1.3 — Distributed Systems Fundamentals

Data Processing at Scale

2026-03-27

Data Processing at Scale — Day 1

Course website

https://vbergeron.github.io/data-processing-at-scale/

This presentation

https://vbergeron.github.io/data-processing-at-scale/1.3-distributed-systems.pdf

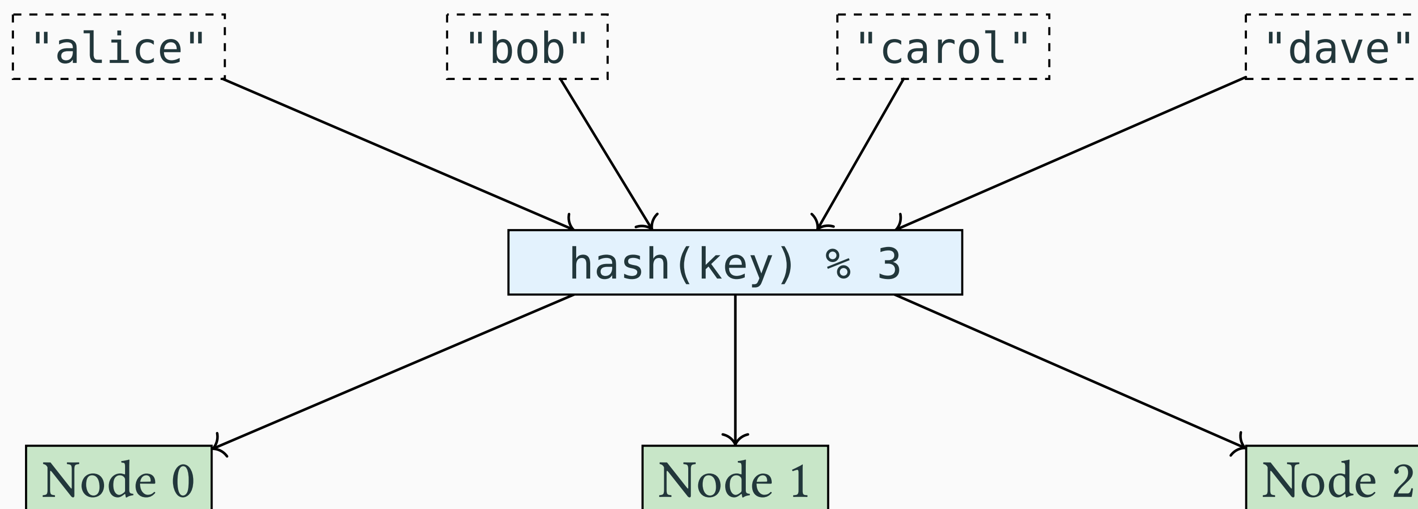
Your data doesn’t fit on one machine.

Partitioning

Why partition?

- One node can't hold all the data, or can't serve all the traffic
- **Partition** (aka **shard**): split data so each node owns a subset
- Goal: distribute load **evenly** — avoid hot spots
- Often make sense economically.

Hash partitioning

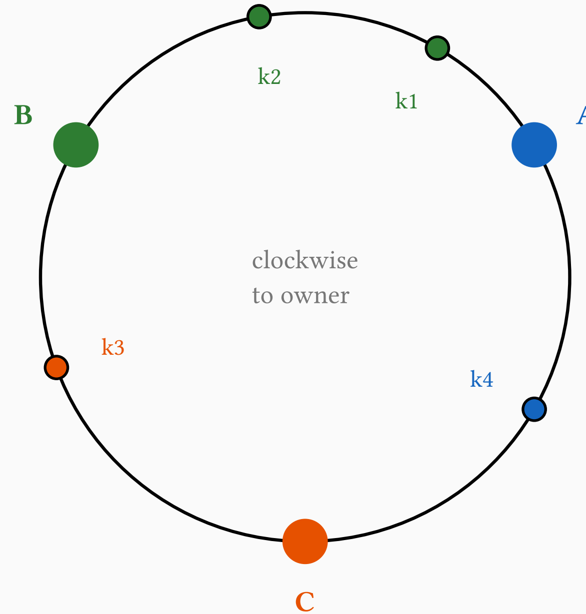


- Uniform distribution — good for point lookups
- `hash("alice-order-1")` and `hash("alice-order-2")` land on different nodes
- Simple to implement, hard to rebalance

The rebalancing problem

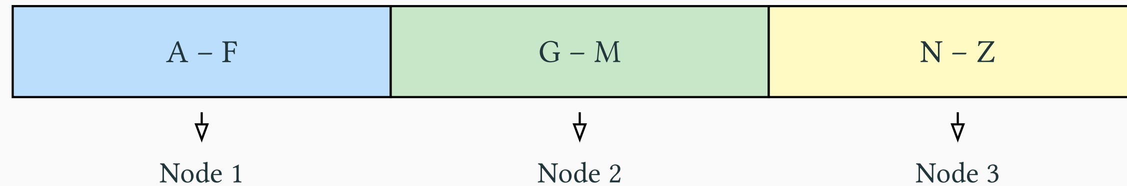
- Adding a node means movings most of the keys
- `P(X moves) = N / (N + 1)`
- With 100 TB across 10 nodes, adding node 11 reshuffles 90 TB

Consistent hashing



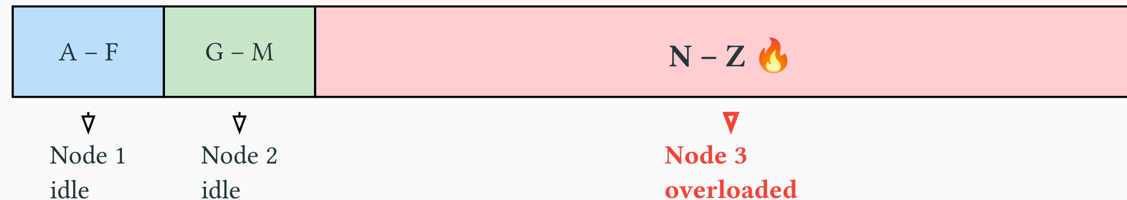
- Each key walks clockwise to the next node — that node owns it
- Adding a node moves only the keys in its arc (1/N of total)
- **Virtual nodes**: each physical node gets multiple positions on the ring

Range partitioning



- Keys are sorted; each node owns a contiguous range
- Enables efficient **range queries** — scan within one partition
- Risk: ranges can be uneven (time-series keys cluster on “today”)

Hot spots — the recurring nightmare



- **Hot spot** / **skew**: one partition gets disproportionate traffic
- Classic: time-series by date (all writes go to "today's" partition)
- Celebrity problem: `user_id = @taylorswift` ⟶ 50% of reads
- Mitigation: key salting, sub-partitioning, application-level splitting

Hash vs range — decision matrix

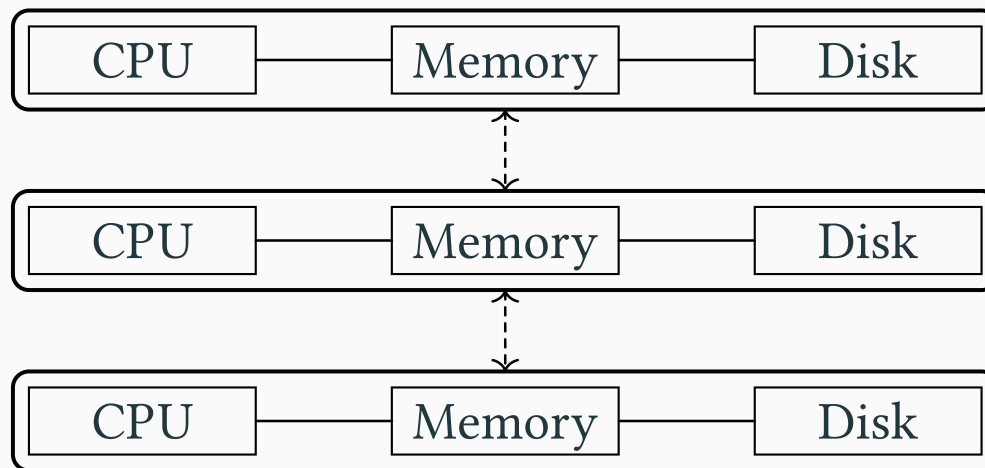
	Hash	**Range**
Distribution	Uniform	Depends on key distribution
Range queries	✗ Not supported	✓ Efficient
Rebalancing cost	Low (consistent hashing)	Medium (split/merge)
Hot-spot risk	Low	High (without salting)

Data is split across machines.
But what if a node dies?

A single machine is simple

- One clock, one address space, shared fate
- Failure is total: the whole machine crashes or nothing does
- Every function call returns — or the process dies

Partial failure — the new problem

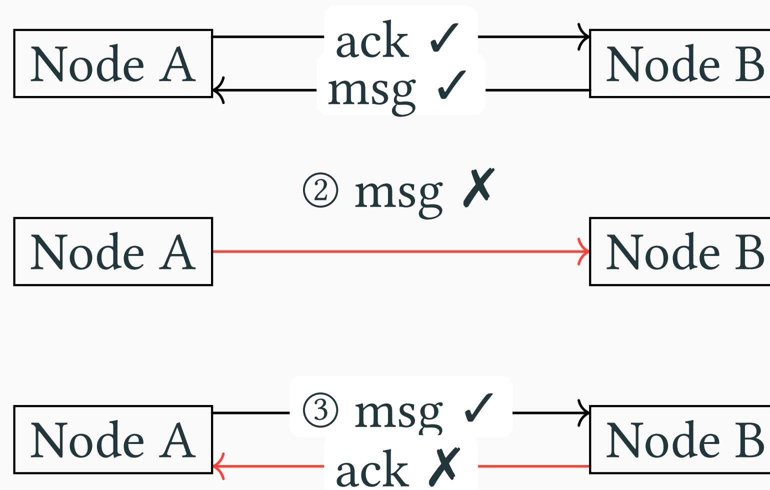


The 8 fallacies of distributed computing

"Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences."

— Peter Deutsch & James Gosling, *The Eight Fallacies of Distributed Computing*, Sun Microsystems, 1994.

Fallacy 1 — The network is reliable



- A cannot distinguish case ② from case ③
- The sender **never knows** if the message was processed

Fallacy 2 — Latency is zero

Operation	**Latency**	**CPU cycles @ 3 GHz**
L1 cache reference	1 ns	3
RAM access	100 ns	300
SSD random read	100 µs	300 000
Datacenter round-trip	500 µs	**1 500 000**
Cross-region (EU↔US)	80 ms	**240 000 000**

- A network round-trip costs **millions** of CPU cycles
- Every RPC, every shuffle, every checkpoint pays this cost
- Batching and locality are not optimizations — they are necessities

Fallacy 3 — Bandwidth is infinite

- A 10 Gbps link moves 1.25 GB/s — a single Spark shuffle can saturate it
- **Noisy Neighbors**: your job competes with every other tenant
- In the cloud, cross-AZ and cross-region bandwidth is metered and throttled

Fallacy 4 — The network is secure

- Any node on the network can send malformed data — or nothing at all
- TLS, authentication, and firewalls add latency and operational complexity
- In data pipelines, the trust boundary is often **inside** the datacenter

Fallacy 5 — Topology doesn't change

- Nodes join and leave: autoscaling, rolling deployments, hardware failures
- IP addresses change, DNS caches go stale
- Hence the need for **Service discovery** (Consul, ZooKeeper, Kubernetes DNS)

Fallacy 6 — There is one administrator

- Cloud infrastructure spans teams, organizations, and providers
- Different SLAs, upgrade schedules, and security policies
- No single person controls the full path from producer to consumer

Fallacy 7 — Transport cost is zero

- Serialization, compression, encryption — all cost CPU
- Cloud egress fees: $0.01–0.09 per GB across regions
- Moving data is often more expensive than processing it

Fallacy 8 — The network is homogeneous

- Datacenter links ≠ cross-region links ≠ public internet
- Different MTUs, reliability guarantees, and congestion behavior
- A pipeline tuned for one network topology may fail in another

The 8 fallacies — recap

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

Peter Deutsch (+ James Gosling), 1994 — the first three dominate data processing.

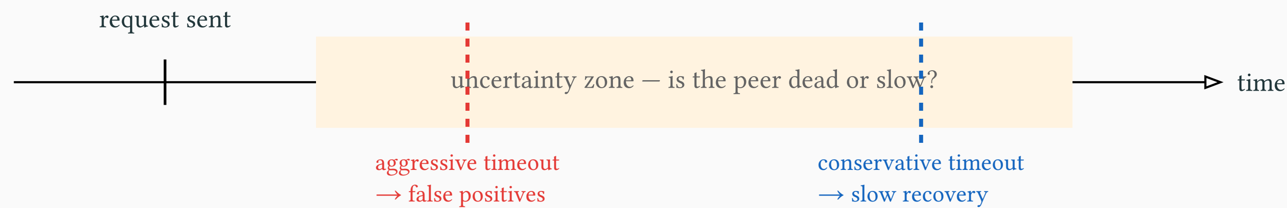
Failure taxonomy

- **Crash failure**: Node stops responding -> Most systems handle this,
- **Omission failure**: Node loose messages -> TCP masks some of these,
- **Byzantine failure**: Node sends malicious messages

Failure taxonomy

- Most data systems only tolerate **crash** failures
- Byzantine fault tolerance exists (**BFT**) using cryptography
- If you are asked for a practical case for **blockchains**, this is it.

Timeouts — guessing at failure



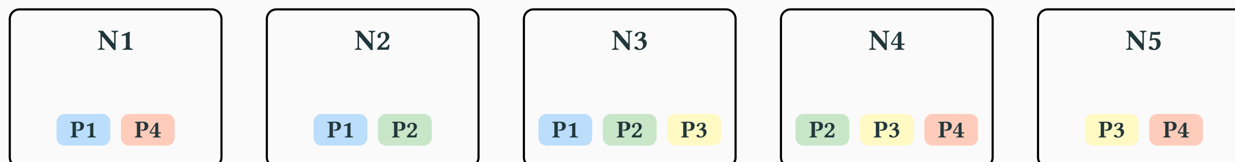
- **Failure detector** — a component that guesses whether a remote node is alive
- Too aggressive ⟶ healthy nodes declared dead (thrashing)
- Too conservative ⟶ long outages before recovery starts

We can’t prevent failures.

Replication

Why replicate?

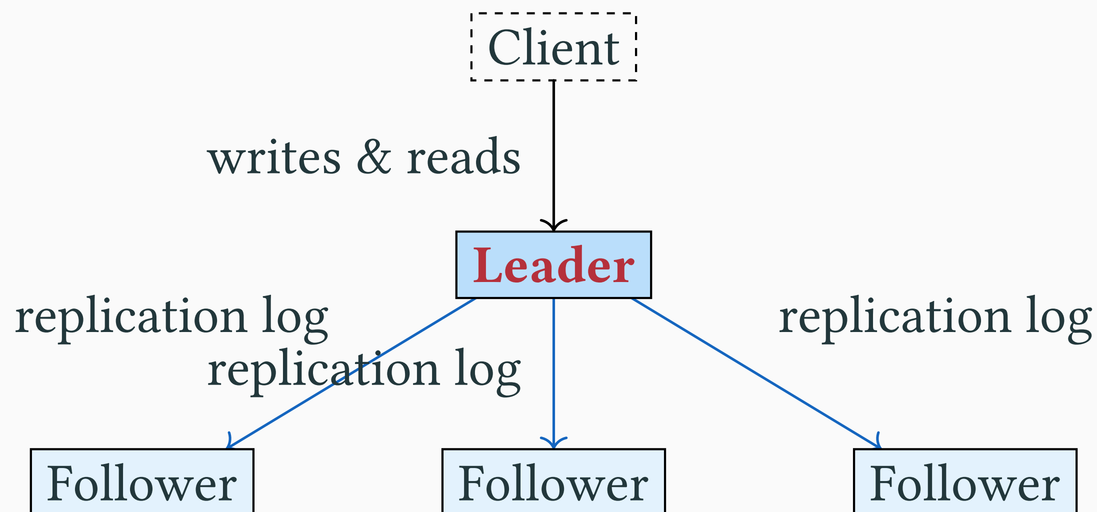
Why replicate?



4 partitions × RF 3 across 5 nodes

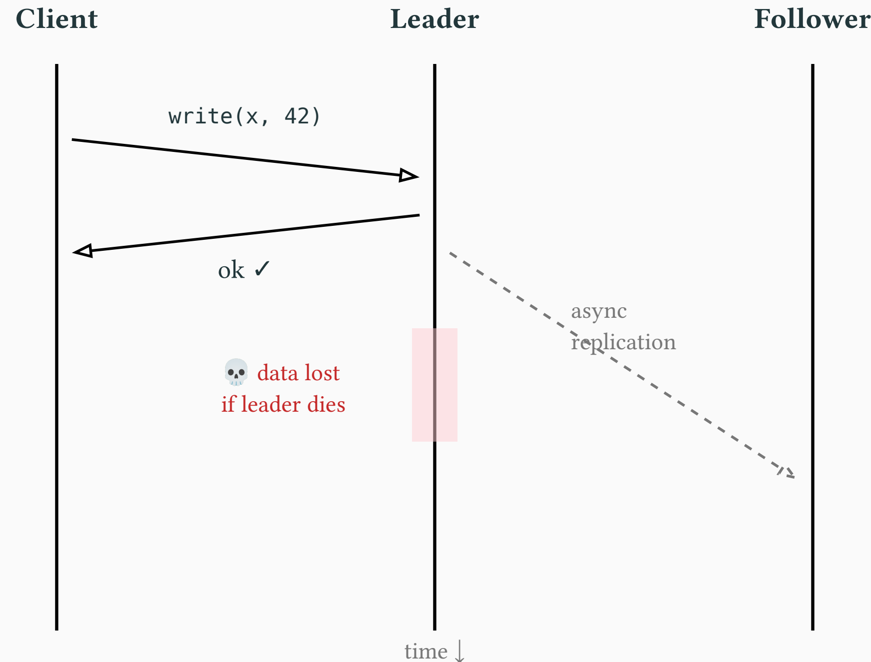
- **Durability**: survive hardware failure — data exists on multiple machines
- **Availability**: serve reads even if one replica is down
- **Latency**: read from the geographically closest copy
- Partitioning and replication are **orthogonal** — you almost always use both

Leader / follower replication



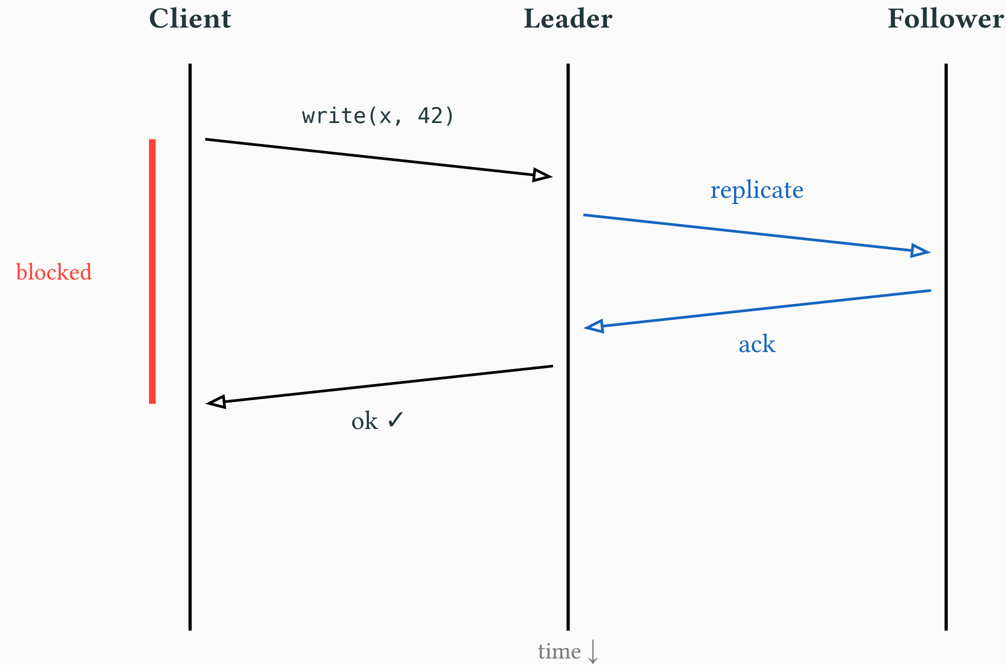
- One **leader** accepts all writes; **followers** replicate the write-ahead log
- No write conflicts — the leader serializes every mutation into a single ordered log
- Reads can go to any replica — scale read throughput by adding followers
- Simple mental model: behaves like a single node with backup copies

Asynchronous replication



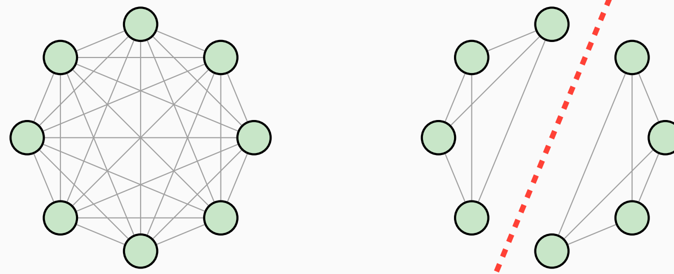
- Leader acks **before** replication — fast writes, low latency
- If the leader crashes in the danger zone, acknowledged writes are **permanently lost**
- Used by: PostgreSQL (default), MySQL, MongoDB, Redis

Synchronous replication



- Leader acks **after** replication — data is durable on multiple nodes before the client proceeds
- Cost: every write pays a round-trip latency penalty; one slow follower blocks everything
- Used by: etcd, ZooKeeper (Raft/ZAB), PostgreSQL (`synchronous_commit`)

A network partition

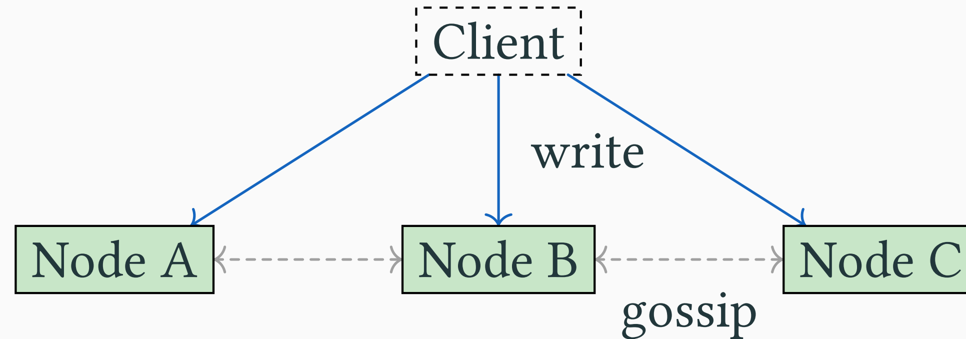


- Both sides are alive, but they **cannot talk to each other**
- Clients can still reach each node independently
- The system must now choose: consistency or availability?

Split-brain

- Both sides elect a leader — two leaders accept **conflicting writes**
- When the partition heals, the system must reconcile divergent state
- Solutions: fencing tokens, consensus protocols (Raft, Paxos)

Leaderless replication



- No designated leader — the **client** sends writes to multiple nodes directly
- Nodes exchange updates among themselves via **gossip** (anti-entropy)
- No failover needed: if one node is down, the others keep serving
- But how does the client know a read is up to date?

Quorum reads & writes — W + R > N

- **Quorum**: a minimum number of nodes that must participate in an operation
- Write to **W** nodes, read from **R** nodes, out of **N** total replicas
- If `W + R > N`, at least one node in every read has the latest write
- The client picks the value with the highest version number

Tuning W, R, N

Config	Consistency	Write speed	Read speed
N=3, W=2, R=2	Strong	Medium	Medium
N=3, W=1, R=1	Eventual	Fast	Fast
N=3, W=3, R=1	Strong	Slow	Fast
N=3, W=1, R=3	Strong	Fast	Slow

- Every configuration is a **trade-off dial** between latency and safety
- W=1 is “fire and forget” — fast writes, but data can be lost
- R=1 is “read from anyone” — fast reads, but might be stale

Leader/follower vs leaderless — when to use which

	Leader/follower	**Leaderless (quorum)**
Write throughput	One bottleneck	Distributed
Read scaling	Add followers	Any node serves reads
Failover complexity	High (leader election)	Low (no single leader)
Consistency	Depends on sync/async	Tunable via W, R, N

Data is split and copied.

But what can we actually guarantee?

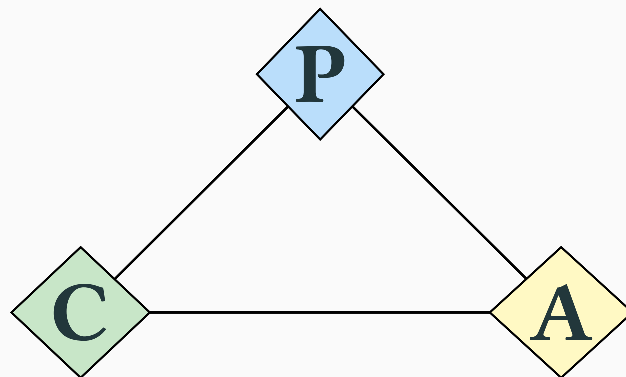
CAP & Consistency

Three properties

C - **Consistency**	Every read returns either most recent write or an error
A - **Availability**	Every request to a non-crashed node gets a response
P - **Partition tolerance**	The system keeps working despite network splits

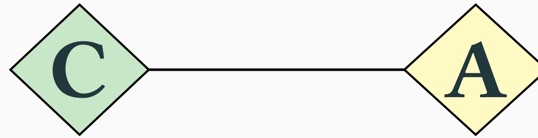
⚠️ “Consistency” here is **not** the C in ACID — different concept, same word.

The CAP theorem



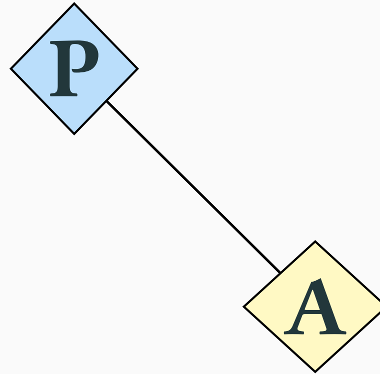
If a system is **Partition Tolerant**,
A system *must* choose **Consistency** or **Availability**.

CA systems



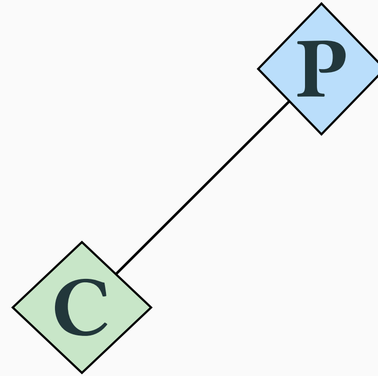
Single node systems: PostgreSQl, Redis (single node), DuckDB

AP systems



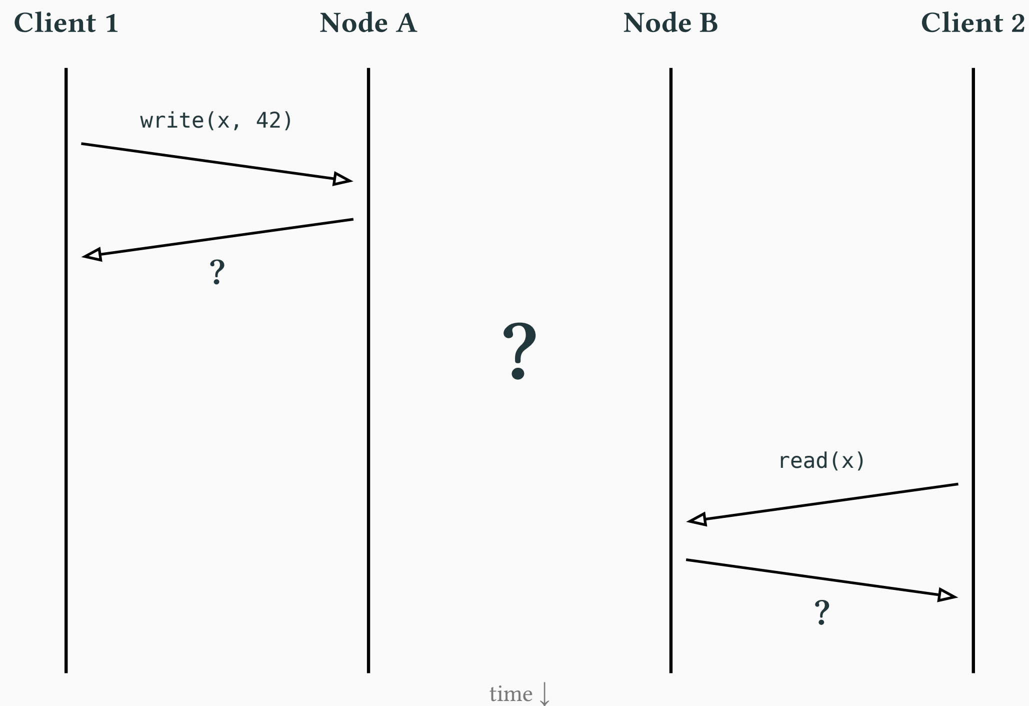
Cassandra, DynamoDB, CouchDB, ElasticSearch

CP Systems



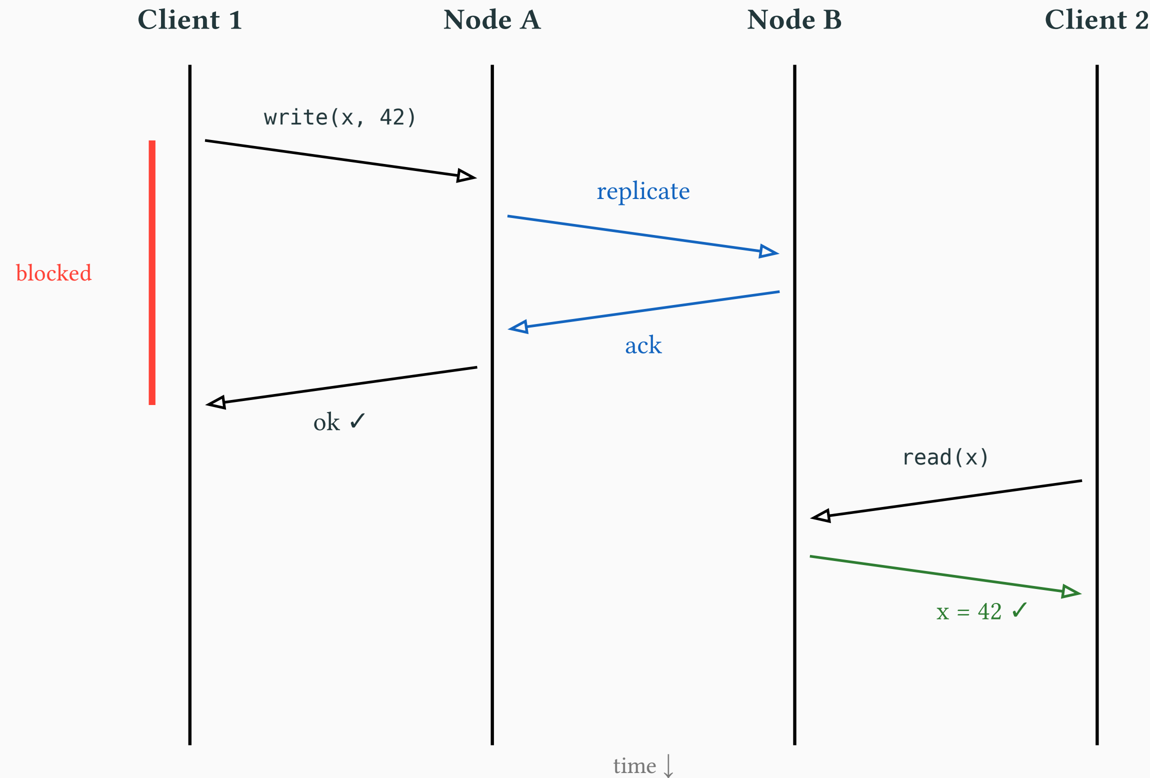
ZooKeeper, etcd, HBase, FoundationDB

Consistency models — what can a reader see?



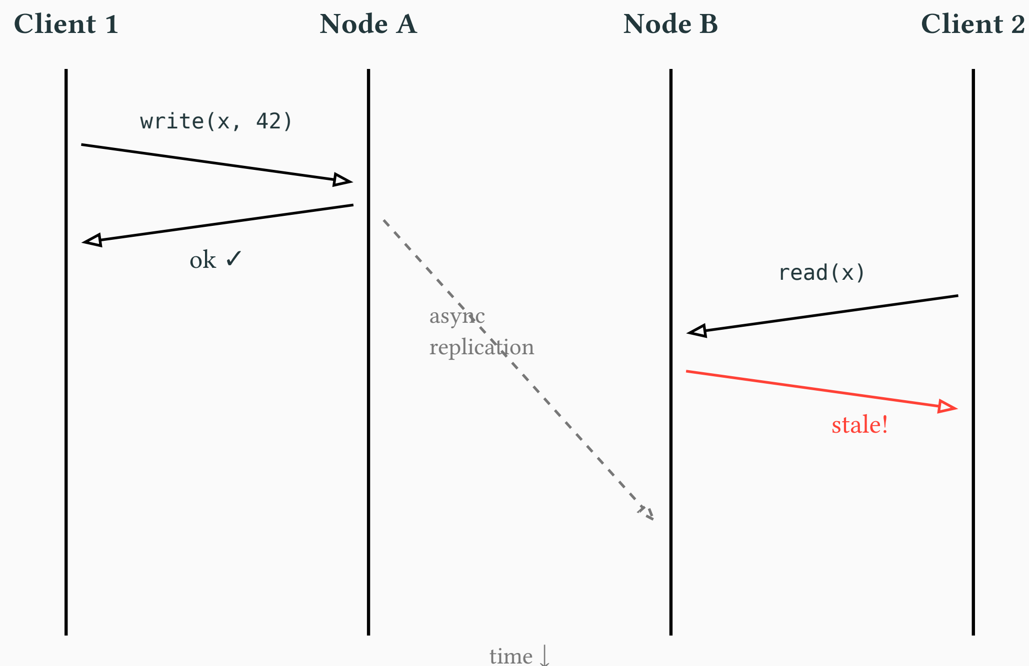
When does A respond? What do A and B exchange? What does B return?

Strong consistency — linearizability



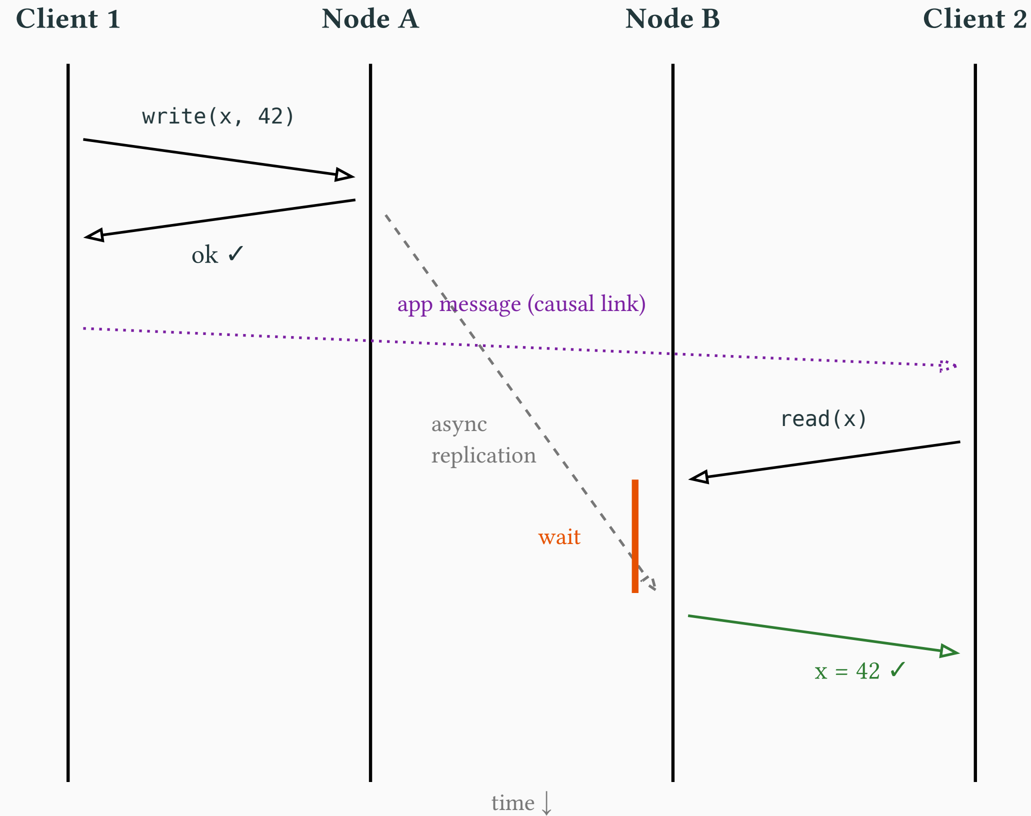
- **Linearizability**: behaves as if there is a single copy of the data
- The leader waits for replication acknowledgment before responding
- Cost: every write pays a round-trip latency penalty

Eventual consistency



- If no new writes arrive, all replicas **eventually converge**
- No bound on when — could be milliseconds, could be minutes
- Cheap and fast, but dangerous when you need read-your-writes

Causal consistency



- The follower delays its response until replication catches up
- Writes are fast (like eventual), reads are causally correct (like strong)

Choosing a consistency model

Model	Latency	Safety	Good for
Strong (linearizable)	High	Highest	Bank transfers, distributed locks
Causal	Medium	Medium	Collaborative editing, social feeds
Eventual	Low	Lowest	Caches, analytics counters, DNS

- Ask: “What happens if a reader sees a stale value?”
- If the answer is “nothing bad” $\longrightarrow$ eventual is fine
- If the answer is “we lose money” $\longrightarrow$ you need strong

One sentence to remember

Too big → too fragile → too complex →
your job is to choose the right trade-offs.

Vocabulary recap

Term	Definition
Partial failure	Some components fail while others keep running
CAP theorem	During a network partition, choose consistency or availability
Linearizability	Behaves as if there is a single copy of the data
Eventual consistency	Replicas converge if no new writes arrive — no time bound
Causal ordering	Causally related operations are seen in the same order everywhere
Partition / shard	A subset of data assigned to one node
Consistent hashing	Hash ring that minimizes key movement when nodes change
Replication lag	Delay between a write on the leader and its application on a follower
Failover	Promoting a follower to leader when the current leader fails
Split-brain	Two nodes both believe they are the leader
Quorum	Minimum number of nodes that must agree for an operation to succeed

Labs

1.3.1-distributed-kv

1.3.2-batch-processing