CS-310 Scalable Software Architectures

Lecture 14: Distributed DB Consistency

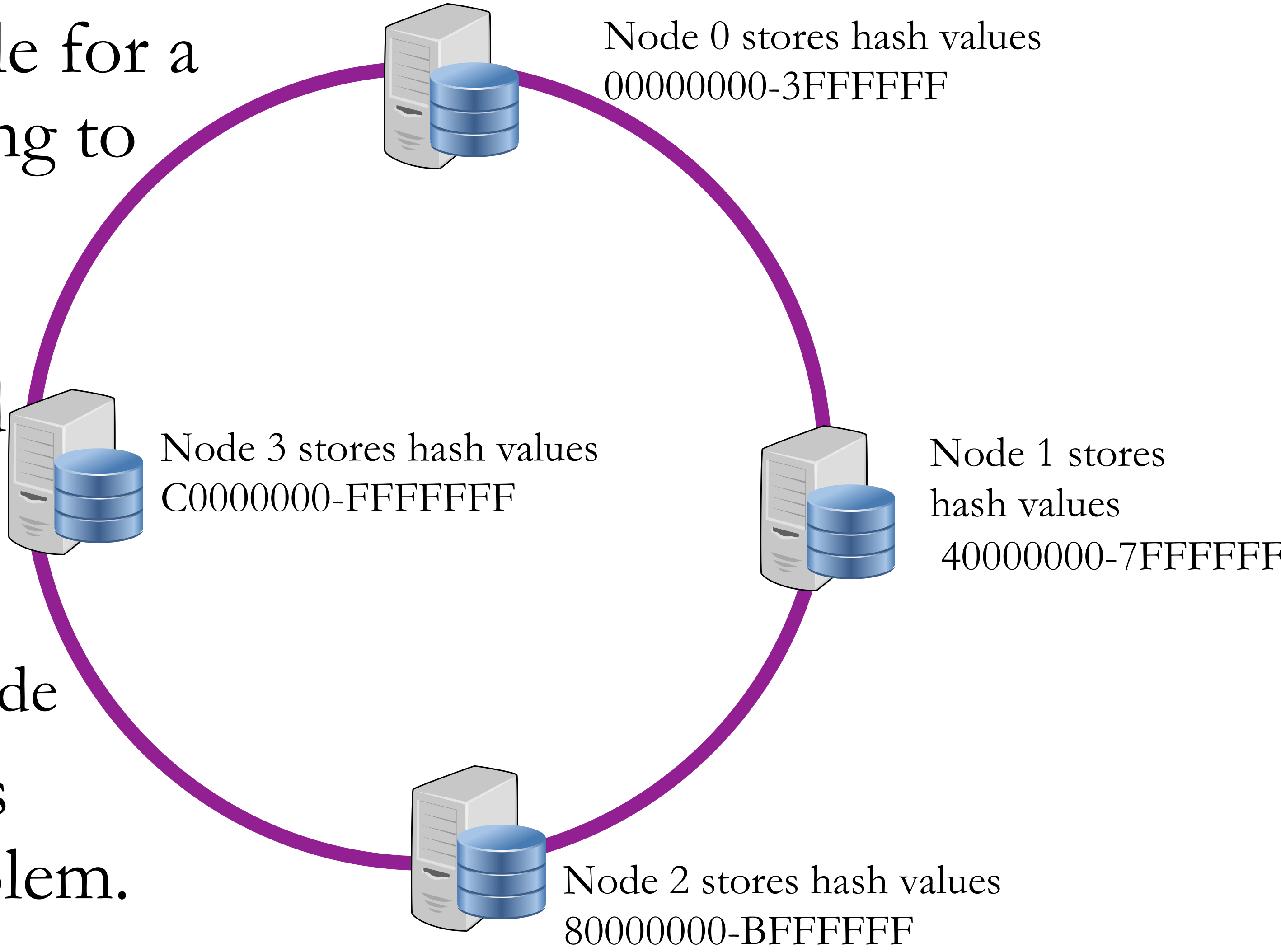
Steve Tarzia

Last Time: NoSQL databases

- **Data partitioning** is necessary to divide write load among nodes.
    - Should minimize references between partitions.
    - Can be treated as a graph partitioning problem.
    - SQL sharding was a special case of data partitioning, done in app code.
- **NoSQL** databases make partitioning easy by eliminating references.
- Without references, data becomes **denormalized**.
    - Duplicated data consumes more space, can become inconsistent.
- **Distributed NoSQL databases** are very scalable, but they provide only a very simple **key-value** abstraction. One key is indexed.
- **Distributed Hash Table** can implement a NoSQL database.
    - The hash space is divided evenly between storage nodes.
    - Client computes hash of key to determine which node should store data.

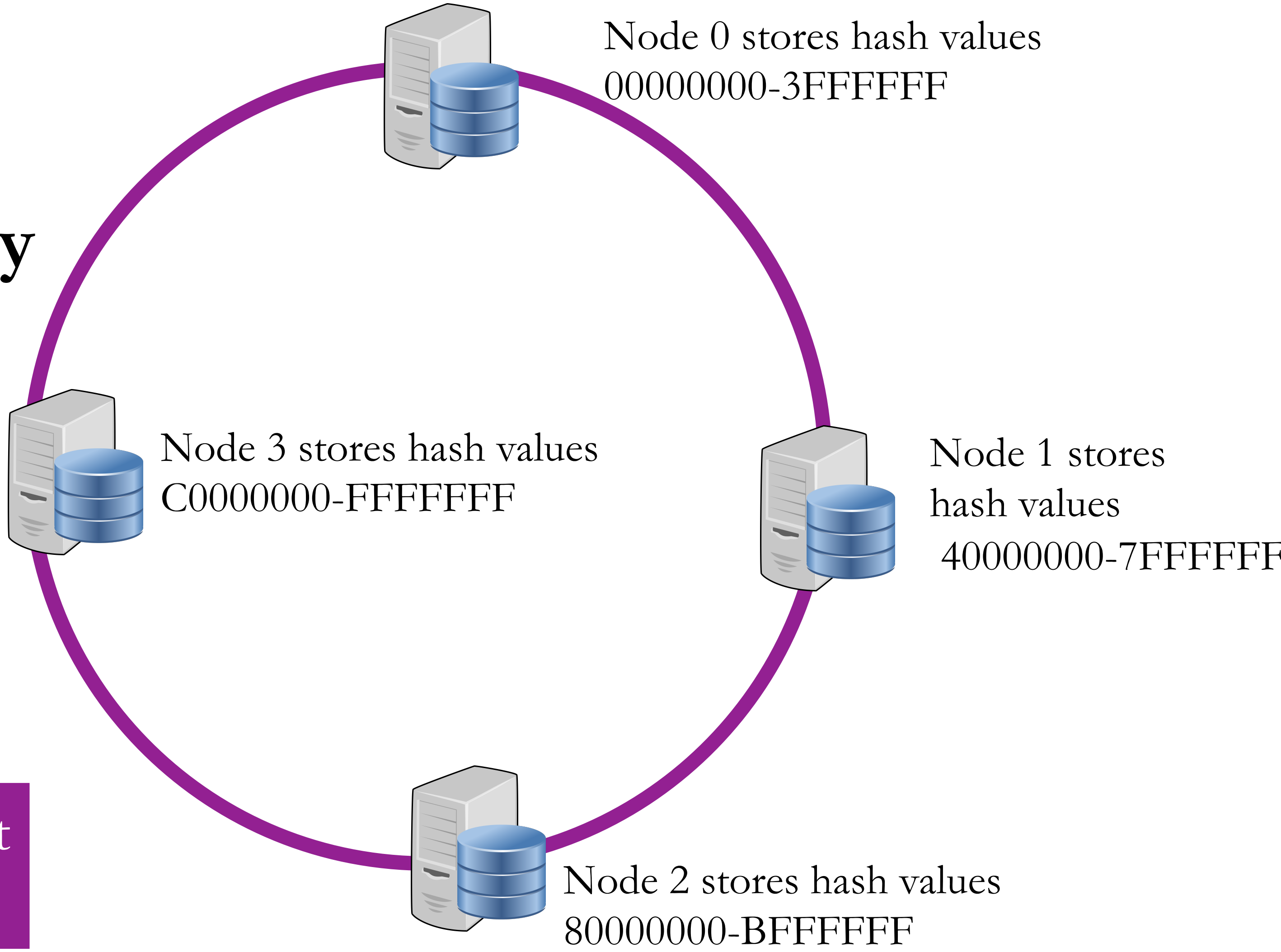
Hash-based partitioning of distributed DB

- aka, a Distributed Hash Table.
- Each cluster node is responsible for a *range of hash values* corresponding to an equal chunk of data
- Hash the **key** to determine where the (key, **value**) is stored
- To find data, client must have:
  - A list of all nodes.
  - <u>hash ranges</u> assigned to each node
- Sharing this node/range info is a **distributed consensus** problem.



A **Shared Nothing** architecture

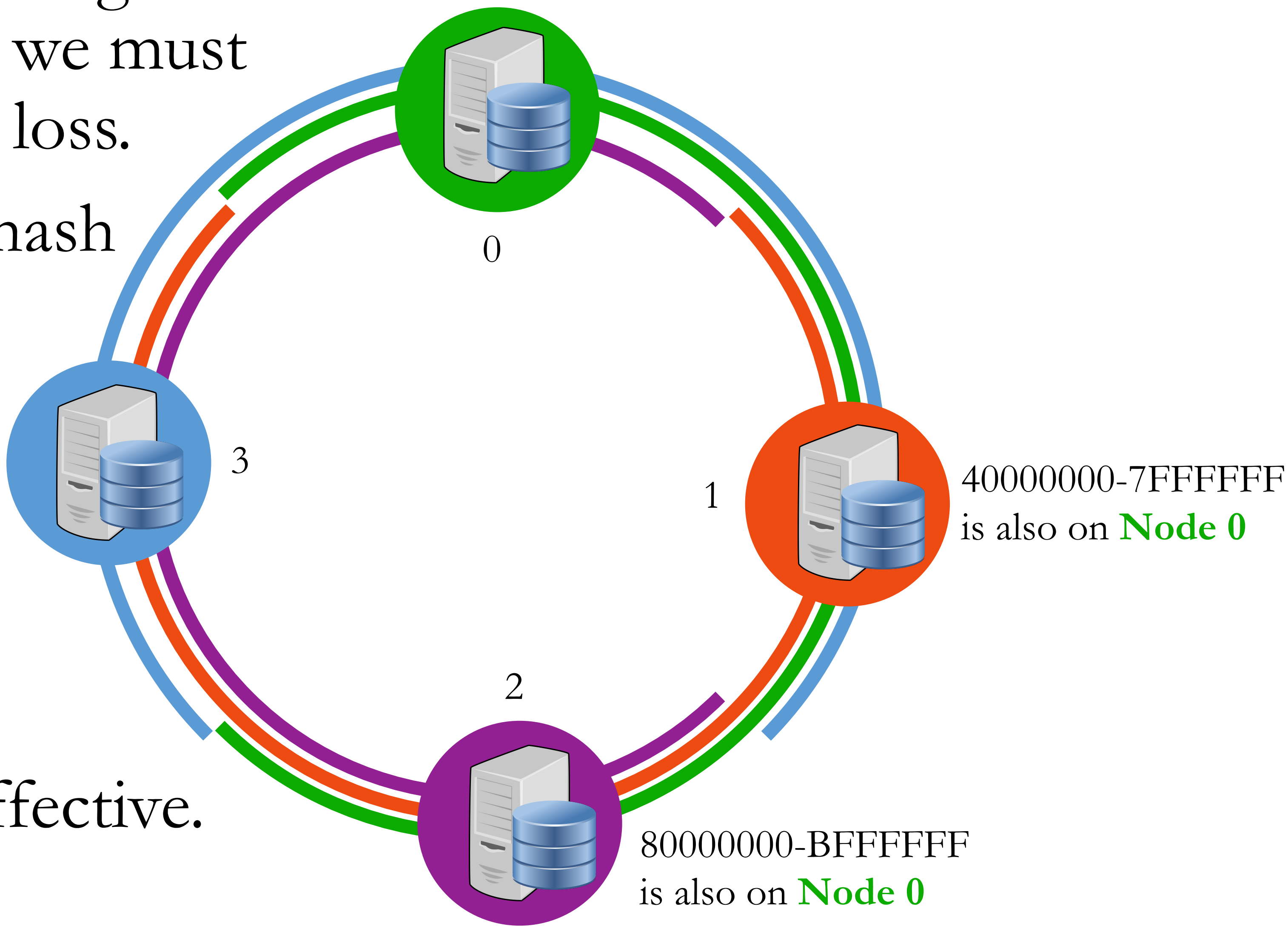
- Each request is handled by **one** node.
  - There are no bottlenecks!
- Both **throughput** and **capacity** are directly proportional to the number of nodes.
- DHTs can scale to thousands of nodes.



But what about **reliability?**

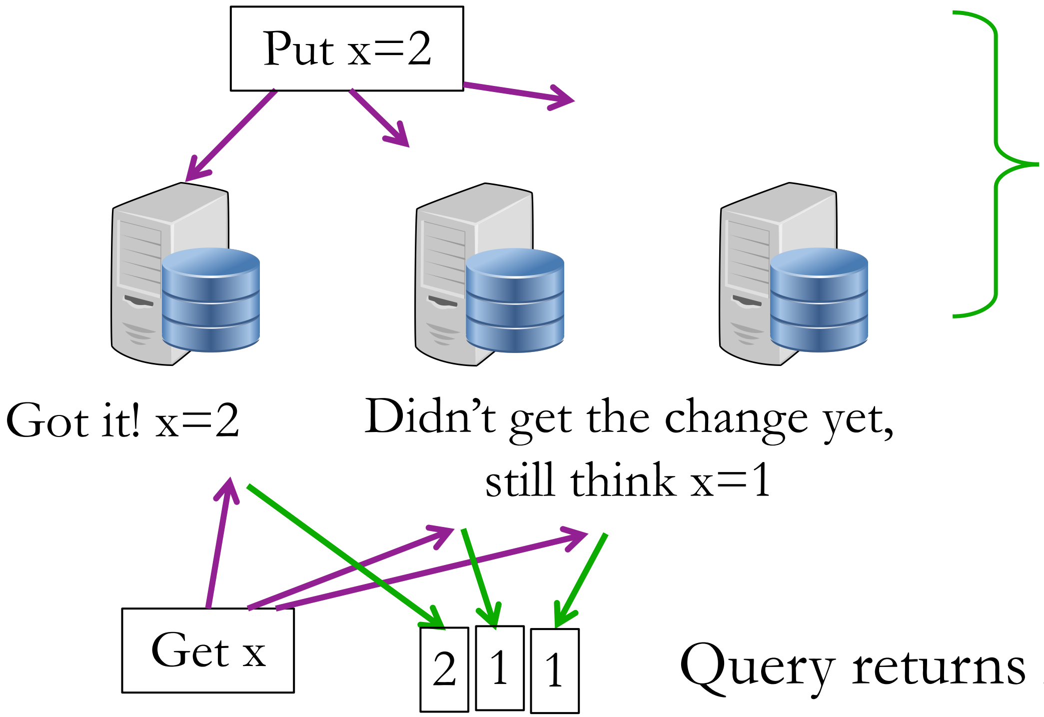
Making the DHT robust

- Having Many nodes means a high chance of a node failure, so we must **replicate** data to avoid data loss.
- Create some overlap in the hash ranges covered by nodes.
  - *Node 0:* 0-7
  - *Node 1:* 3-F
  - *Node 2:* 0-3 and 8-F
  - *Node 3:* 0-7 and C-F
- Other schemes are possible, but this one is simple and effective.



Consistency

- Whenever data is replicated, there is a possibility of **inconsistency**.
  - Eg., an update was sent to three replicas, and one of them gets it first:



Communication and queueing delays are unpredictable. We can send messages, but don't know exactly when they'll arrive.

Query returns inconsistent answers!

- What happens if we try to read while replicas are inconsistent?

CAP Theorem

The most famous result in distributed systems theory.
It says that a distributed system cannot achieve *all three* of the following:

- **C**onsistency: reads always get the most recent write (or an error).
- **Availability:** every request received a non-error response.
- **Partition tolerance:** an arbitrary number of messages between nodes can be dropped (or delayed).

"Pick Two"

In other words:

- When distributed DB nodes are *out-of-sync* (partitioned), we must either accept **inconsistent** responses or **wait** for the nodes to resynchronize.
- To build a distributed DB where every request immediately gets a response that is globally correct, we need a network that is 100% reliable and has no delay.

Client-centric consistency models

- The CAP theorem gives us a tradeoff between **consistency** & **delay**.
- Inconsistency is bothersome. It can cause weird bugs.
- Fortunately, delay is usually something our apps can handle.
- If we really need both consistency and timeliness, then we must go back to a centralized database (probably a SQL relational DB).

- Distributed (NoSQL) DB designs give different options for handling the consistency/delay tradeoff.
- We'll consider a client connecting to the DB cluster.
- What consistency properties might we want to ensure?

Client-centric consistency properties

"More recently written" can include any write by another client.

Monotonic Reads

- If a client reads the value of **x**, later reads of x *by that same client* will always return the same value or a more recently written value.

Read your Writes

- If a client *writes* a value to **x**, later reads of x *by that same client* will always return the same value or a more recently written value.

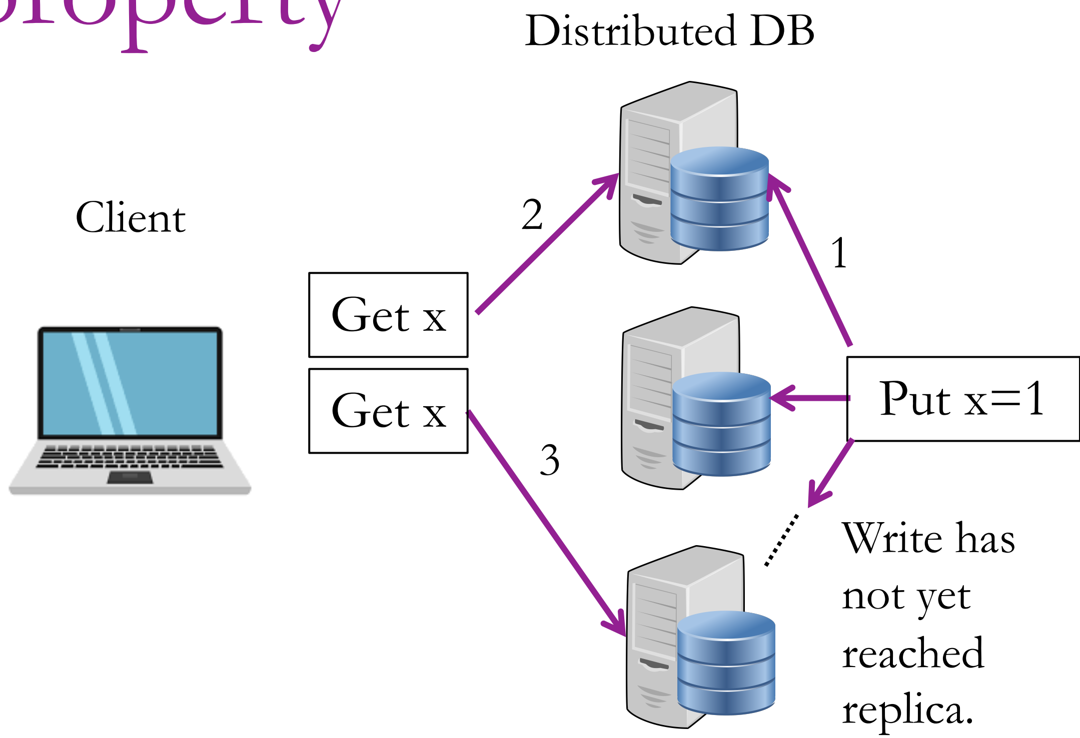
Monotonic Writes

- If a client writes twice to **x**, the first write must happen before the second.

Failing the **Monotonic Read** property

Definition of Monotonic Reads:

- If a client reads the value of **x,** later reads of x *by that same client* will always return the same value or a more recently written value.



How might it fail?

- Read from two different nodes during an incomplete write.

How to prevent this problem?

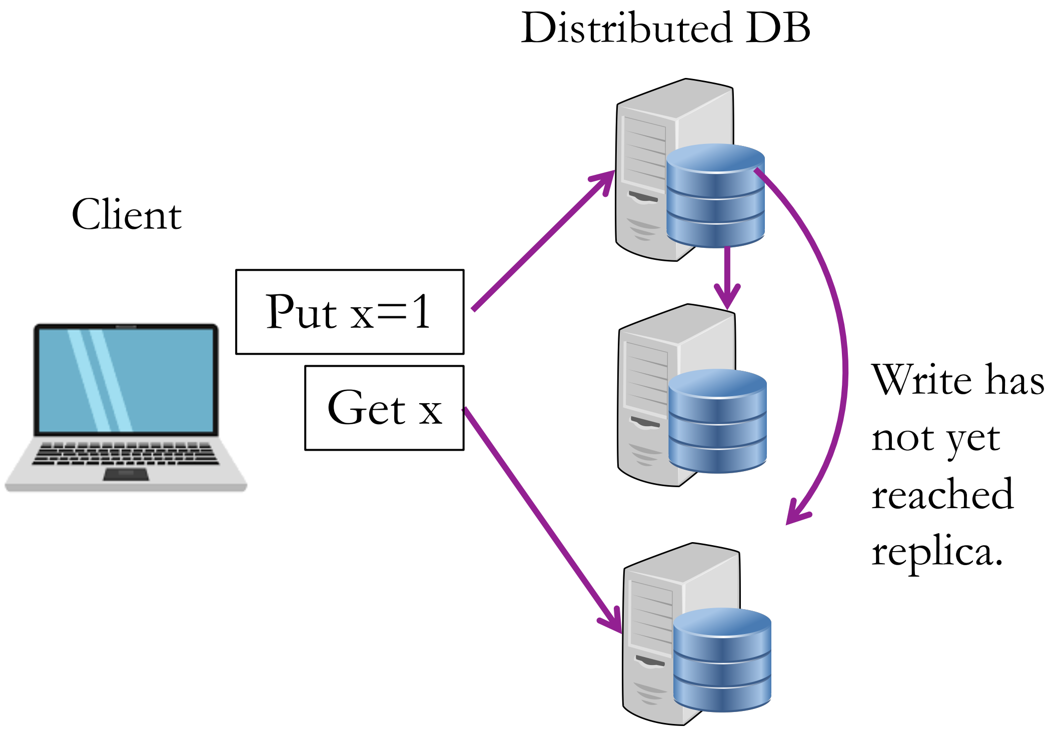
- Make client connect to same node for every request.
- Or delay the second request…

Failing to **Read your Writes**

Definition of Read your Writes:

- If a client *writes* a value to **x**, later reads of x *by that same client* will always return the same value or a more recently written value.

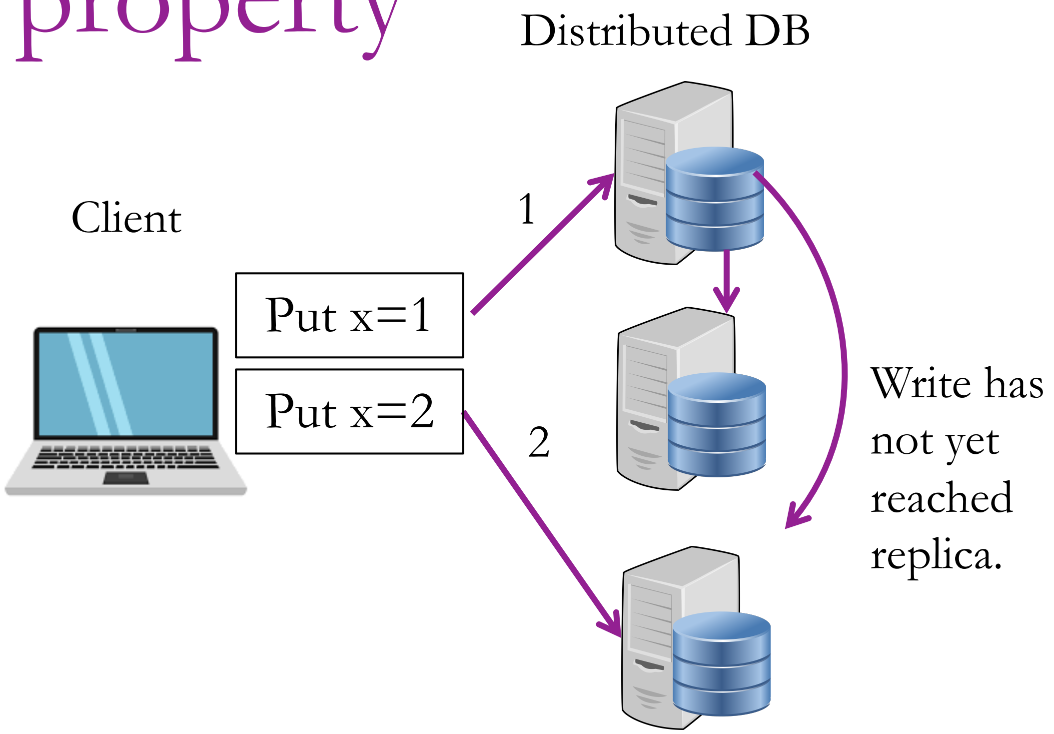
- If the system allows you to write on one node and read from another, you can get the old value if you read too quickly.
- Again, to fix this problem, stick with one node or "slow down."



Failing the **Monotonic Writes** property

Definition of Monotonic Writes:

- If a client writes twice to **x**, the first write must happen before the second.



- The second write can occur on a node before the first arrives.
- Does this matter?
  - Not unless the writes are cumulative. (eg., an increment operation)
  - Note that including a sequence number or timestamp would prevent the delayed write x=1 from being accepted on the third node.
- Solution: same as before.

Two alternatives for achieving Consistency

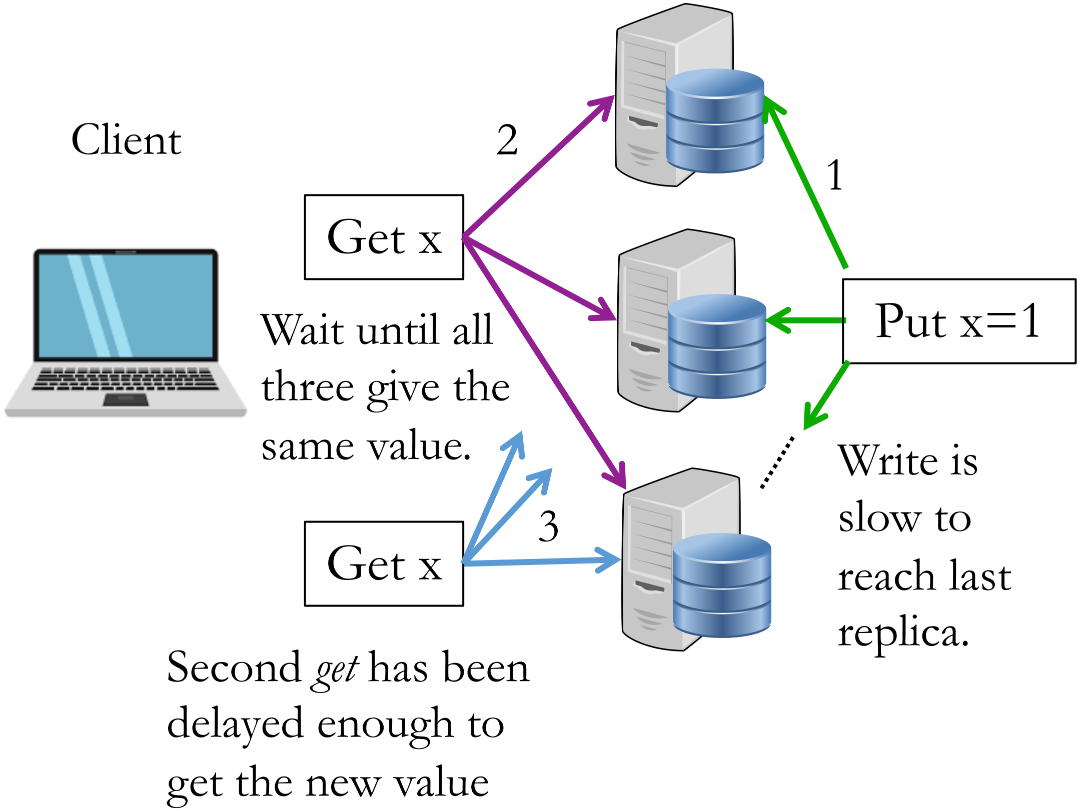
Set some rules for client and replication behavior to achieve consistency.

1. Make client send all requests to **one replica node**.
   - *Pro*: Simplicity.
   - *Con*: Consistency problems arise when a node fails.
     - Client must switch to another node, and the consistency problems are again possible.
     - *Note*: if don't care about fault tolerance, then avoid replication to get consistency. MongoDB does not replicate data and thus has Consistency and Partition Tolerance but lacks Availability because a failed node causes downtime (**C**A**P**).
2. Make client **wait** until the the read or write is synchronized across the whole system.
   - For efficiency, we only care about the single key/value being synchronized.
   - How do we know when the value is synchronized? 

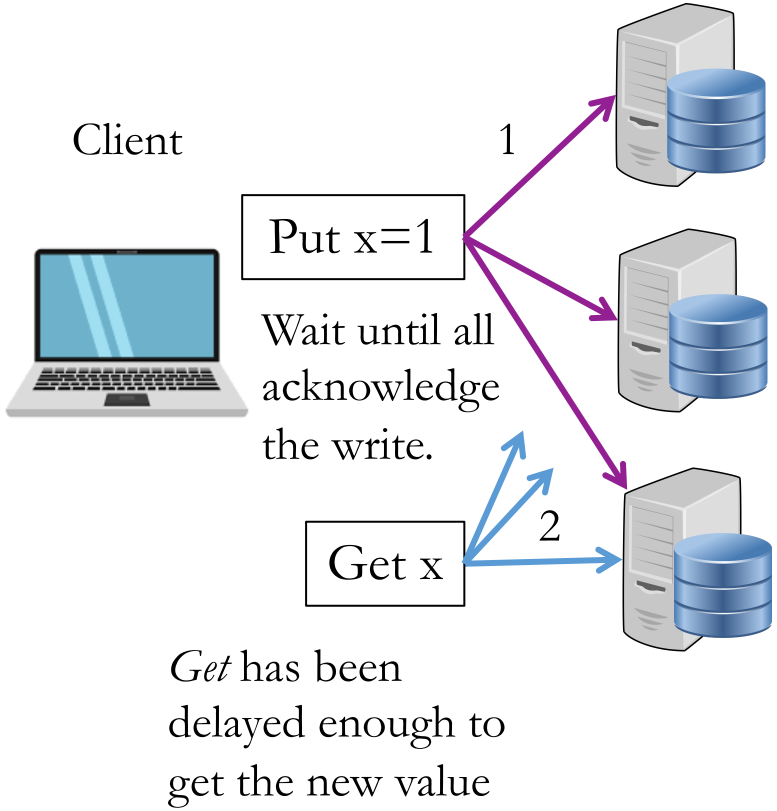
   - Simplest approach is for the client to send the request to all nodes and wait!

Waiting for Consistency

Monotonic Read:



- Read your Write:



Waiting for Consistency with **Quorums**

- A set of solutions for consistency in distributed DBs.
  - A **quorum** is a minimum percentage of a committee needed to act.
- Wait for an acknowledgement of consistent data from a certain number of replicas before considering the read/write completed.
  - **Prevents progress** until the replicas have a certain degree of consistency.

Write Quorum	Read Quorum	Optimized for
All	One	Fast reads
Majority	Majority	Balanced read/write performance
One	All	Fast writes

  - We send requests to **all** nodes but wait for the prescribed # of responses.

Majority-read, majority-write example (three nodes)

- Client wants to write X=1.
  - Sends three write requests to three replicas.
  - When an **acknowledgement** from **two replicas** is received it can proceed.
  - The third/last write proceeds in the background.
- Client reads X
  - Sends three read requests to three replicas.
  - At this point, one of the replicas may still have old data, but that's OK!
  - Client will be satisfied when it receives two responses.
  - If they're different, use the most recent one.
    (Every write is timestamped by the client.)
- Because writes are not finished until at least two acknowledge, there is at most one old value being stored. At least one of two must be new.

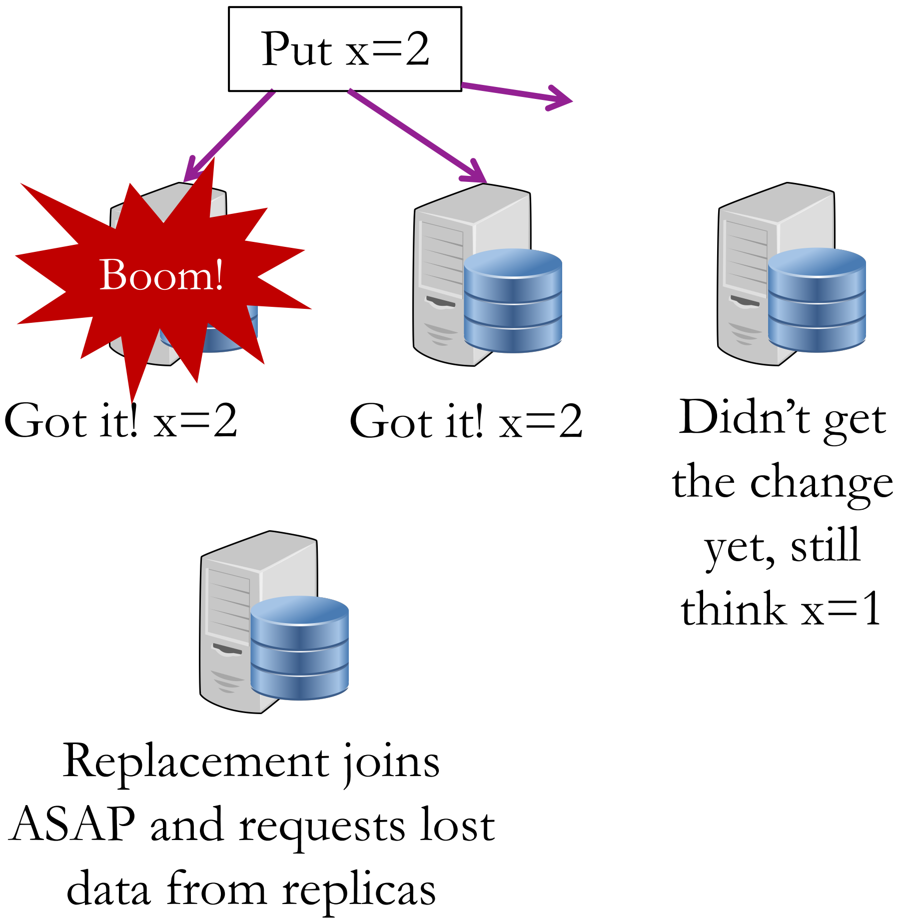
Single-read, unanimous-write example

- Client wants to write X=1.
  - Sends three write requests to three replicas.
  - Must wait until all **three replicas acknowledge** before proceeding.
- Client reads X
  - Sends three read requests to three replicas.
  - At this point, all three replicas must have received my previous write!
  - Client will be satisfied when it receives any **one** response.
  - Note that the responses from different nodes may be different (due to partial writes from other clients), but all will reflect data state after my own write.
    - Choose the latest value.
- Notice that writes are slow (max latency of the 3), but reads are fast (min latency of the 3).

Question: What happens if a DHT replica fails?

Example 1: write and read quorum of two (of three replicas).

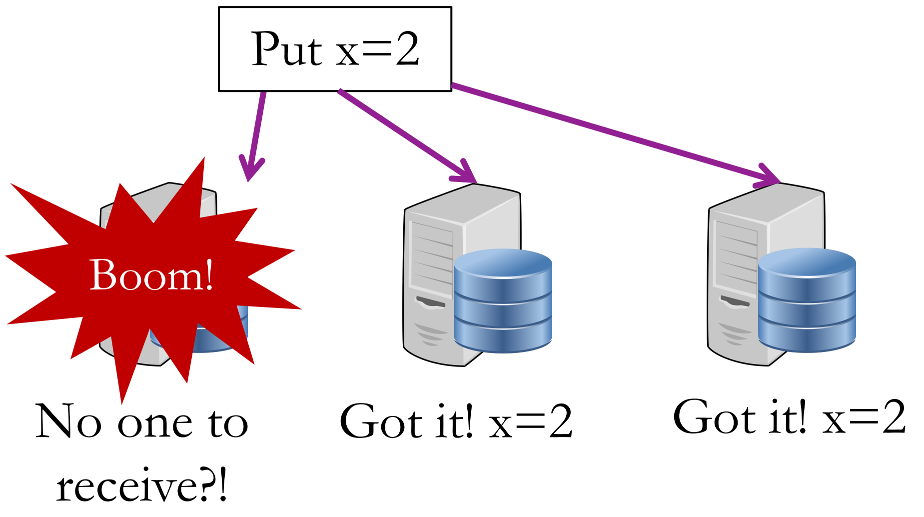
- Client performs a write, gets two ACKs and proceeds.
- At this point, replicas store two new values, and one old value.
- Now one of the written-to replicas fails!
- Can read and writes proceed?
- Yes. Two different values will be read, but client can choose the most recent one.
- The 3rd write will eventually be received, and two copies made available.

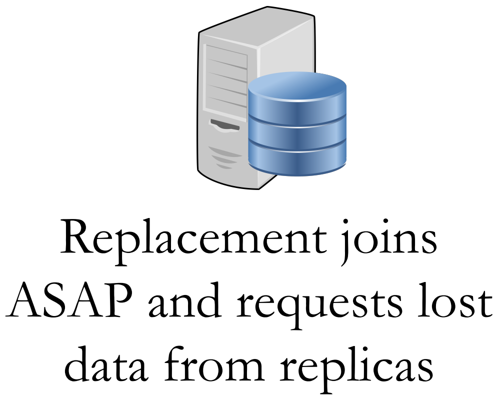


Question: What happens if a DHT replica fails?

Example 2: write quorum of three (read quorum of one)

- A replica fails!
- Can reads and writes proceed?
- Client performs a write, and cannot get three ACKs.
    - Write is impossible! (but reads can proceed)
    - Part of the system is stalled, *temporarily*.
- The write can be retried after a replacement joins the DHT and gets copies of all the data.

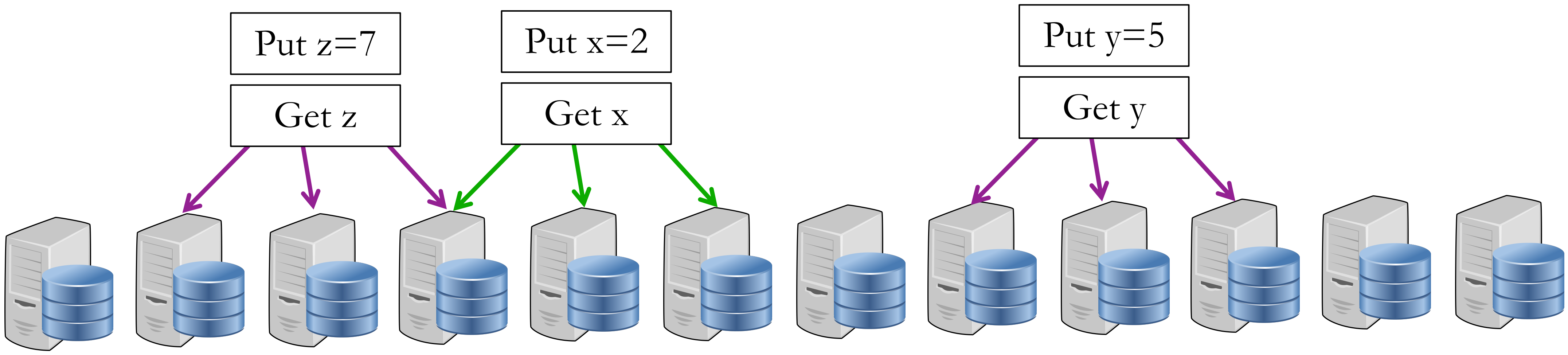




Reminder: Why is this scalable?

- My consistency examples showed only three nodes == three replicas.
- This was not a scalable system because all nodes stored all data.
- In practice you can have a very large number N of nodes, and a constant number of replicas for each data key.
- Hashing will map each data key to a subset (often 3) of the N nodes.
- Quorum only apply to replica nodes. "Write to all" means all replicas (3 nodes).

Why use more than 3 replicas?



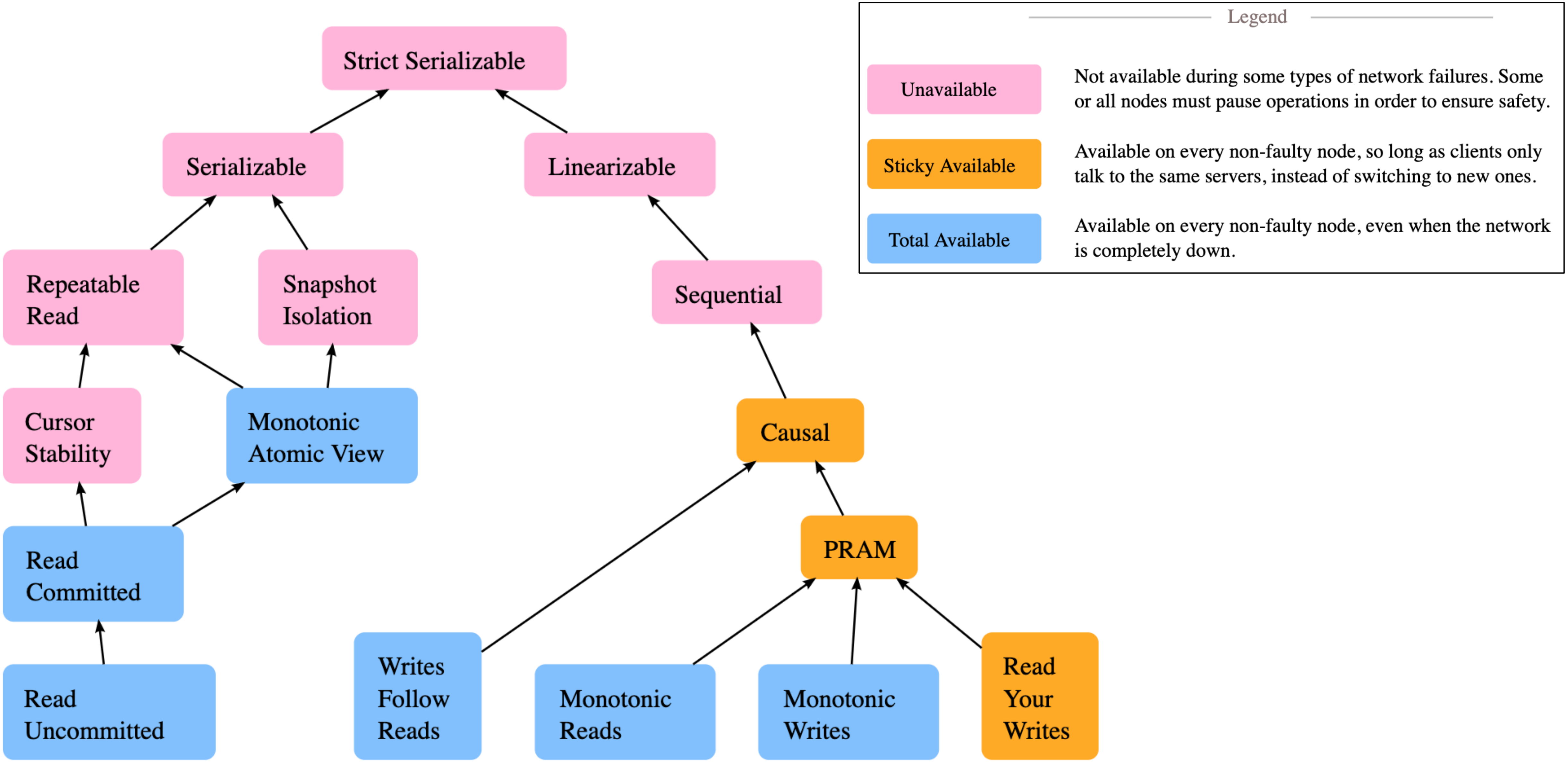
Distributed DB

Another way of looking at consistency

- A distributed system is **linearizable** if the *partial ordering* of distributed actions is preserved.
  - The distributed actors each know the order of their **own** actions.
  - This certain knowledge must never be contradicted by the distributed system.
  - This creates a partial ordering of all the events in the distributed system
- For example:
  - if Anita does A, B, C (in that order)
  - and Sam does S, T, U, (in that order)

  These happen concurrently
  - Then no one should see B before A, nor U before T, etc.
- Every observed **serialization** of the parallel activity must be agreeable to the individual actors. Observations will vary across the system.
  - There are all valid: (A, B, C, S, T, U) (S, A, B, T, U, C) (S, T, U, A, B, C)

Consistency is a subtle topic, with many models.



For more info on this fascinating topic

- CS-345 Distributed Systems
- Chapter 7 of Distributed Systems by van Steen and Tanenbaum.
- Part II (and Chapter 9 in particular) of the Designing Data Intensive Applications book by Kleppmann.
  - We covered the client-centric view of consistency.
  - Other models take a data-centric view.

- It's a nice mixture of CS theory and real system design.

NoSQL databases use DHTs or similar schemes

- Amazon DynamoDB
- Apache Cassandra
- ElasticSearch
- MongoDB (*hashed sharding* option)

Distributed filesystems can also use DHTs

- Filename/path is the key.
- Value is the file's contents.
- Hadoop HDFS, Google File System (Colossus, BigTable), Amazon S3

Recap: Distributed DB Consistency

- **Replication** of data ensures that a single failure does not lose data.
  - The more nodes you have, the more likely a failure!
- However, replication introduces **consistency** problems.
  - Tradeoff: must choose 2 of **C**onsistency, **A**vailability and **P**artition Tolerance.
- A distributed DB client, at very least, would want to achieve:
  - Montonic reads, monotonic writes, read your writes (together: linearizability).
- Ensure consistency by **waiting** for responses from multiple replicas.
- Different **quorum** levels (all, majority, one) trade delay of reads/writes and determine whether reads or writes are unavailable during recovery.
  - Cassandra DB lets programmer choose the quorum level for each read/write.
  - Other NoSQL databases are designed to use just one read/write strategy.