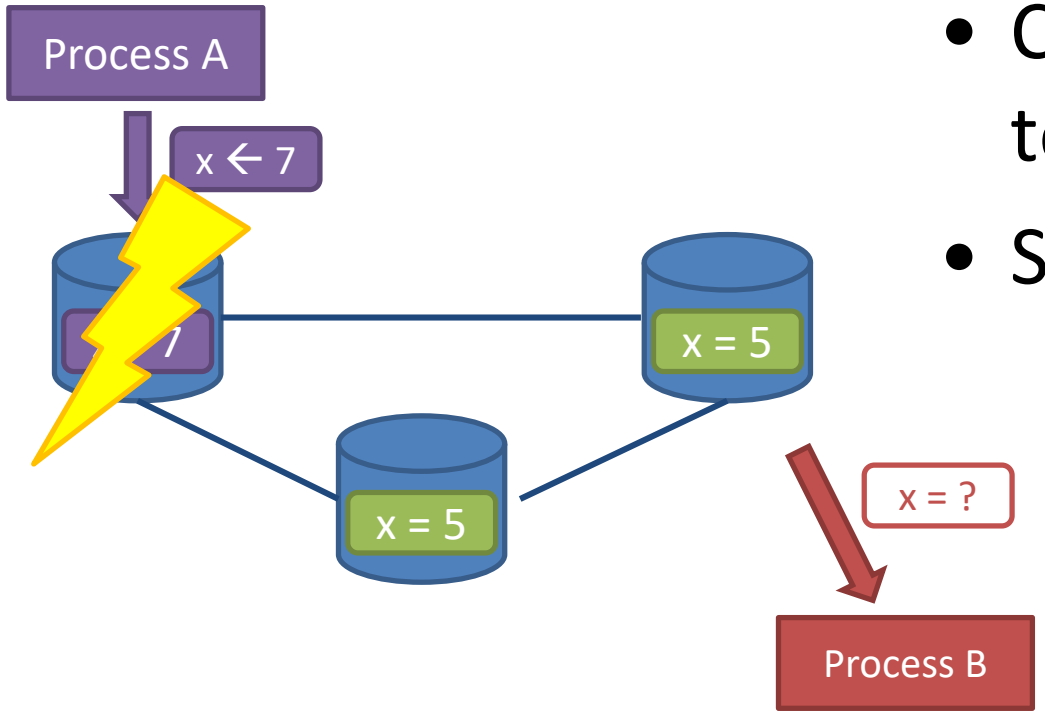
Tracking mutable, replicated state



Replication:

- Can processes read from / write to any replica?
- Synchronous or asynchronous?

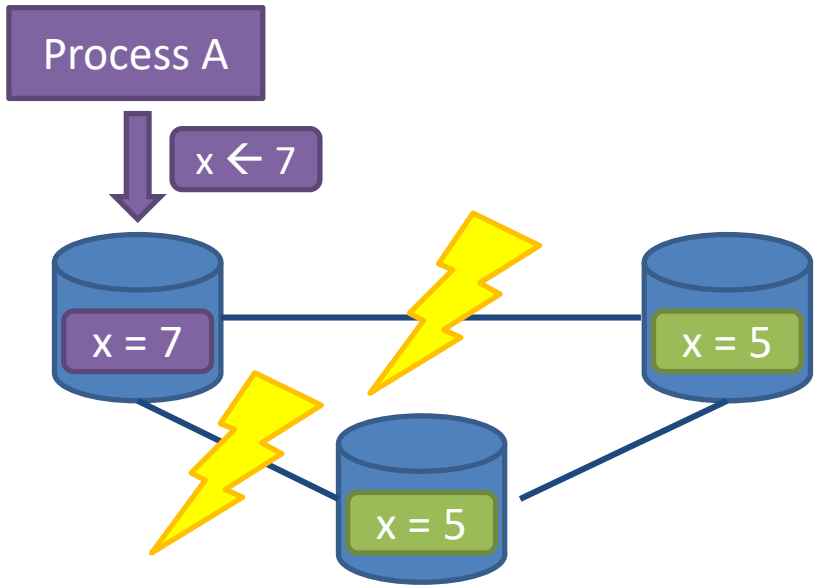
Tracking mutable, replicated state



Replication:

- Can processes read from / write to any replica?
- Synchronous or asynchronous?

Tracking mutable, replicated state

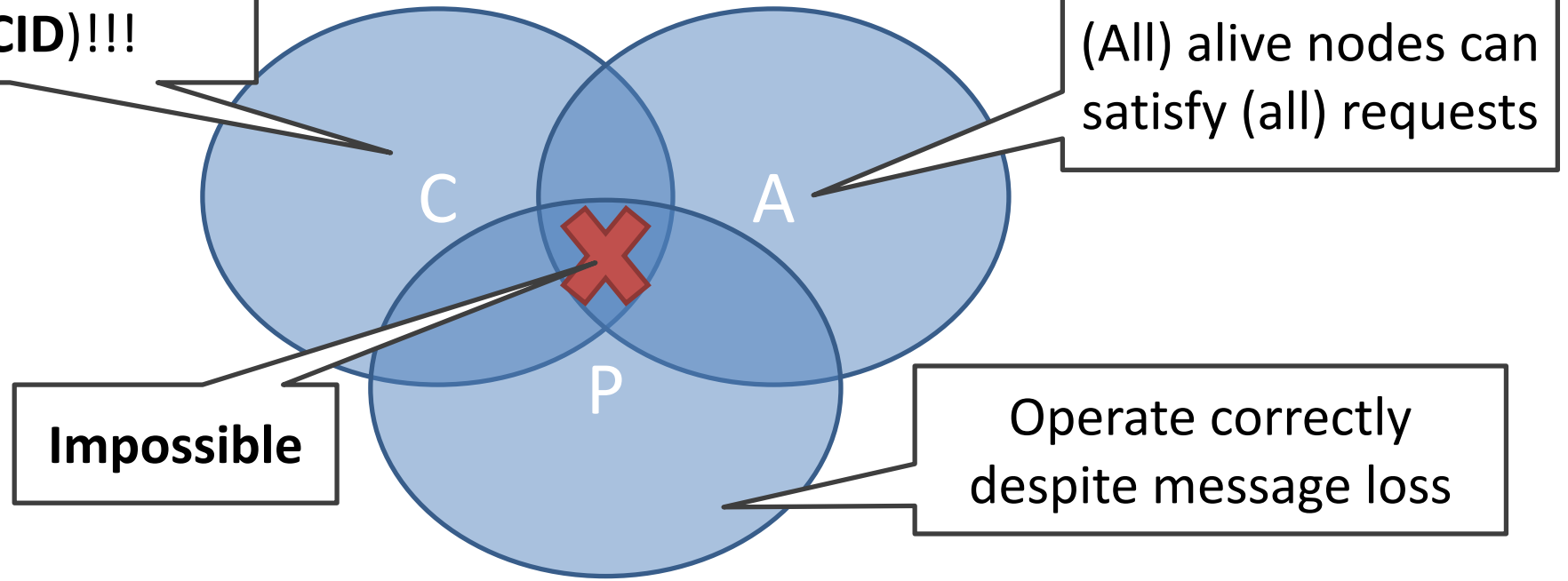


Replication:

- Can processes read from / write to any replica?
- Synchronous or asynchronous?

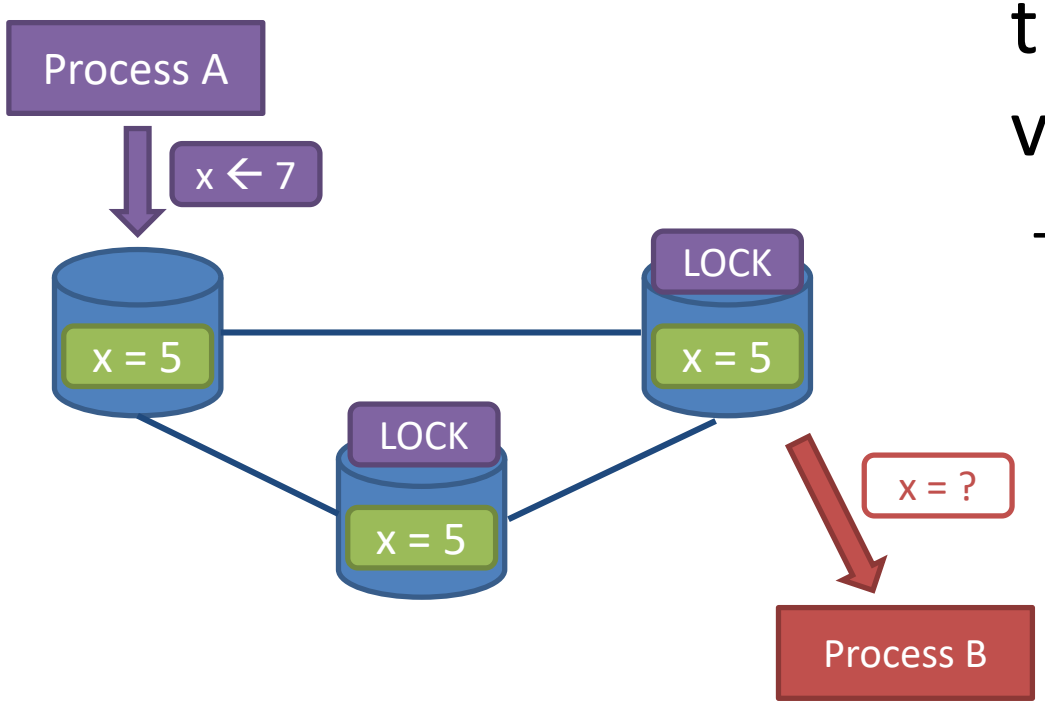
CAP theorem:
A reasoning tool for distributed systems

Consistency
Availability
Partition Tolerance



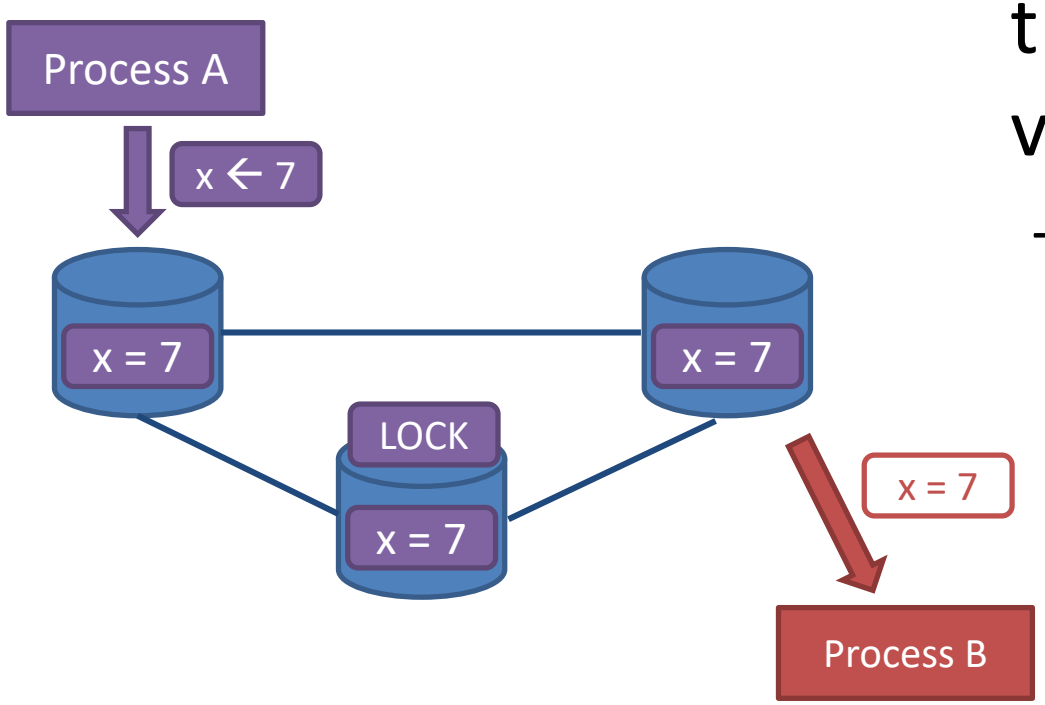
How should a system behave in the presence of a network partition?

CAP - Consistency



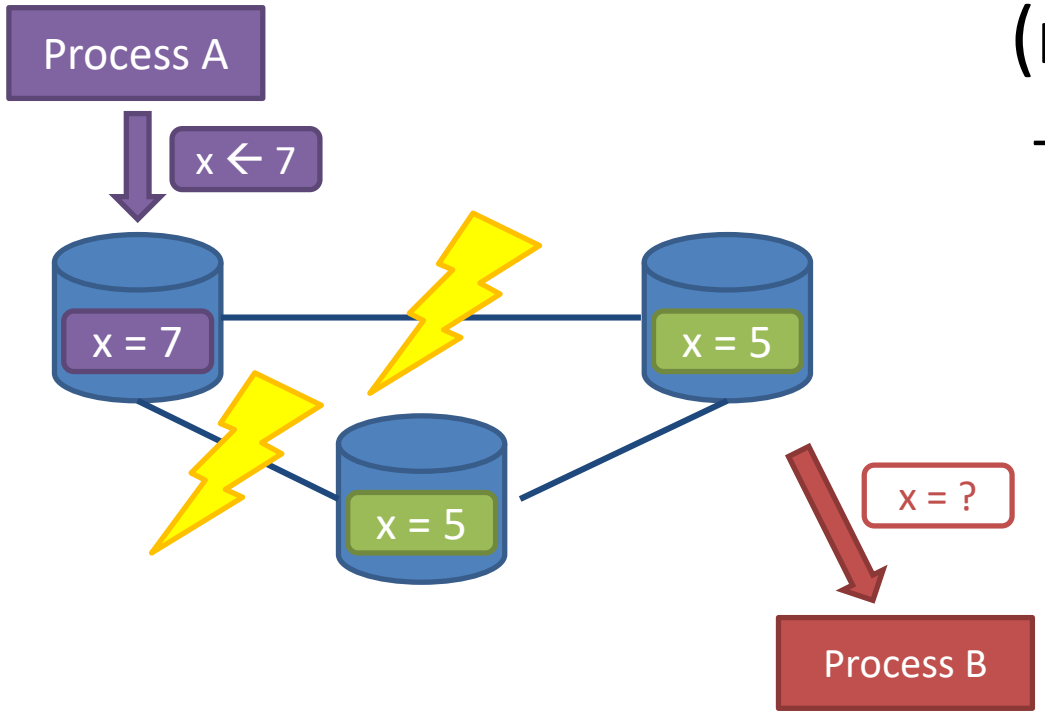
- If master says the txn committed, then it should be immediately visible on replicas.
  - Step in this direction: Synchronous replication

CAP - Consistency



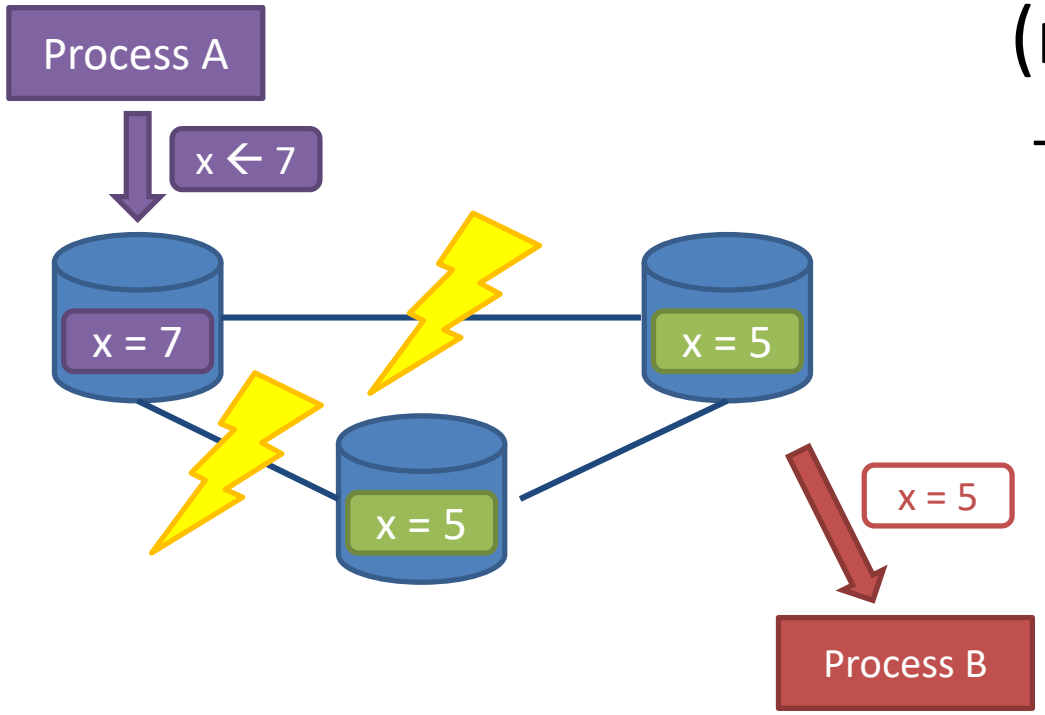
- If master says the txn committed, then it should be immediately visible on replicas.
  - Step in this direction: Synchronous replication

CAP - Availability



- Every request receives a (non-error) response
  - Without guarantee that it contains the most recent write!

CAP - Availability



- Every request receives a (non-error) response
  - Without guarantee that it contains the most recent write!

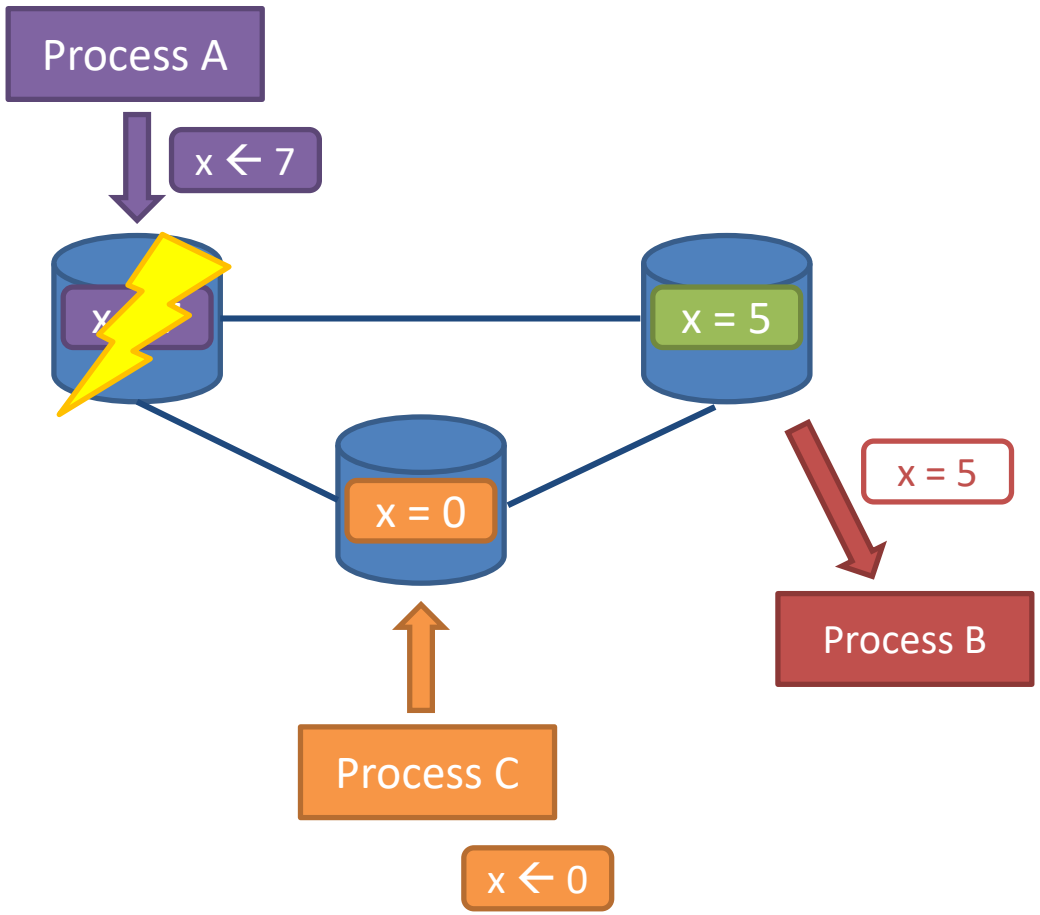
NoSQL: From ACID to **BASE**

- **B**asically **A**vailable **S**oft state **E**ventual consistency
- **E**ventual consistency: If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value
  - At some point, system state will converge (Merge conflicting writes)
  - Typically calibrate consistency degree with use of **read and write quorum**

Quora

- Return the read(write) as successful only if a pre-determined number of servers agree on the same value
- Assume:
  - N = # of servers storing the copies of the data
  - R = # of servers that needs to agree on a value for a read operation
  - W = # of servers that needs to agree on a value for a write operation
- If R(W)-many servers agree on the same value, return the read(write) as successful

W=1, R=1



W+R <= N:

- No overlap of read, write quorum
- No guarantees

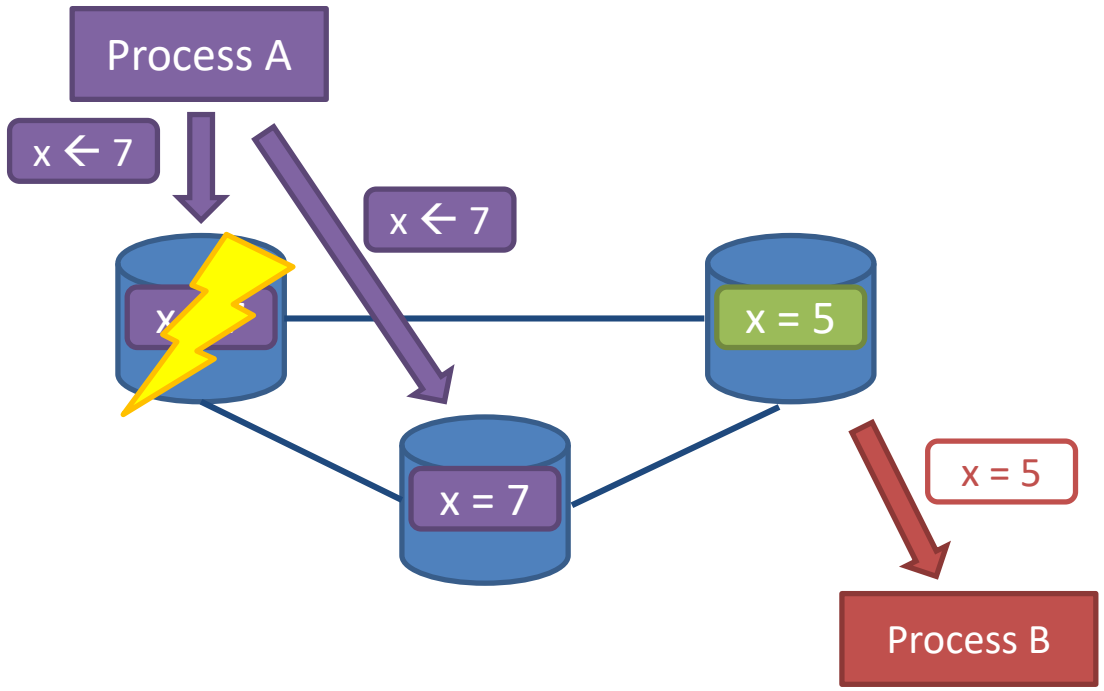
W < (N+1)/2:

- Conflicting writes possible

W = 1:

- No durability guarantees

W=2, R=1



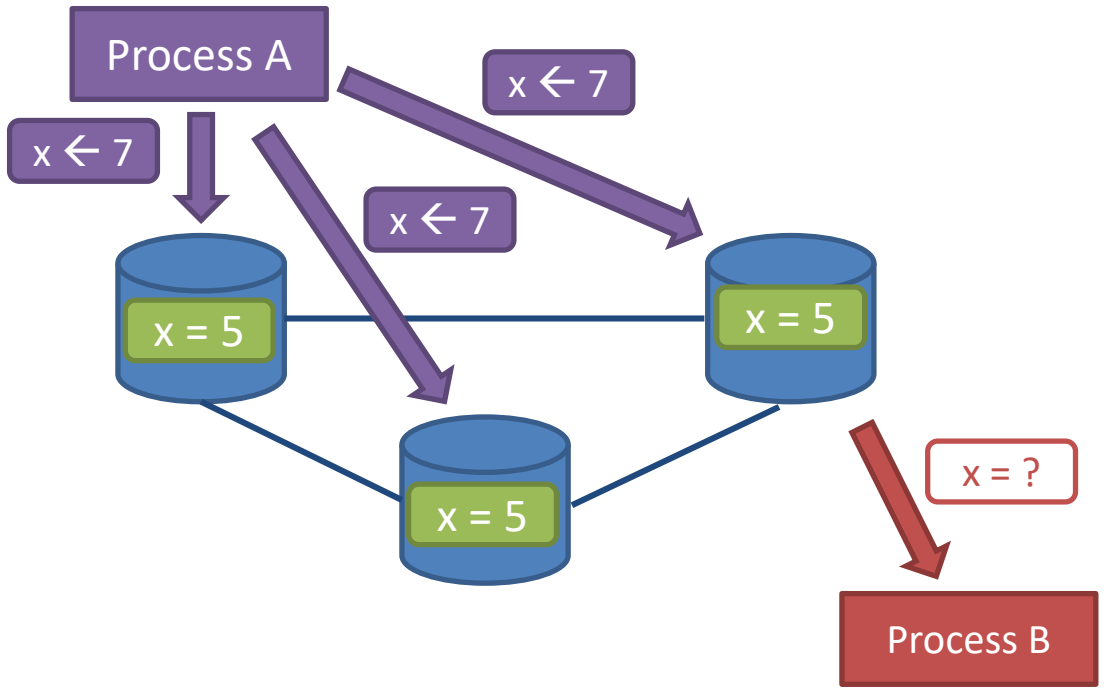
W+R <= N:

- No overlap of read, write quorum
- No guarantees

W = 2:

- Can tolerate one node failure (reg. Write Durability)

W=N, R=1



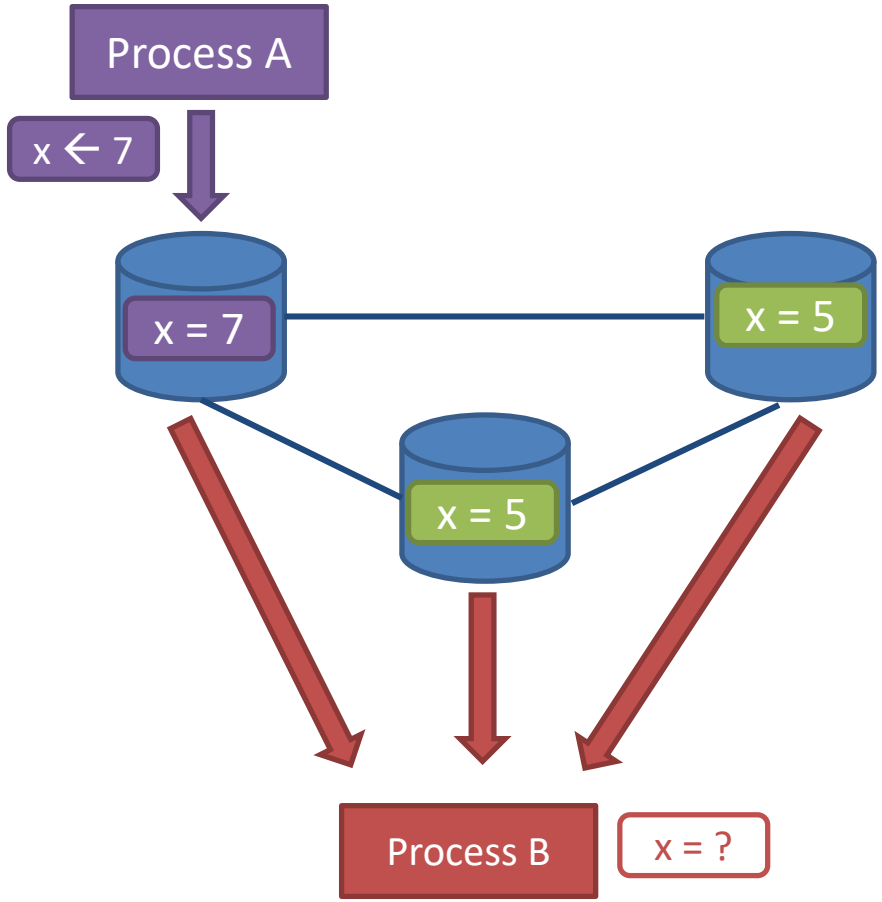
W+R > N:

- Synchronous repl:
  Strong consistency guaranteed
- Asynchronous:
  Cannot make timing assumptions

W = N:

- Can tolerate node failures
  (reg. Availability)

W=1, R=N



W+R > N:

- Synchronous repl:
  Strong consistency guaranteed
- Asynchronous:
  Cannot make timing assumptions

W < (N+1)/2:

- Conflicting writes possible

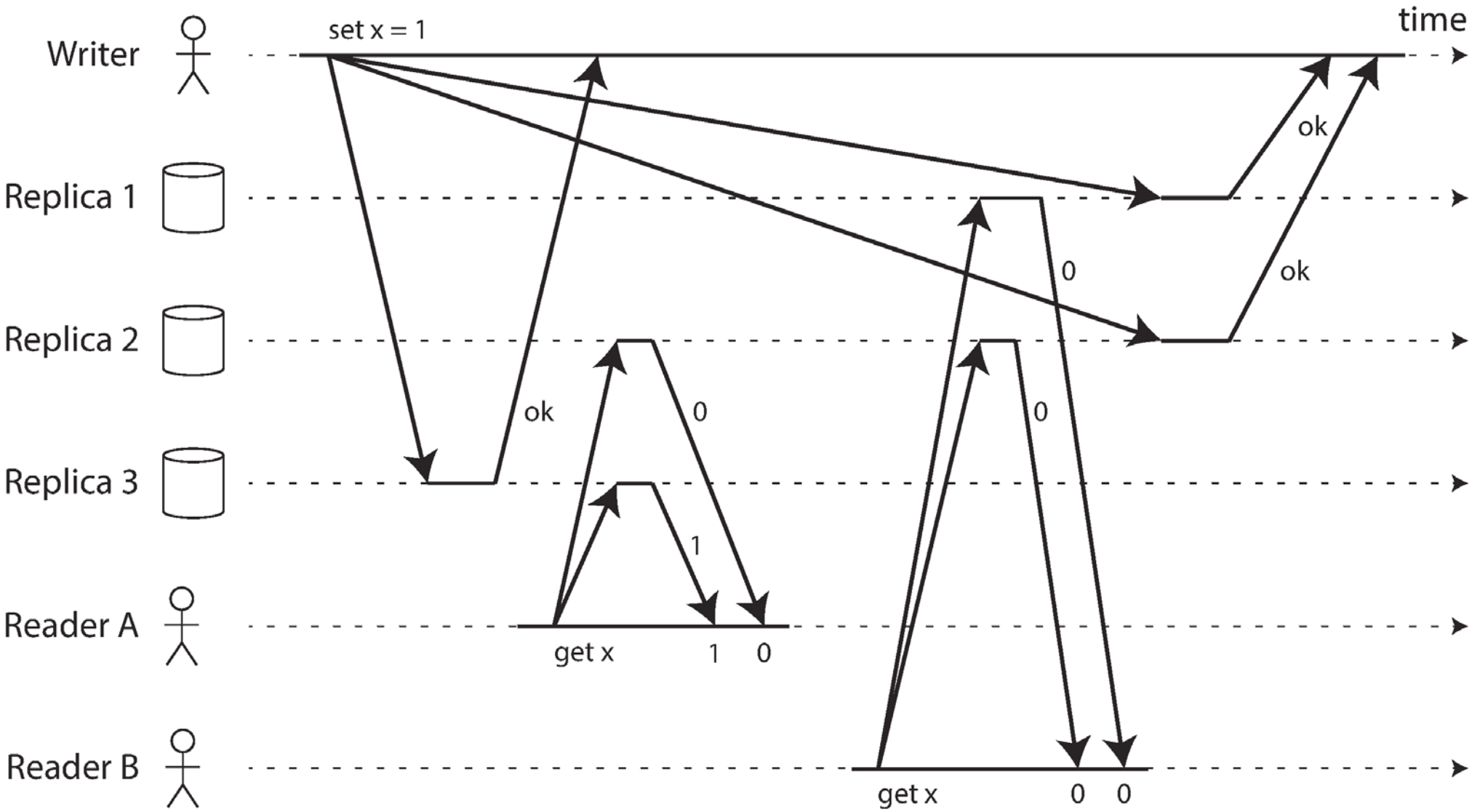
W = 1:

- Very fast write
- No durability guarantees

How to set R and W?

- R = 1 or W = 1 provides the fastest response
- R = N or W = N provides the slowest response
- Sacrifice consistency for faster response-time by tuning R and W parameters
  - If read(write)-optimized, set R(W) = 1
- R + W > N
  - Strong consistency – at least one server overlaps
  - **(Assuming synchronous replication!)**

Strict quorum does NOT imply linearizability



[Source: Martin Kleppmann]

OTHERSIDE

RED HOT CHILI PEPPERS

Replicated Consistency (Explained Through Football)

Concept and Slides
adapted from Doug Terry

Data Replication in the Cloud

Question:

What consistency choices should cloud storage systems offer applications?

Strong Consistency (1)

Windows Azure Storage

Google App Engine

Eventual Consistency (0)

Amazon S3

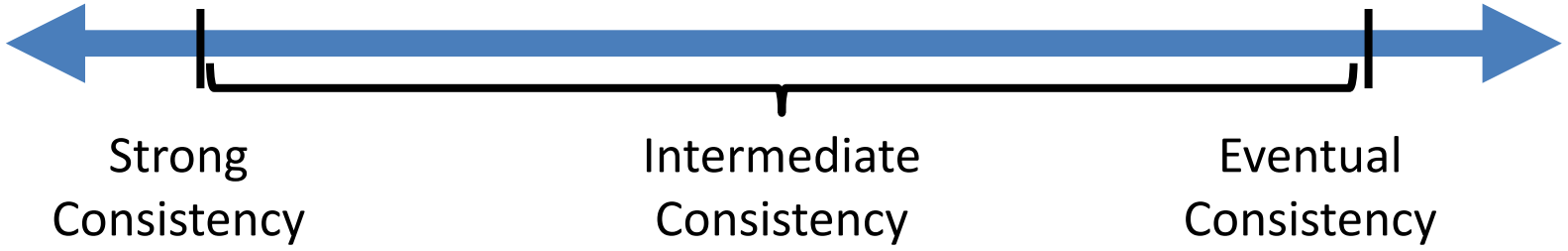
Amazon Simple DB

Yahoo! PNUTS

Cassandra

Most systems can be toggled to “1” or “0”

Consistency is a spectrum



Lots of consistency degrees proposed in research community:
probabilistic quorums, session guarantees, epsilon serializability, fork consistency, causal consistency, demarcation, continuous consistency, ...

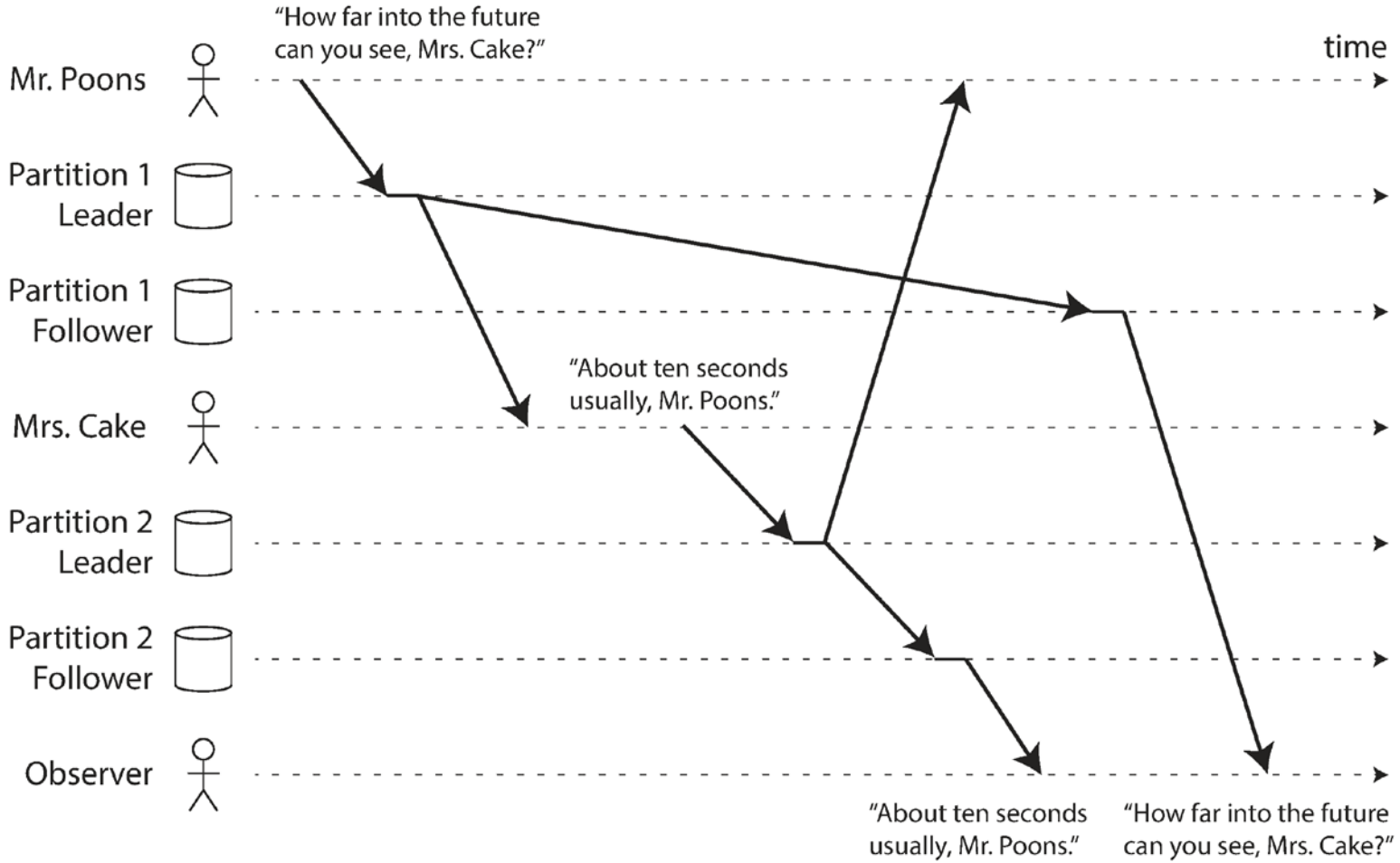
Popular Consistency Guarantees

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.
Bounded Staleness	See all “old” writes.

Can mix + match guarantee levels to create new ones

Consistent Prefix (by counter example)

[Source: Martin Kleppmann]

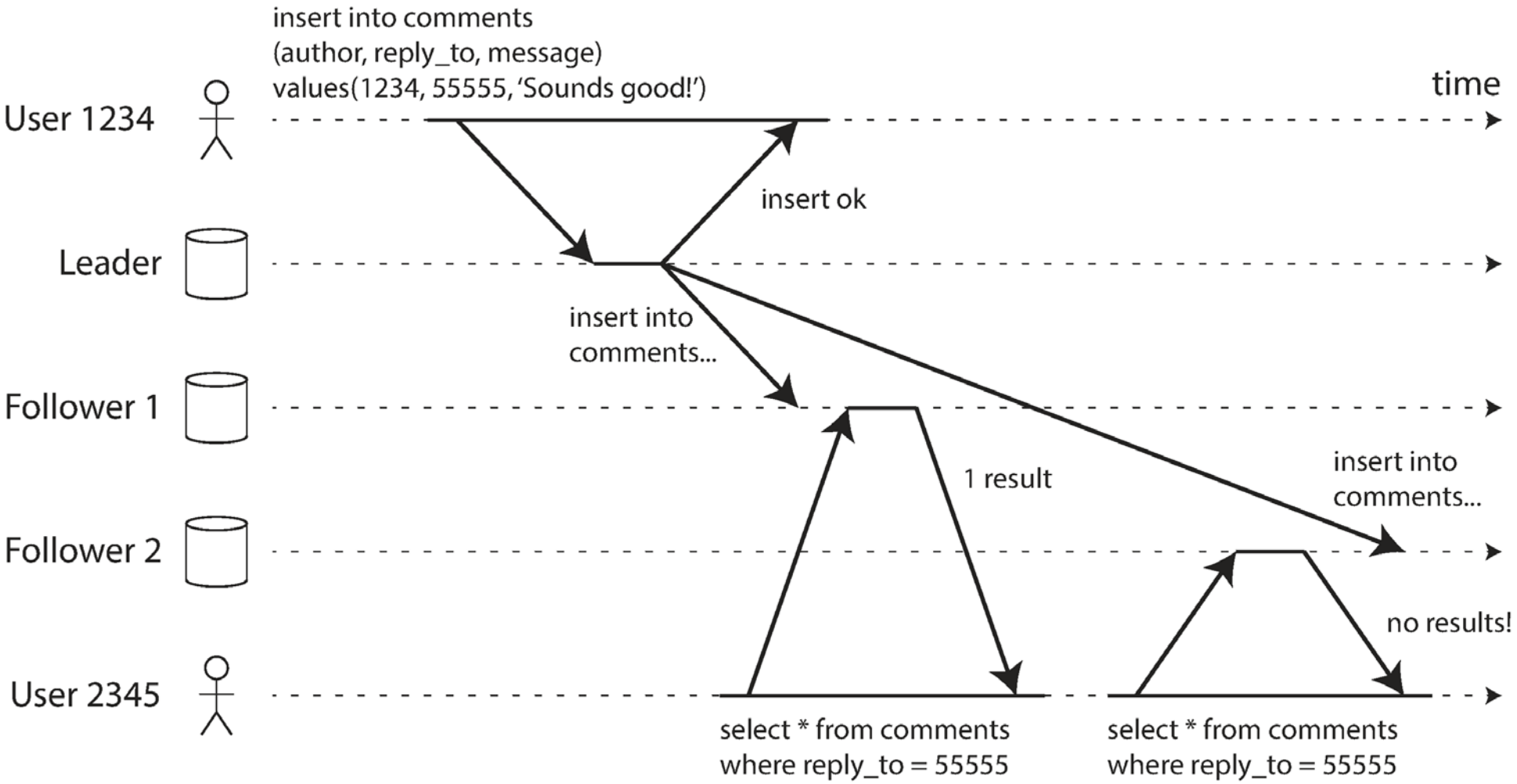


Fix 1: Send causally related writes to same partition (NOT CHEAP)

Fix 2: Track causal dependencies (e.g., Version Vectors)

Monotonic Reads (by counter example)

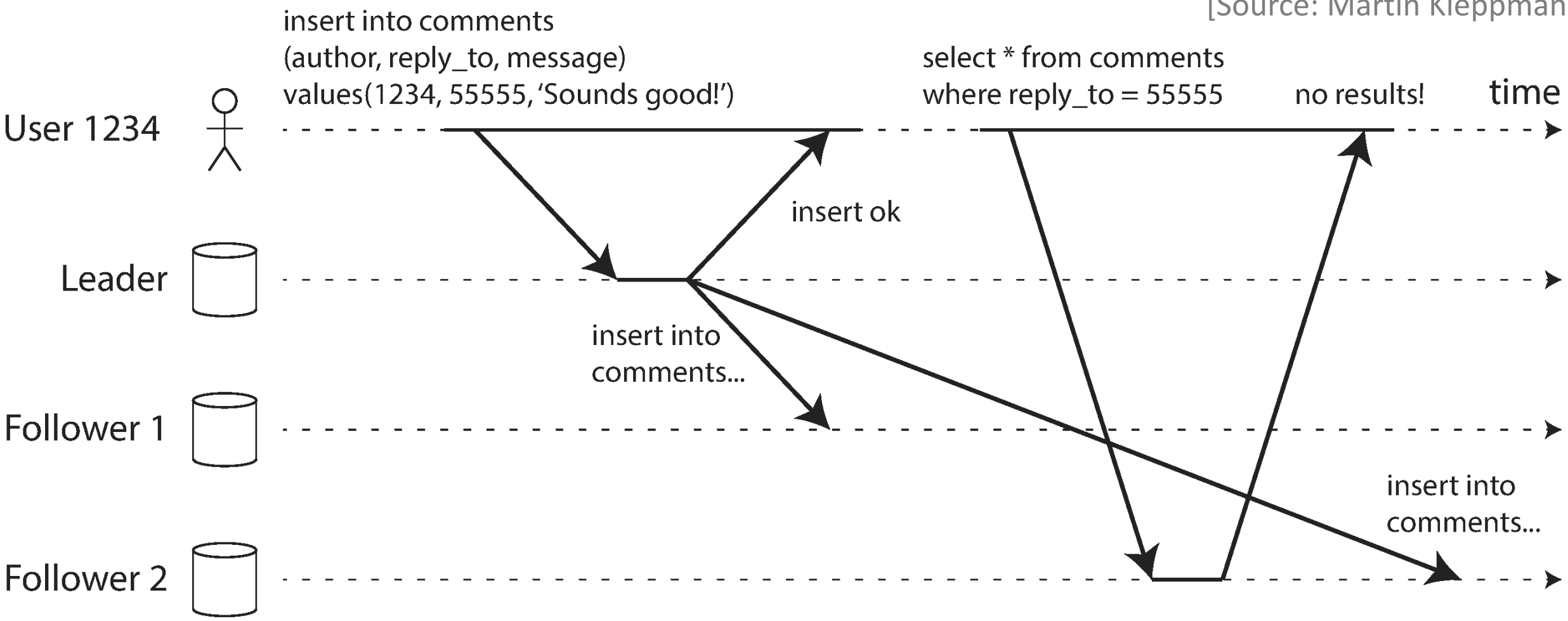
[Source: Martin Kleppmann]



Fix: Always read from same replica

Read My Writes (by counter example)

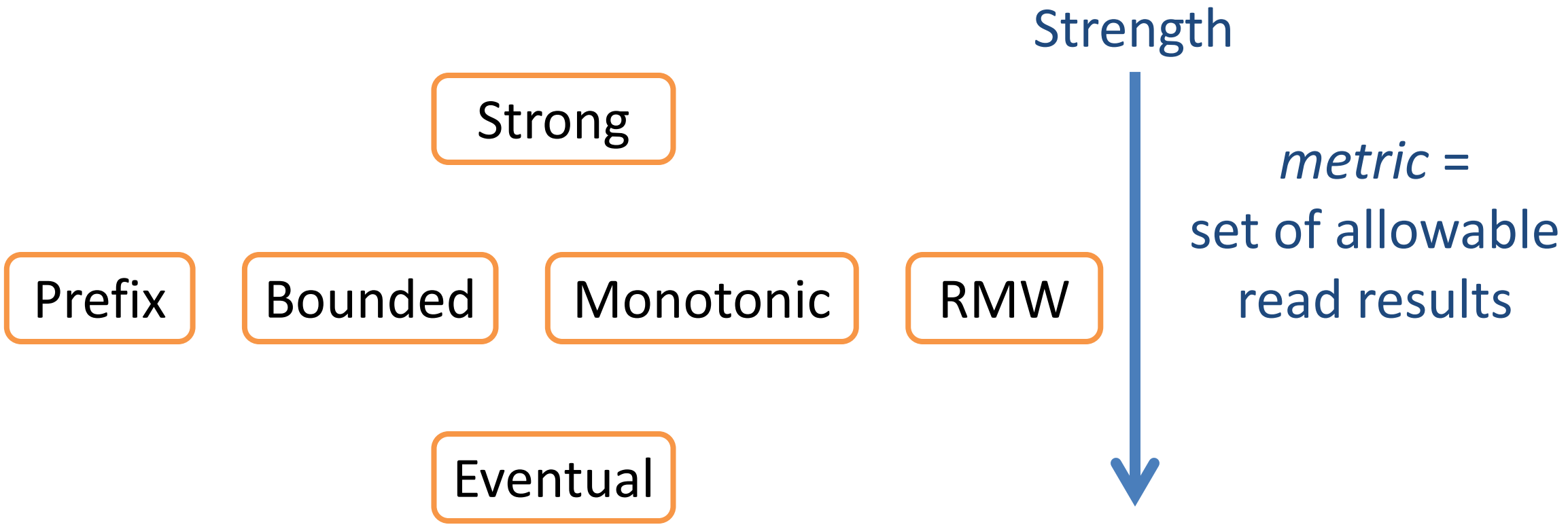
[Source: Martin Kleppmann]



Fix 1: Client always communicates with same replica

Fix 2: Client keeps version info for its writes

Comparing Consistency



Different applications / users require different guarantees

Real-world Example

Home

Visitors

The Game of Football

```
for half = 1 .. 2
    if visitor team scores
       score =  Read ("visitors");
       Write ("visitors", score + 1);
    if home team scores
       score = Read ("home");
       Write ("home", score + 1);
end game;
```

Sample Game:

```
Write("home",1)
Write("visitors",1)
Write("home",2)
Write("home",3)
Write("visitors",2)
Write("home",4)
Write("home",5)
```

5

2

Different spectators have different requirements

Official Scorekeeper (aka *The Only Writer*)

Desired consistency?

Strong

= Read My Writes!

```
score = Read ("visitors");
Write ("visitors", score + 1);
```

Current Score:

5

2

Guarantees
Strong Consistency
Eventual Consistency
Consistent Prefix
Monotonic Reads
Read My Writes
Bounded Staleness

Read My Writes	Writer: 5 - 2
	Rest: 0-0, 1-0, 2-0,3-0,4-0,5-0,0-1,1-1,2-1,3-1,4-1,5-1,0-2,1-2,2-2,3-2,4-2,5-2

Referee

Desired consistency?

Strong Consistency

```
if end of 2nd half then
vScore = Read ("visitors");
hScore = Read ("home");
if vScore == hScore
   start overtime();
else
   end game;
```

Current Score:

5

2

Strong Consistency	5 - 2

Guarantees
Strong Consistency
Eventual Consistency
Consistent Prefix
Monotonic Reads
Read My Writes
Bounded Staleness

Radio Reporter

Desired consistency?

Consistent Prefix

AND Monotonic Reads (or Bounded Staleness)

```
do {
    vScore = Read ("visitors");
    hScore = Read ("home");
    report vScore and hScore
    sleep(30 minutes);
}
```

Guarantees
Strong Consistency
Eventual Consistency
Consistent Prefix
Monotonic Reads
Read My Writes
Bounded Staleness

Current Score:

5

2

Consistent Prefix	**0-1, 1-0, 1-1, 2-1, 3-1, 3-2, 4-2, 5-2**
Monotonic Reads	(after 3-1): 3-1, 4-1, 5-1, **3-2**, **4-2**, 5-2

Sportswriter

Desired consistency?

Eventual?

Strong =

Bounded Staleness

```
while not end of game {
    drink beer;
    eat hotdog;
}
go out for dinner;
vScore = Read (“visitors”);
hScore = Read (“home”);
write article;
```

Guarantees
Strong Consistency
Eventual Consistency
Consistent Prefix
Monotonic Reads
Read My Writes
Bounded Staleness

Statistics Writer

Desired consistency?

Strong (1st read)

Read My Writes (2nd)

```
wait for end of game;
score = Read (“home”);
stat = Read (“season-goals”);
Write (“season-goals”,stat+score);
```

Guarantees
Strong Consistency
Eventual Consistency
Consistent Prefix
Monotonic Reads
Read My Writes
Bounded Staleness

Casual Fan

Desired consistency?

Eventual

```
do {
    stat = Read (“season-goals”);
    argue with friends reg. stat;
    sleep(1 day);
}
```

Guarantees
Strong Consistency
Eventual Consistency
Consistent Prefix
Monotonic Reads
Read My Writes
Bounded Staleness

Consistency Trade-offs

		consistency	performance	availability
Strong Consistency	See all previous writes.	A	D	F
Eventual Consistency	See subset of previous writes.	D	A	A
Consistent Prefix	See initial sequence of writes.	C	B	A
Bounded Staleness	See all “old” writes.	B	C	D
Monotonic Reads	See increasing subset of writes.	C	B	B
Read My Writes	See all writes performed by reader.	C	C	C

Conclusions

- Consistency is a spectrum
- Replication schemes involve tradeoffs between consistency, performance, and availability
- Consistency choices can benefit applications