Distributed Systems 101

@lvh

_@lvh.io

Slides

`www.lvh.io/DistributedSystems101`

Introduction

Who am I?

PyCon

Rackspace

AutoScale

AutoScale

- Distributed system
- Manages distributed systems
- Running on distributed systems
- Orchestrating distributed systems
- At scale

Goals

"Just enough" distributed systems

- to whet your appetite
- to shoot yourself in the foot

Goals

- *Not* exhaustive, *not* pedantically correct
- Give you an idea of what there is & where to look
- Convince you distributed systems are tricky

Distributed systems?

What *is* a distributed system?

[…] when a machine I've never heard of can cause my program to fail.

– Leslie Lamport

Paradox

- Why do we use them? Reliability!
- Experts' primary concern? Failure!

Fundamental constraints

1. Information travels at c
2. Components fail

Fallacies

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.

Examples of distributed systems

- Basically everything (e.g., your laptop)
  - Speed of light isn't infinite
  - RAM is *all the way over there*
- Typically:
  - Any system with > 1 machine
  - Connected via network

Bad news

Theory and consequences

CAP theorem

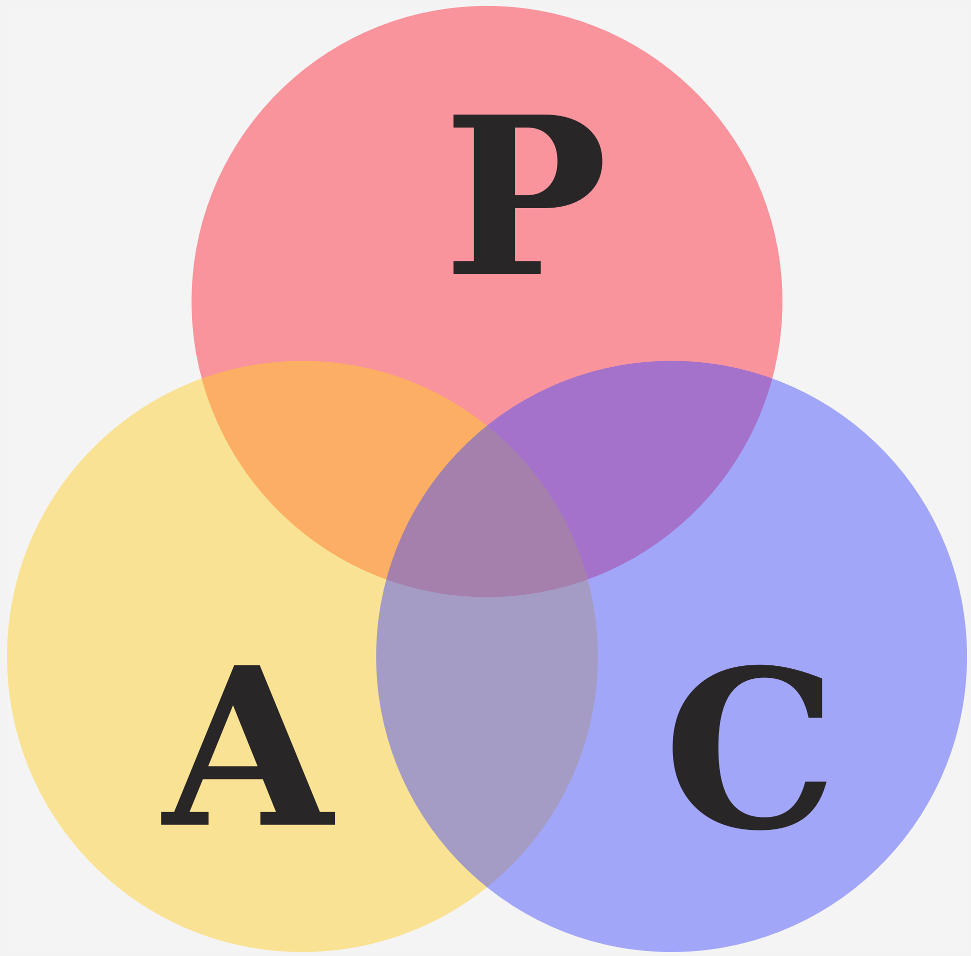
Pick any two:

- Consistency
- Availability
- Partition tolerance

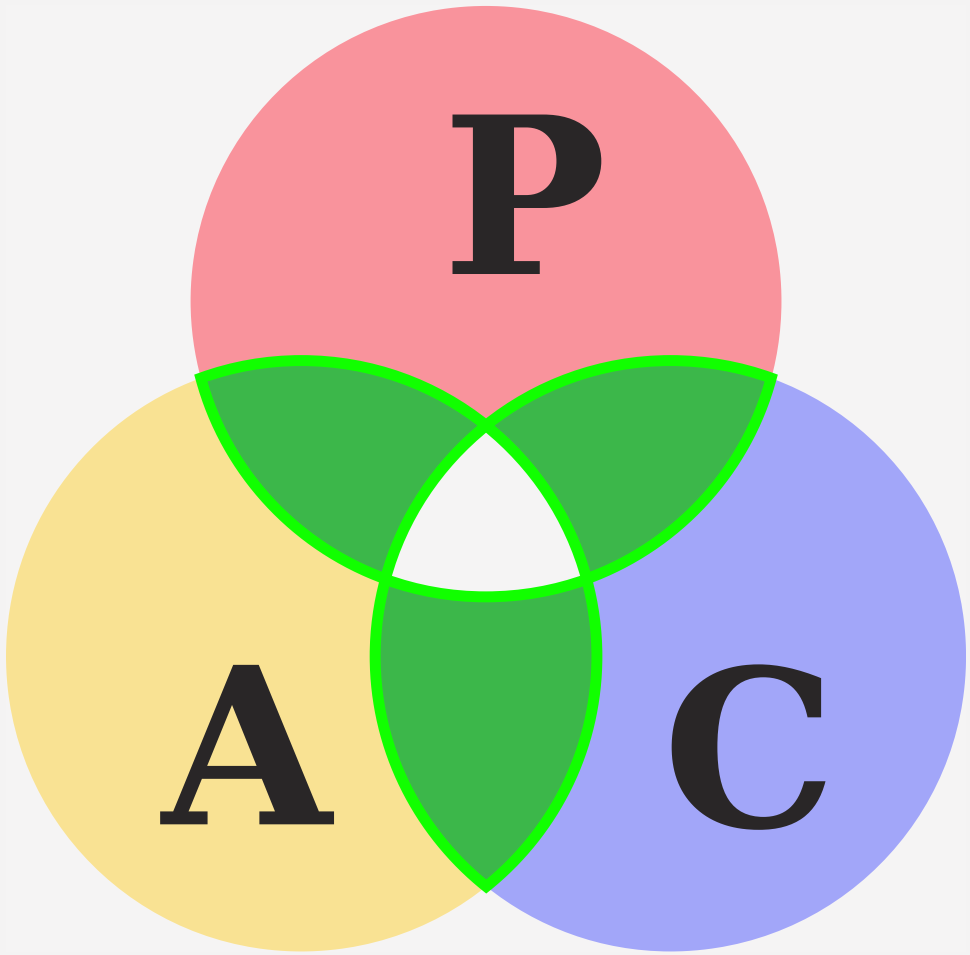
What does that even mean?

- C: linearizability (~ local behavior)
- A: all active nodes answer every query
- P: resistance to failures

Pick any two



Pick any two



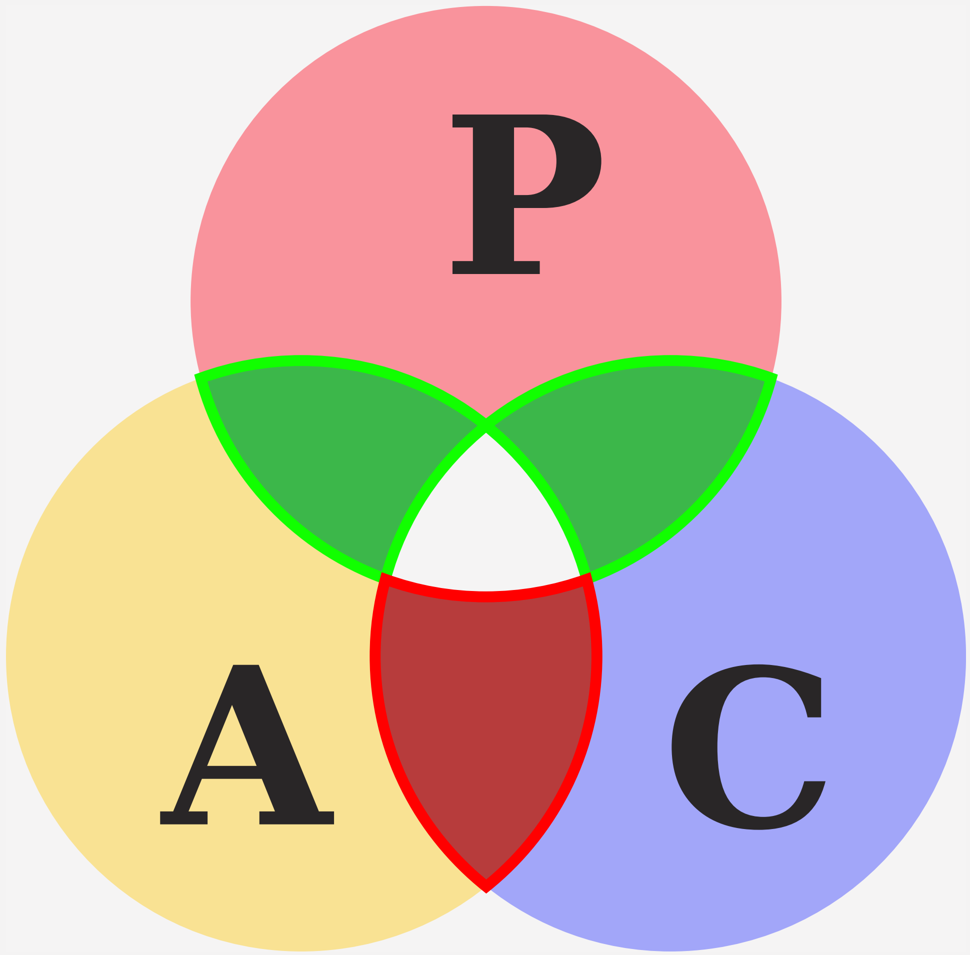
Can't sacrifice partition tolerance

- Partition tolerance is failure tolerance
- Networks, nodes fail all the time
- Latency happens; indistinguishable
- $P(\text{no failures}) < 1 - P(\text{one node works})^N$
  - Cascades, Hurst exponent

CA (!P)

"half of the time it doesn't actually work"

Pick any two: AP or CP



CP example: Zookeeper

Consistent ops that sometimes fail (!A)

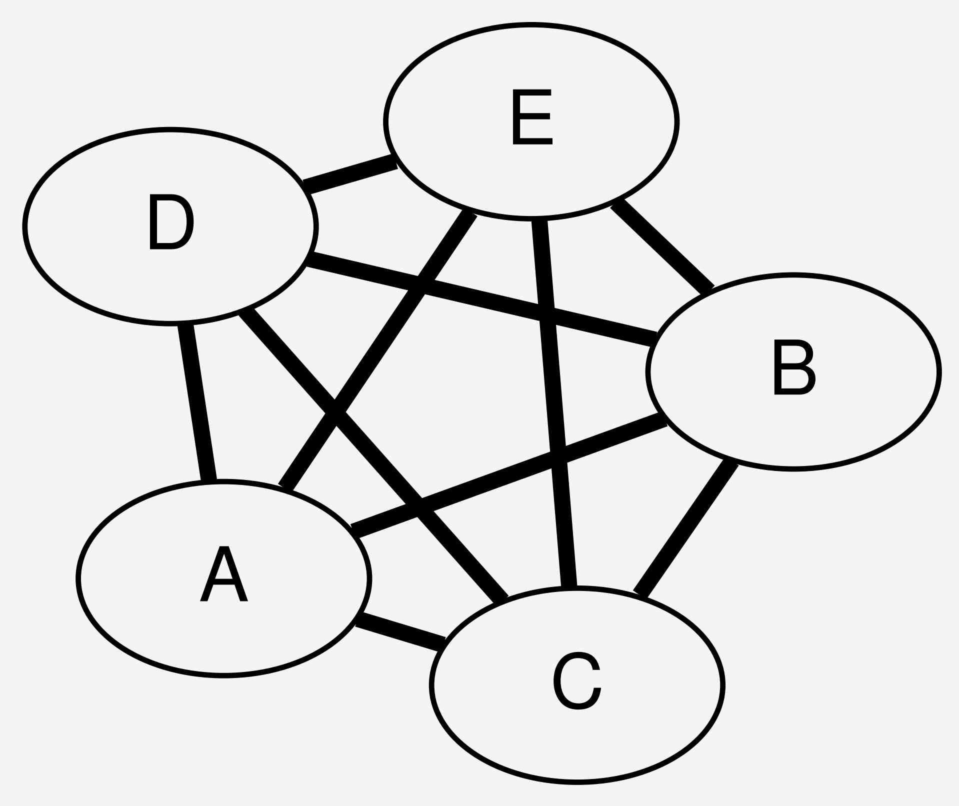
AP example: Cassandra

Inconsistent ops (!C) that (usually) succeed

Informally

Let's look at a 5 node cluster

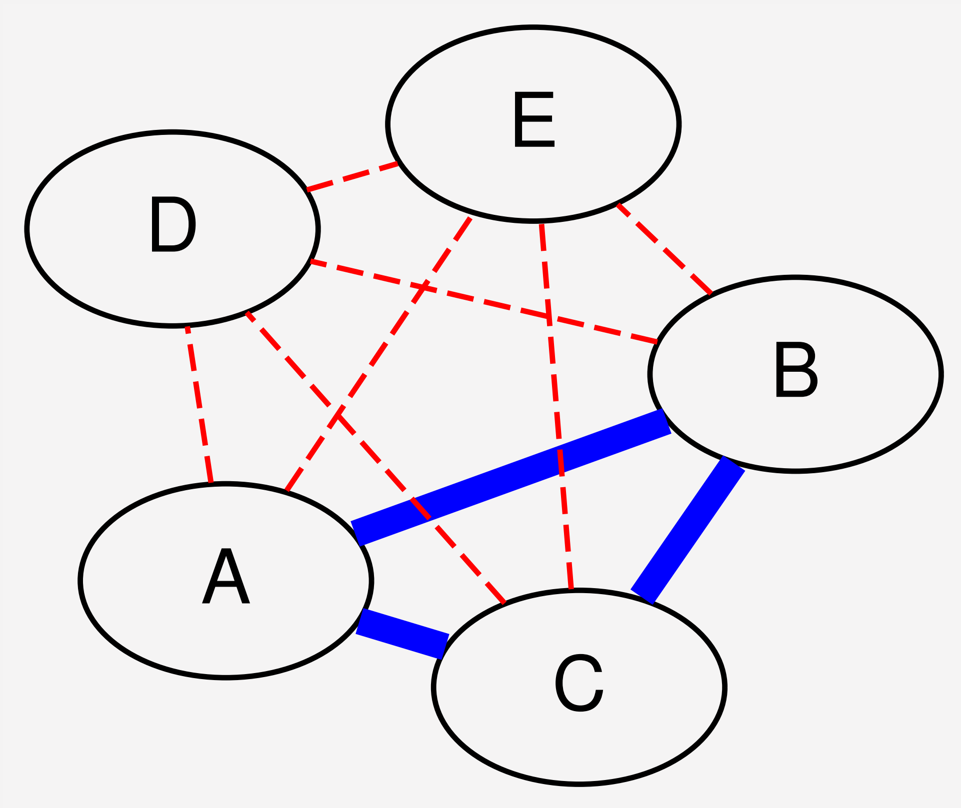
5-node connected cluster



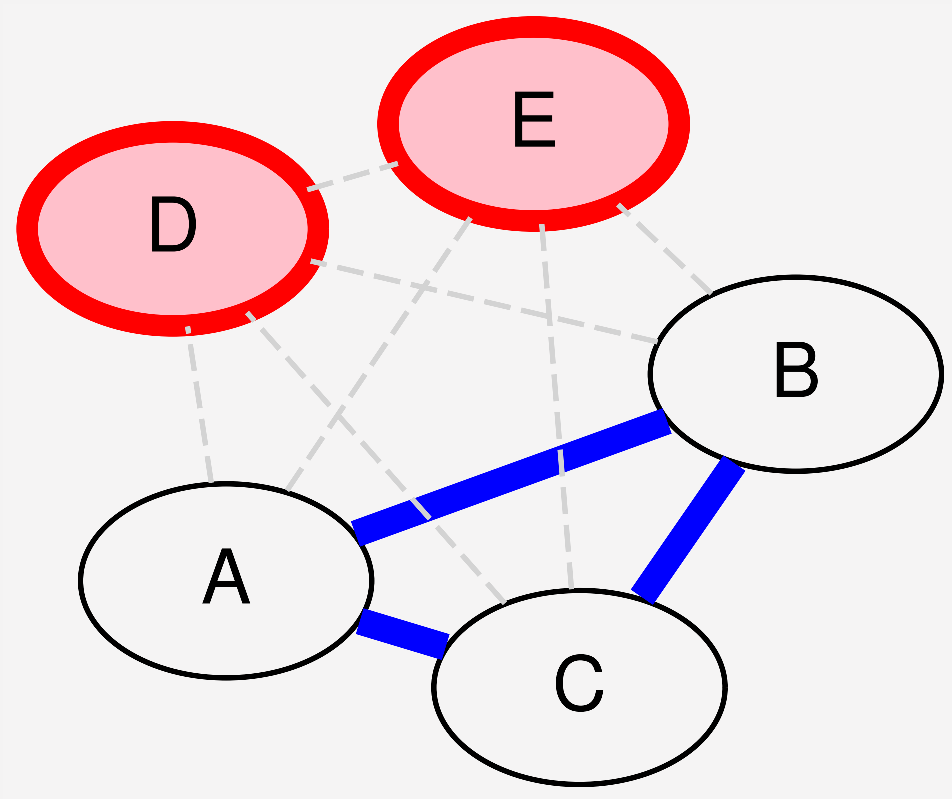
What happens when stuff fails?

- I still want to read/write!
- Idea: (N+1)/2 quorum

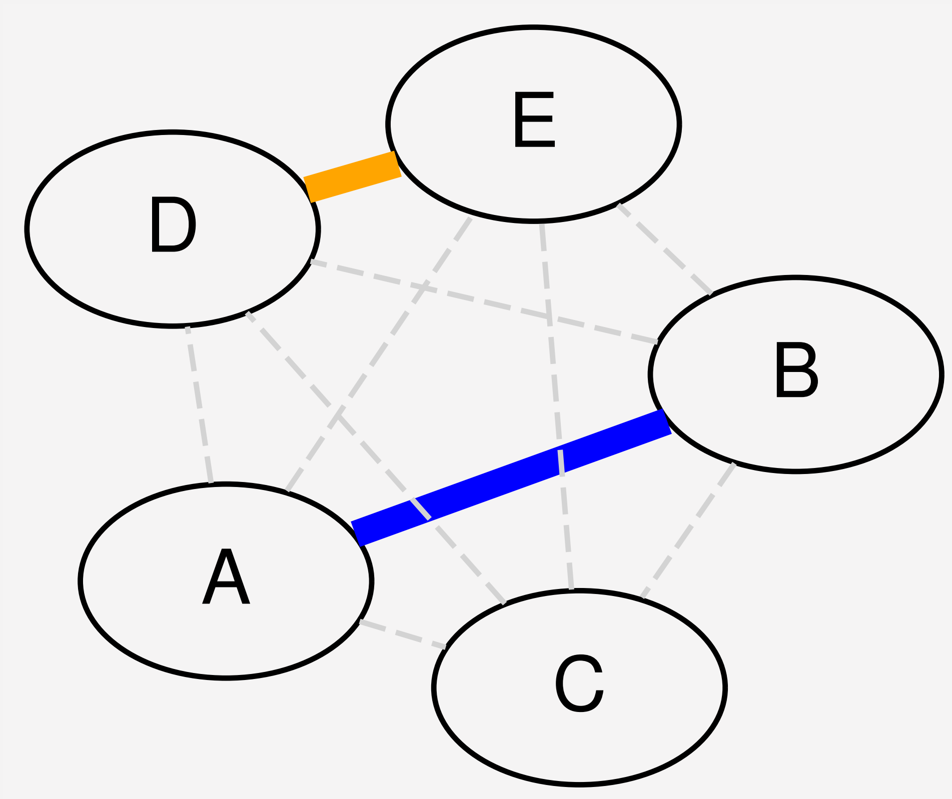
Some failures



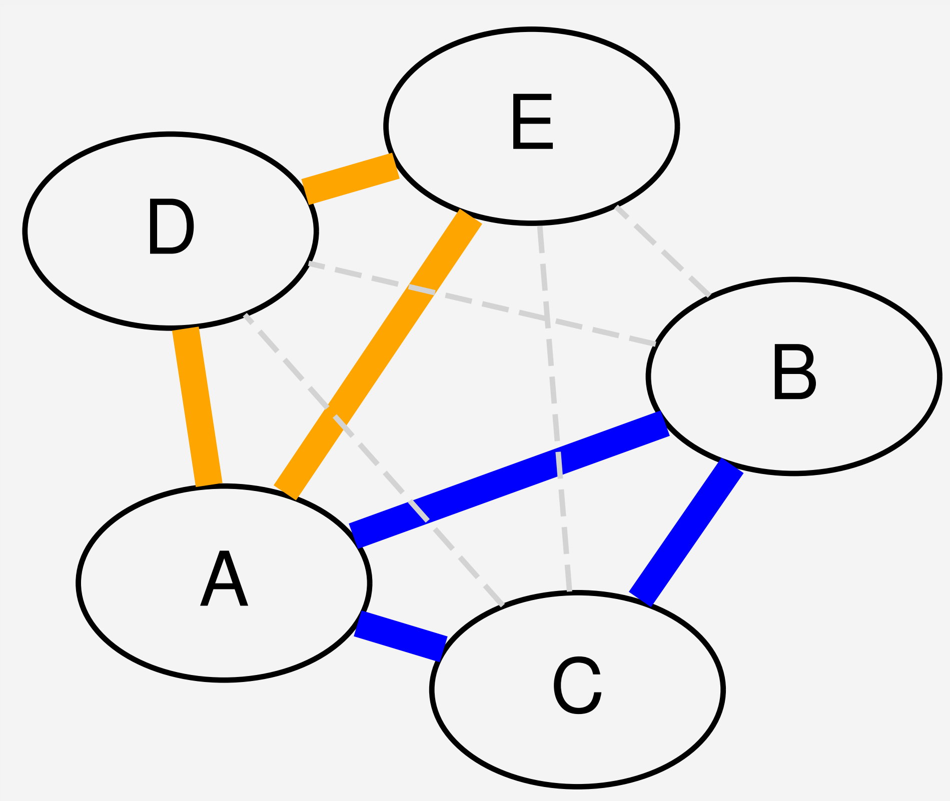
Failures are indistinguishable



Too many failures, no quorum



Simple quorum isn't enough



Not far-fetched

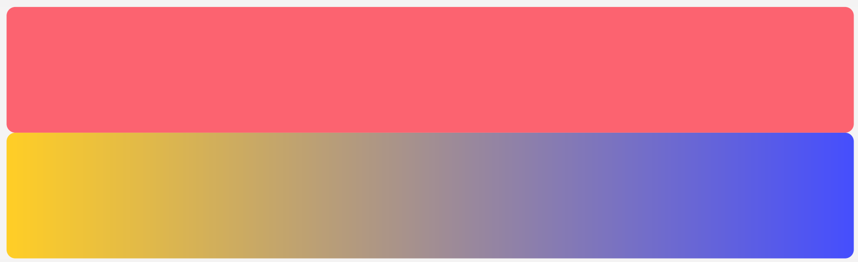
Real failures are:

- partial
- complex

Back to reality

- CAP's C is linearizability
- CAP's A is any *op* on *any* node
- These are very strong guarantees!

Gradations

AP  **CP**

Trade-offs

Availability	Consistency
Performance	Ease of reasoning
Scalability	Transactionality

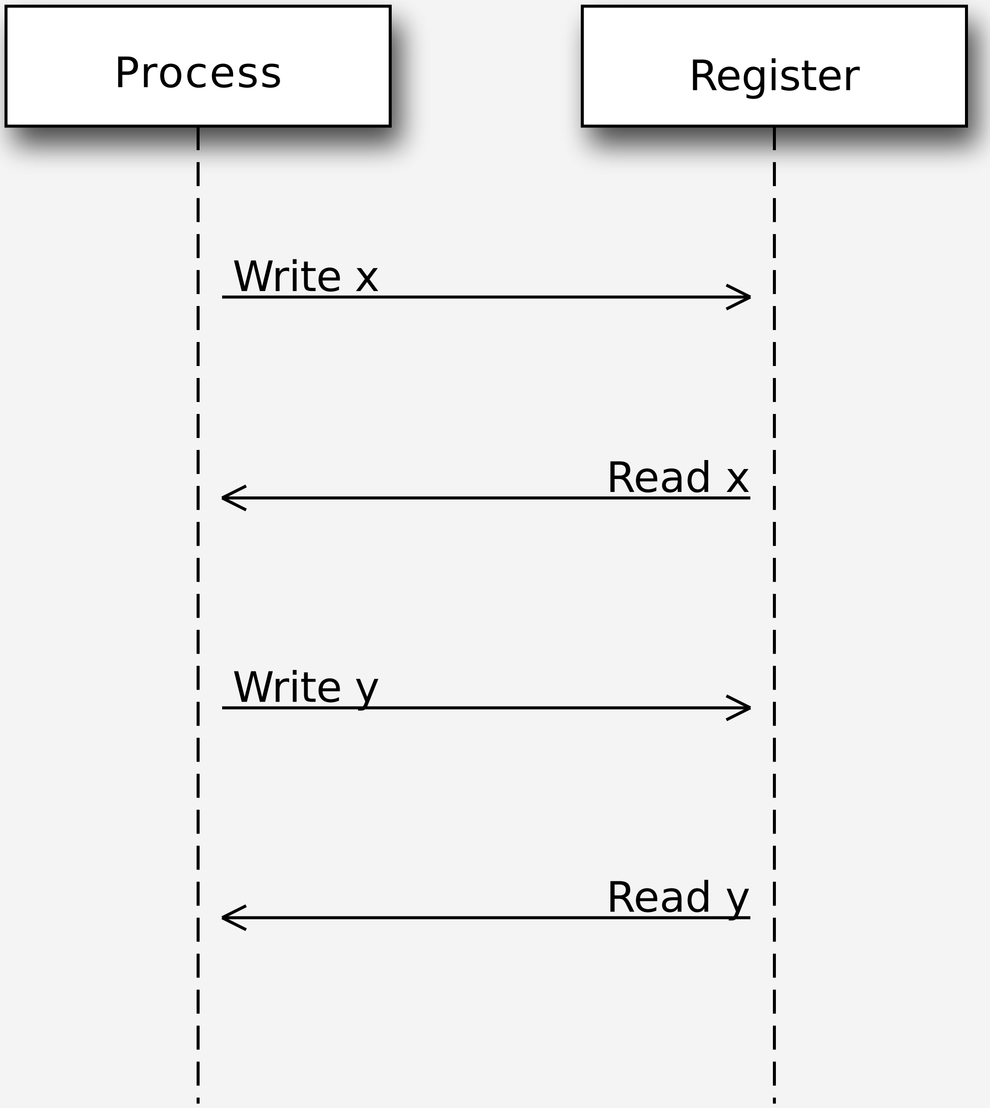
No universally correct choice!

Example of creative sacrifice: `etcd`

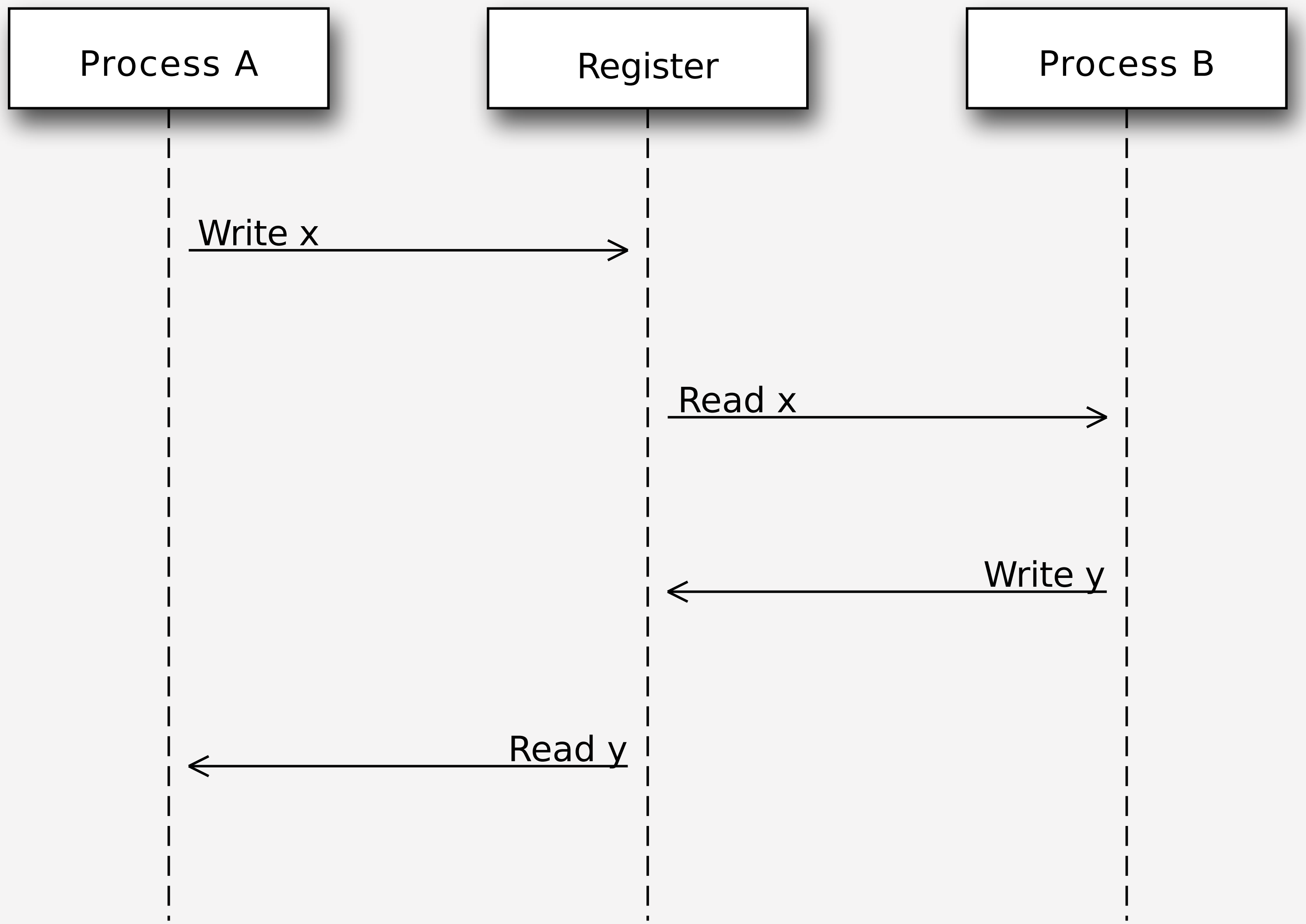
- Normally: consistency all the way
- Option of doing inconsistent reads
- Maybe get some stale data
- … but still works under partial failure

Consistency models

One process, one register



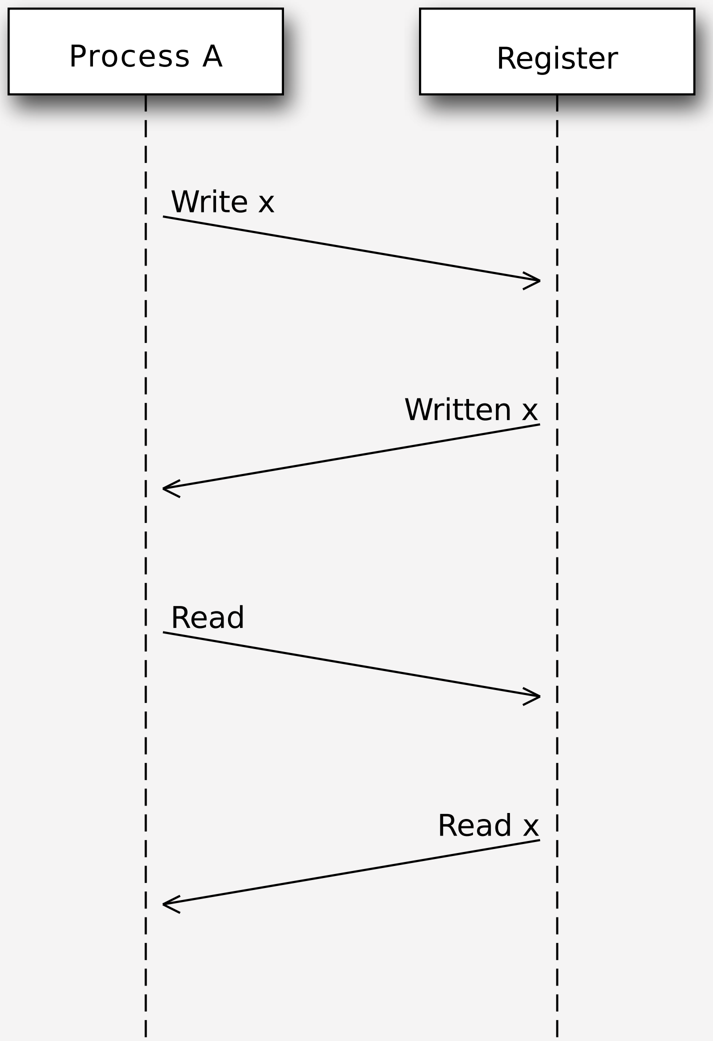
Two processes, one register



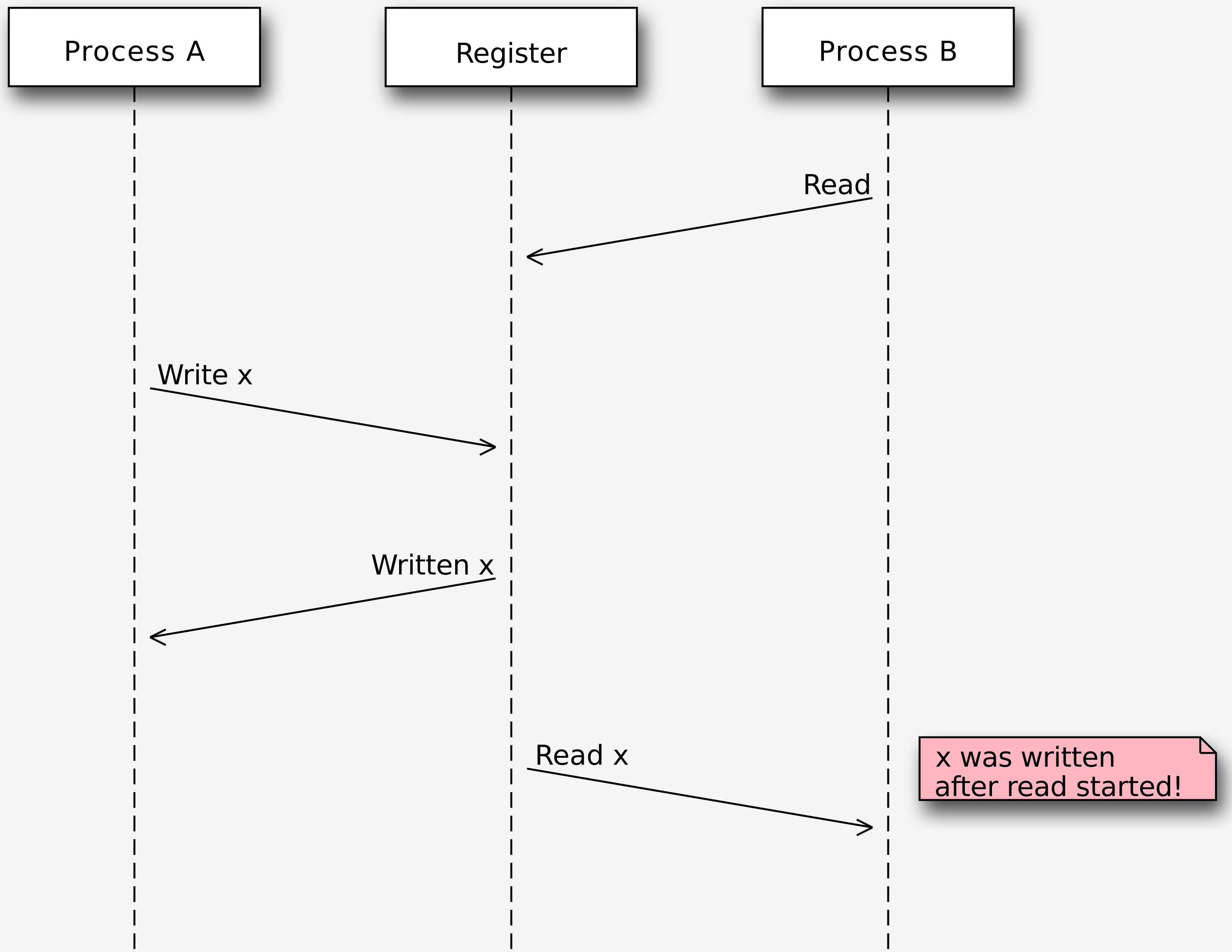
This is how we expect stuff to work

We are a spoiled bunch

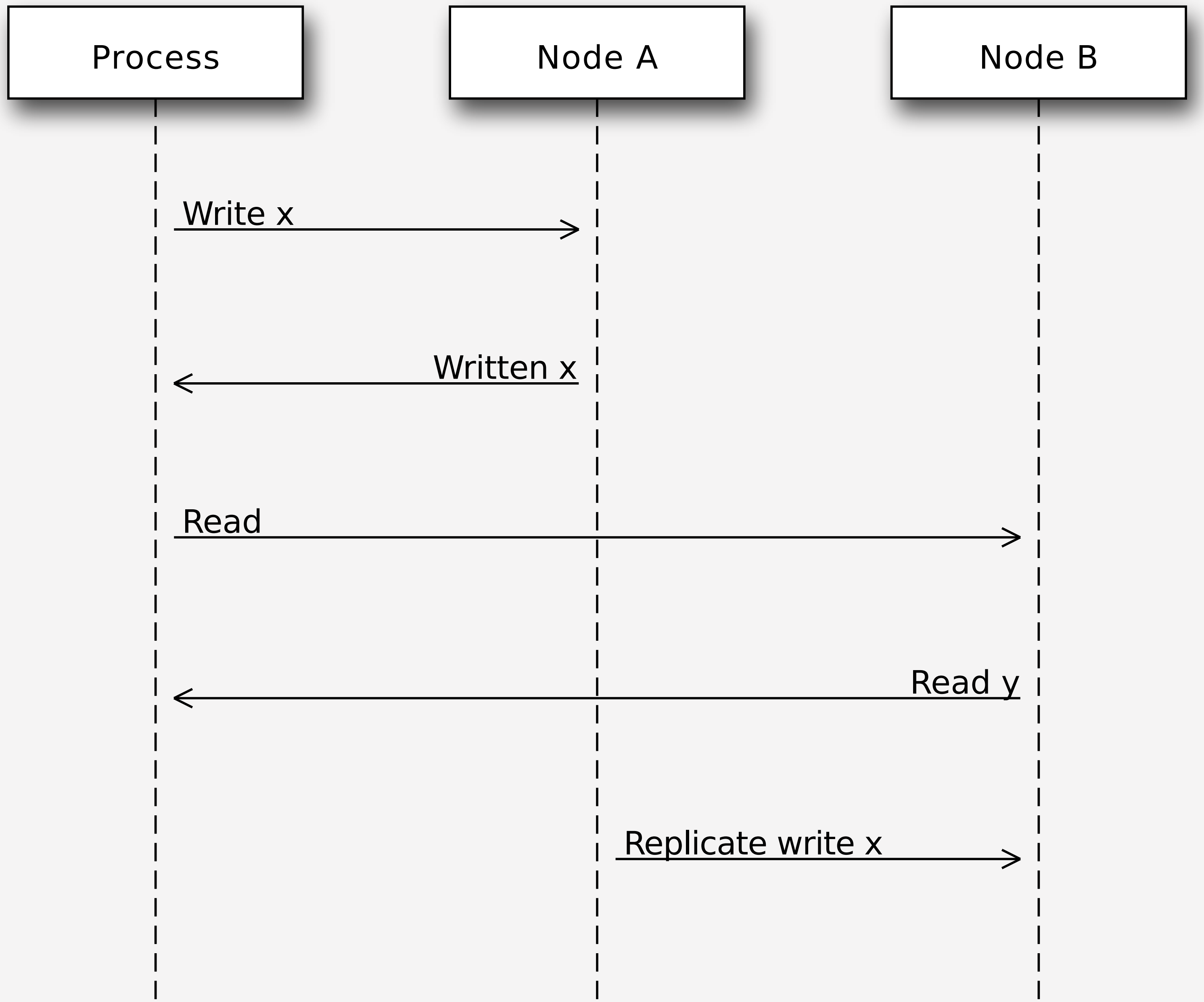
Information travels at *c*



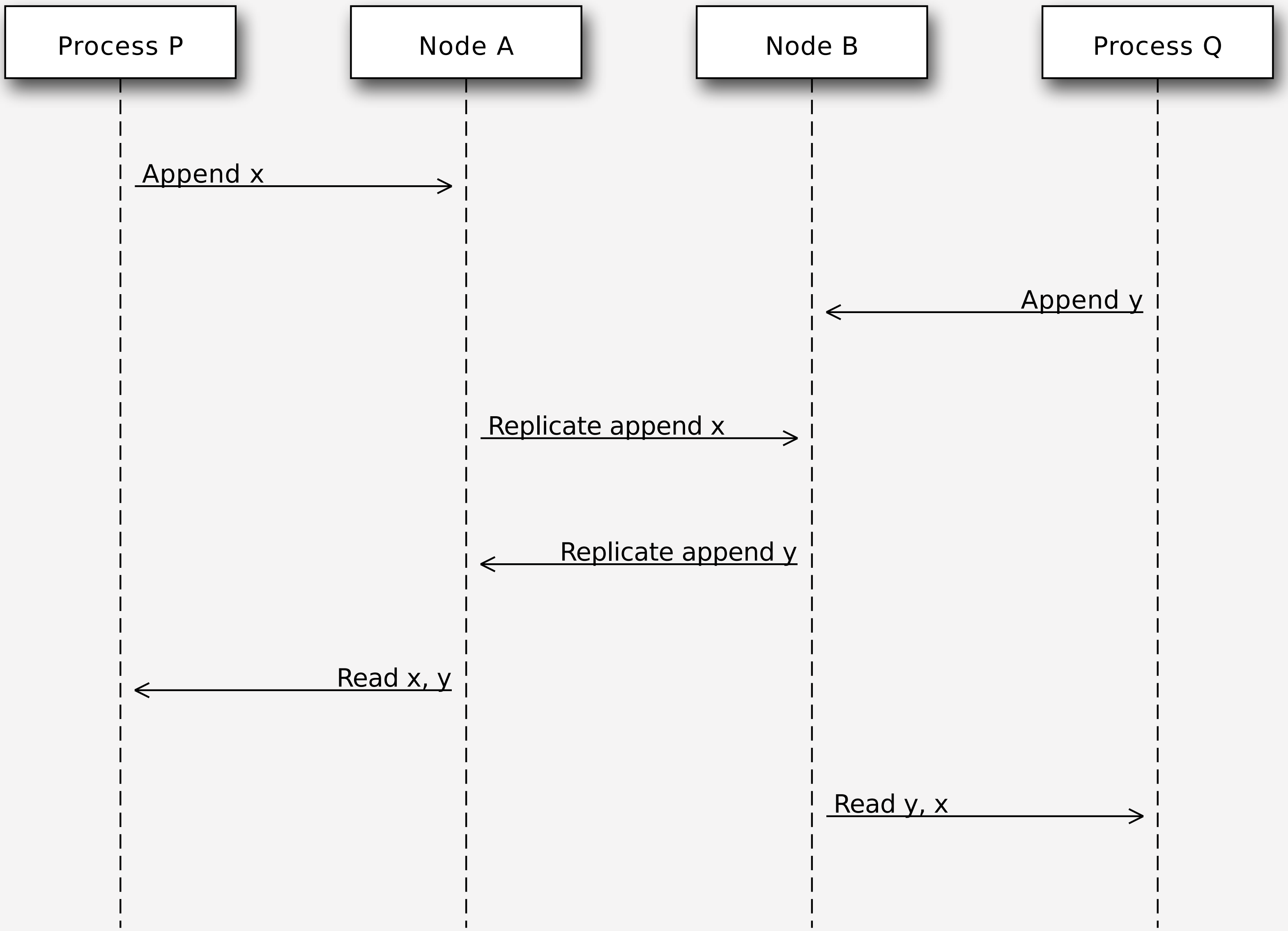
Slow stuff can overlap



Writes don't replicate instantly


Process
Node A
Node B
Write x
Written x
Read
Read y
Replicate write x

Writes can get reordered



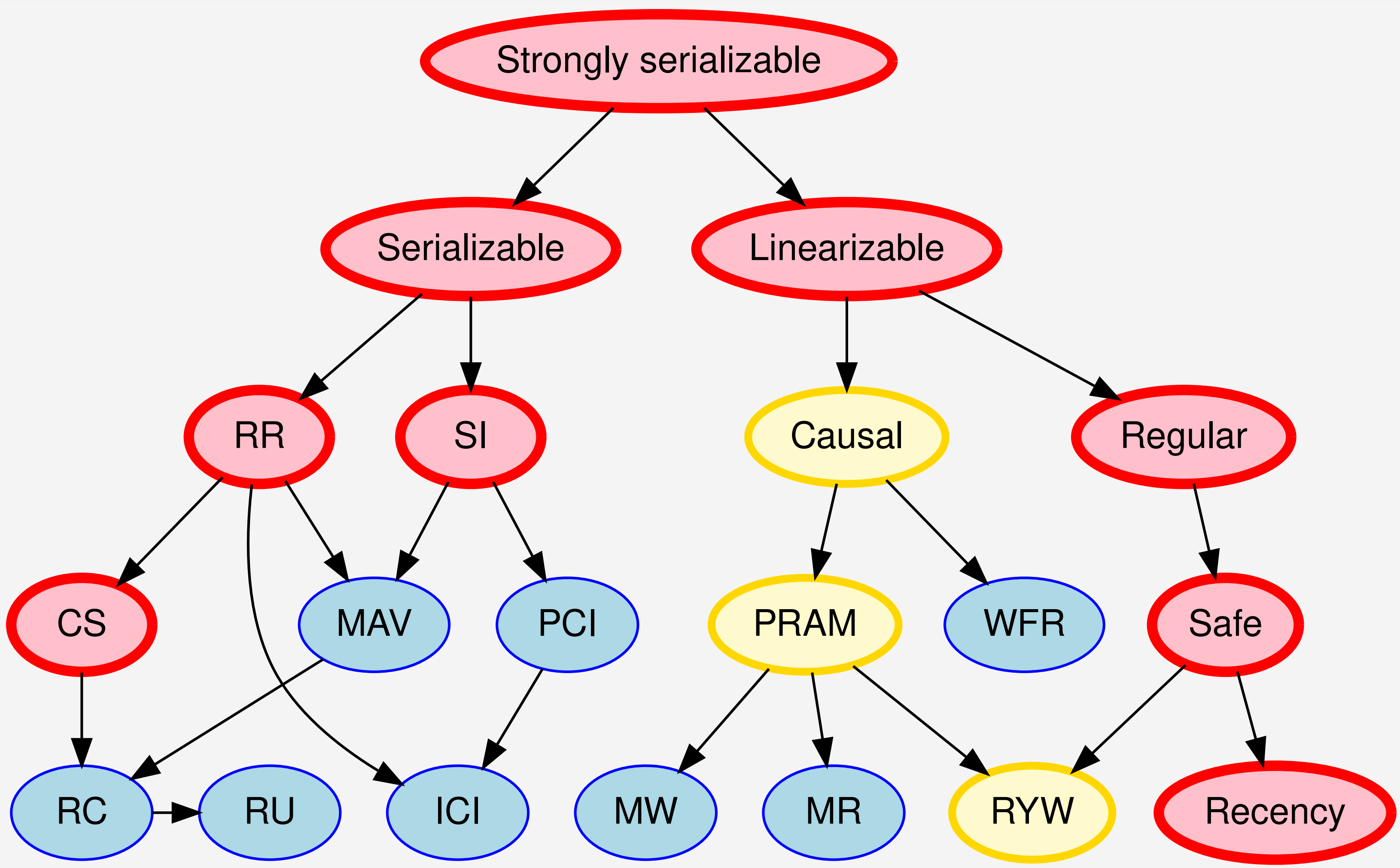
All sorts of stuff can happen

- Multiple registers
- More semantics
- More nodes
- More failure modes

Reasoning about the system

- What can and can't happen?
- What *can* happen: consistency model

Theoretical consistency models



Serializability

- ∃ serial execution with the same result
- *Some* serial execution: fairly weak
- No restrictions on which one

Example: serializability being weak

Precondition: $x = 0$

1. $x \leftarrow 0$
2. $x \leftarrow 1$
3. $x \leftarrow 2$

Example: serializability being strong

Precondition: $x = y = 0$

1. $y \leftarrow 2$, assuming $y = 1$
2. $x \leftarrow 1$, assuming $x = 0$
3. $y \leftarrow x$, assuming $x = 1$

Linearizability

All operations appear to happen instantly

Strong serializability

Linearizable & serializable

Your computer is distributed

Models in "centralized" systems

- SQL databases
- Clojure reftypes

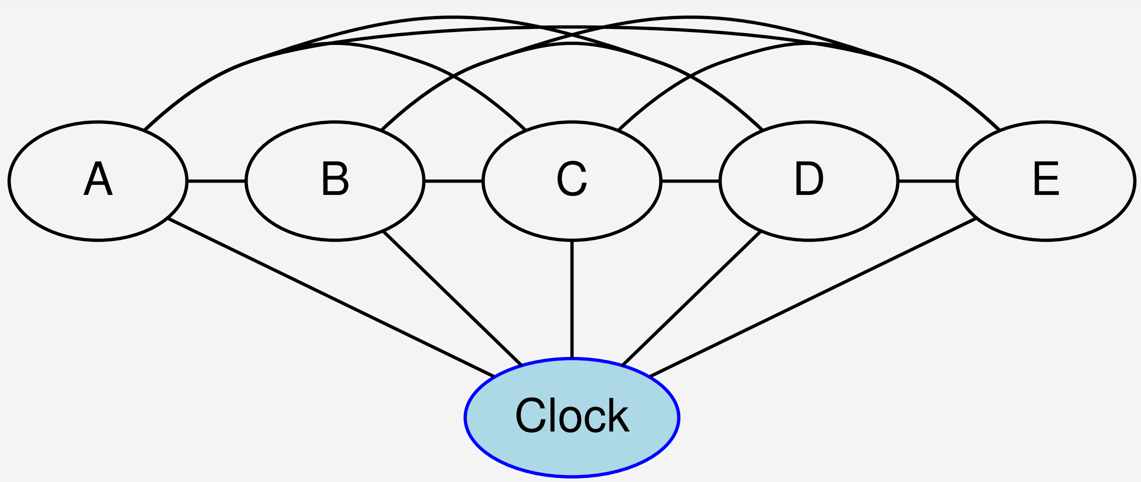
Twisted vs threads

In terms of concurrency models

- Twisted: strongly serializable
  - Event loop with 1 reactor thread
  - Serializable: reactor finds the ordering
  - Linearizable: callbacks run by themselves
- Threads: no defined model
  - Unladen Swallow tried to figure it out
  - Nothing fancy; whatever your CPU gives you
  - Probably okay (heap + GIL)
  - Correct use of locks?

Time

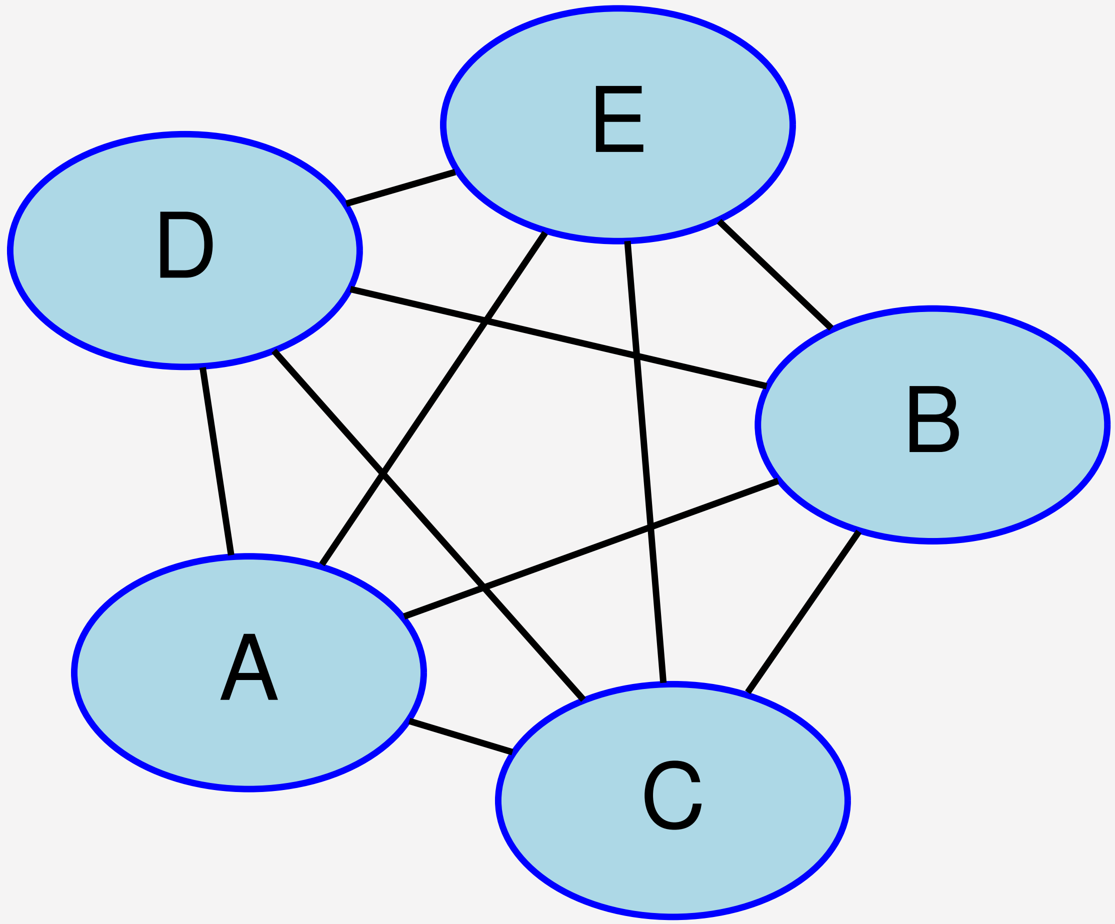
Global clock model



Global clock

- Everyone sees the same clock
- Access instant, uncertainty 0
- Can compare different timestamps
- Mental model: wallclock

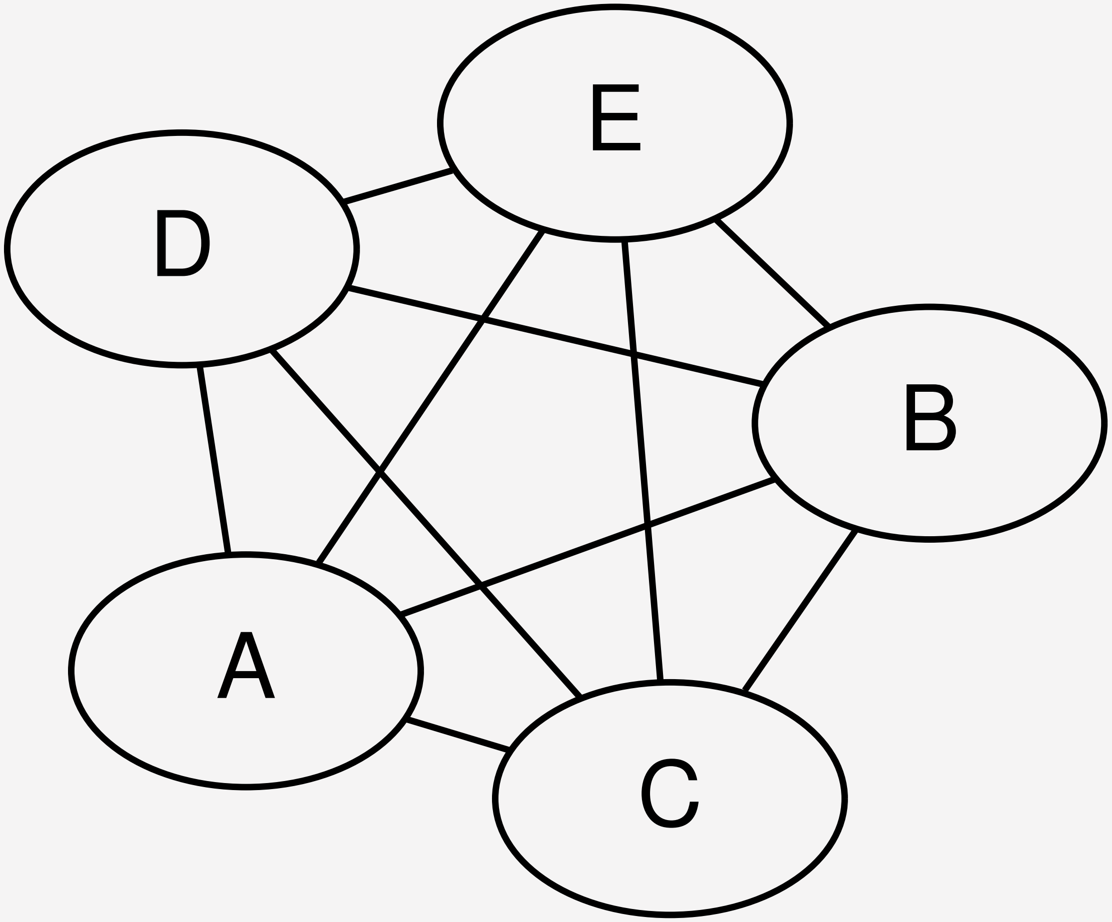
Local clock model



Local clocks

- Each clock is kinda reliable
- Can't compare with other timestamps
- Mental model: stopwatch

No clock model



Can't have a global clock

- Can pretend they *almost* exist
- Going to be wrong often

Example: Google Spanner

- GPS & atomic clocks
- "Atomic clocks [...] drift significantly"
- "uncertainty [...] generally <10ms"

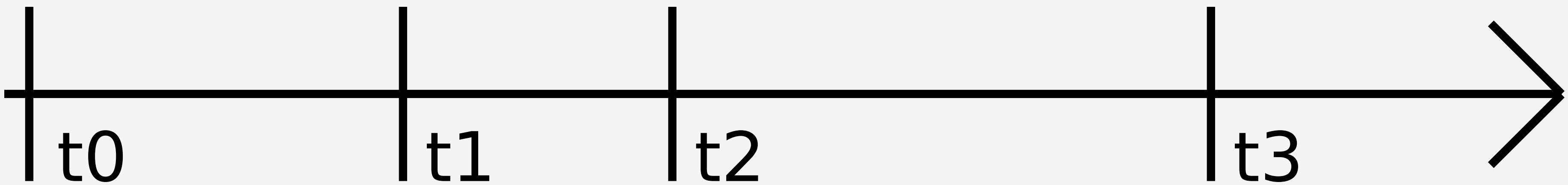
Can't have *no* clock

- Need time for failure detection
- FLP result says nothing works

Timestamps are often a proxy

- *Actually* care about progression, partial order
- Timestamps don't have to match real-world time

Example timeline



Sequential numbers vs real timestamps?

Lamport & vector clocks

Lamport clocks

(informally)

keep a version number of what you've seen

Vector clocks

(informally)

keep version numbers of what you've seen other nodes see

Good news

Stuff you *can* rely on

Queues

Consensus protocols

Getting computers to agree on things

Examples

- ZAB (Zookeeper)
- Paxos* (Chubby)
- Raft (`etcd`)

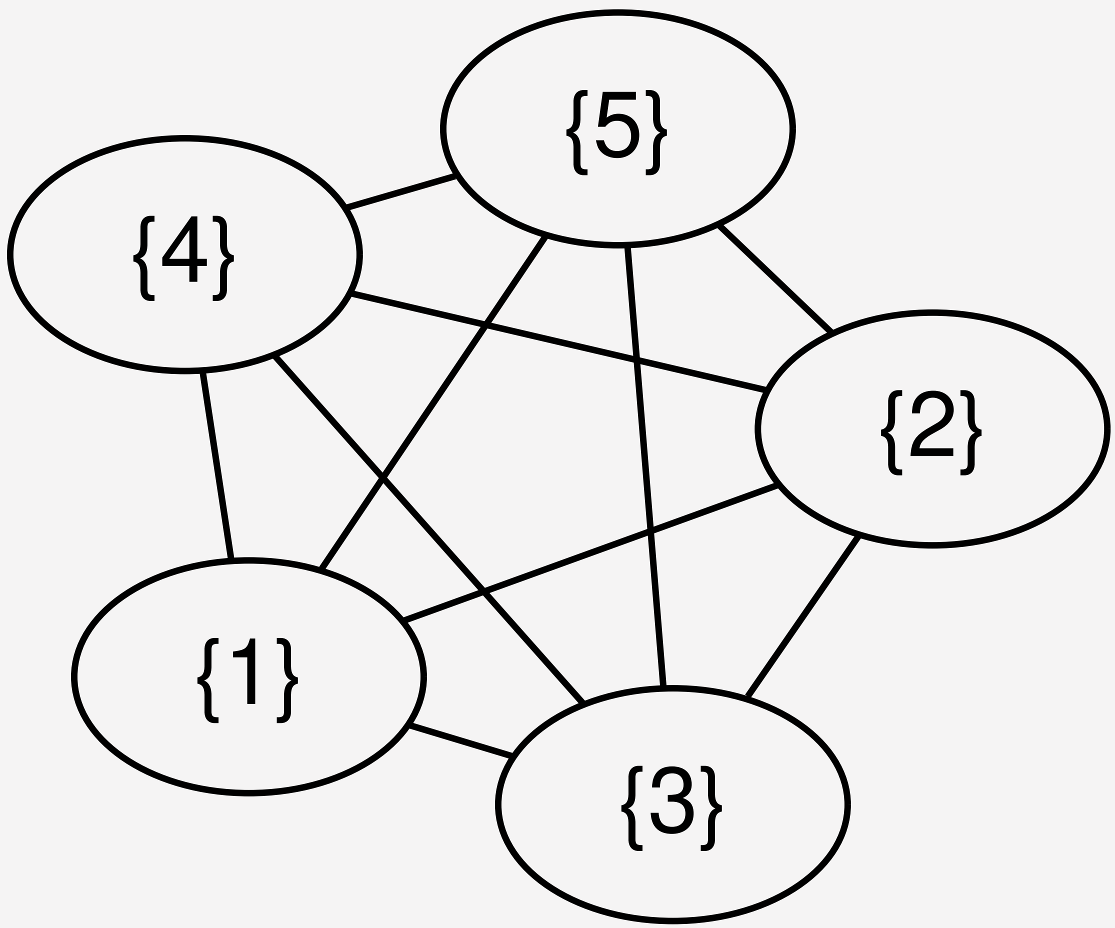
Recipes

On top of consensus protocols:

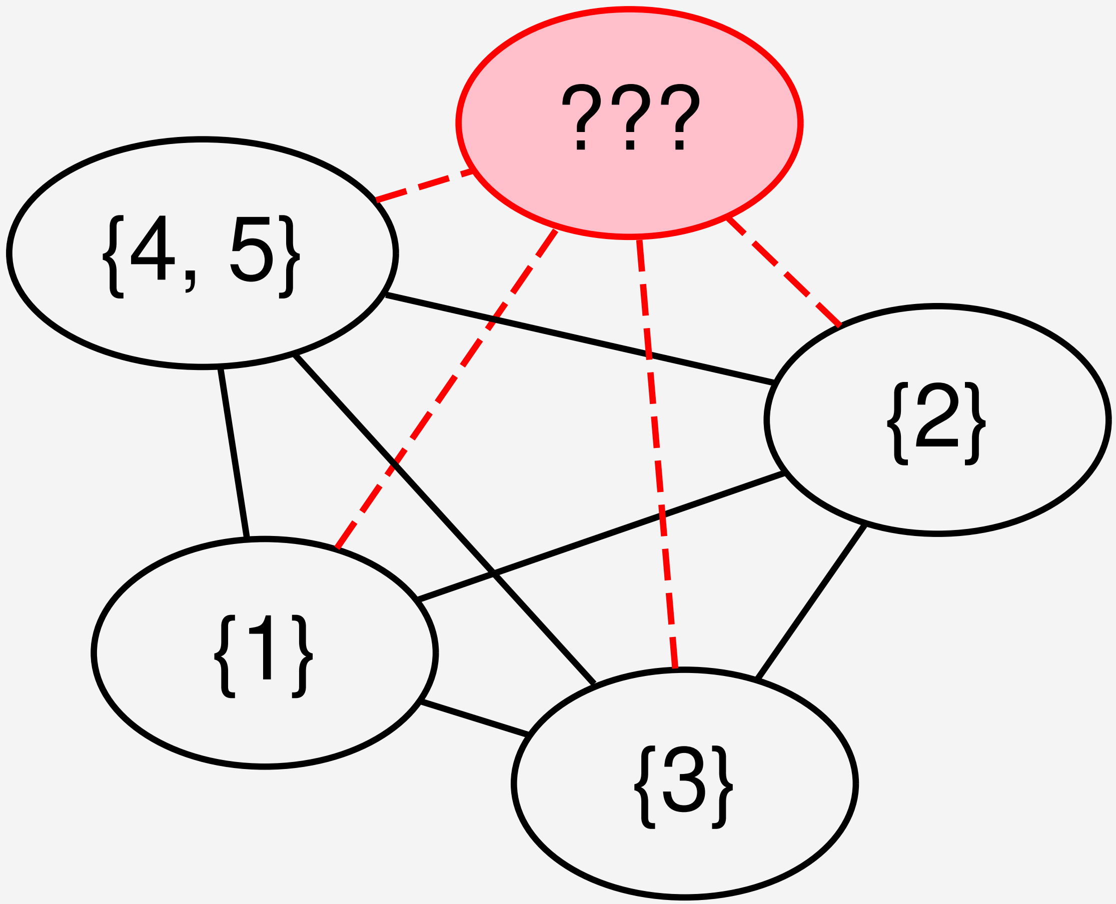
- Locks
- Barriers
- Set partitioning
- …

Set partitioning

{1, 2, 3, 4, 5}



Recovery from failure



CRDTs

Conflict-free replicated data type

Problem

- Read, compute, write back
- Concurrency: multiple results
- Conflicts!

Solutions?

- Last write wins? Most writes lose :-(
- Coordination? Expensive! :-(

I want highly available data stores...

.. but I don't want nonsense data

Idea!

- Describe what you want
- Describe conflict resolution

Specializations

The C in CRDT can mean:

- Commutative (CmRDT)
- Convergent (CvRDT)

Commutative RDTs

- Broadcast operations
- Merge operation:
  - Commutative: $f(x, y) = f(y, x)$
  - Associative: $f(f(x, y), z) = f(x, f(y, z))$
  - Not idempotent $f(x, y) != f(f(x, y), y)$

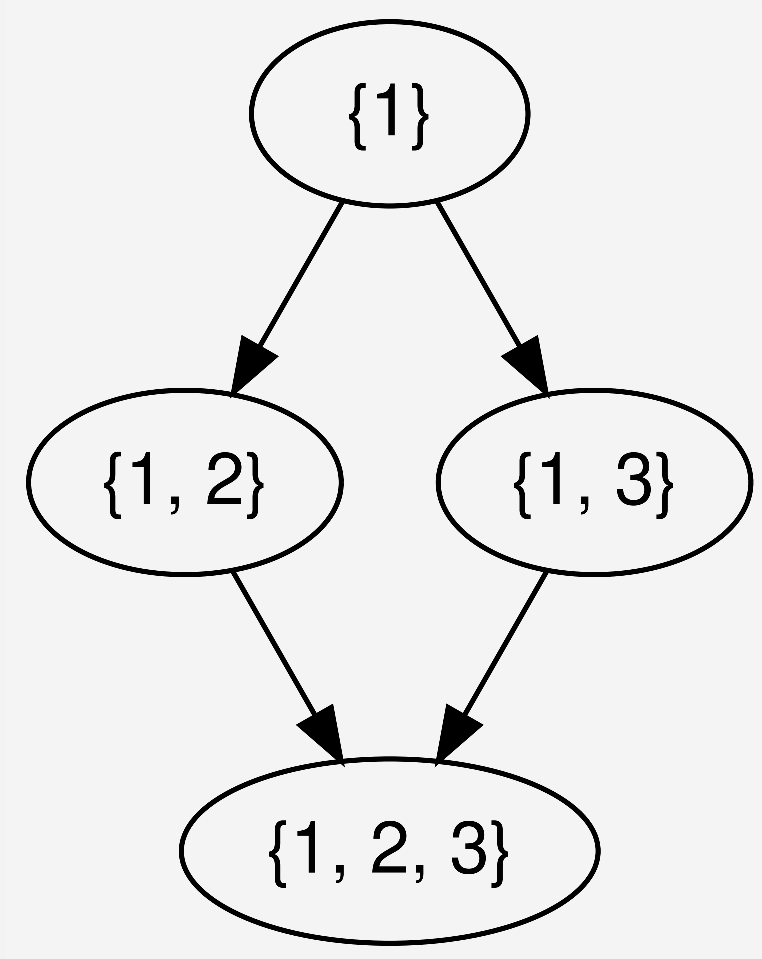
Example: integers

- +1, -2, +3, +5, -4: +3
- Always get same answer:
  - As long as I see all ops *once*
  - Duplicate an op, get wrong answer
  - Order doesn't matter, though

Convergent RDTs

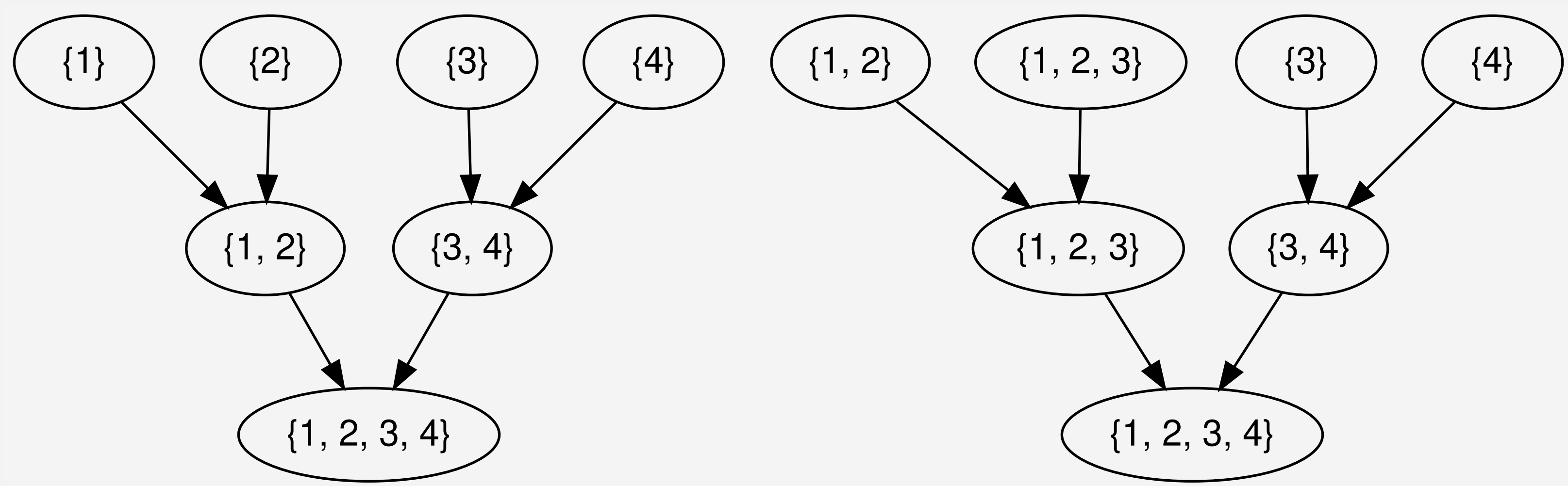
- Broadcast (sometimes partial) states
- Merge operation has many properties:
  - Commutative: $f(x, y) = f(y, x)$
  - Associative: $f(f(x, y), z) = f(x, f(y, z))$
  - Idempotent: $f(x, y) = f(f(x, y), y)$
  - Informally: apply lots until done

Simple CvRDT conflict resolution



Complex CvRDT conflict resolution

It's okay if you see writes more than once!



CRDTs in practice: usually CvRDT

Solve local problem once

vs

Solve distributed problem constantly

Examples

- Counters (G, PN)
- Sets (G, 2P, LWW, PN, OR)
- Maps (sets of *(k, v)* tuples)
- Graphs (using multiple sets)
- Registers (LWW, MV)
- Sequences (continuous, RGA)

Using CRDTs

- Designing them is tricky
- Using them is fairly easy

Riak <3

Flags, registers, counters, sets, maps

Wrap-up

Yay, distributed systems!

- More resilient
- More performant
- Make problems tractable

Argh, distributed systems!

- Incredibly hard to reason about
- Huge state space, no repeat scenarios
- Expensive to operate

Lots of distributed systems

- Everything is about tradeoffs
- Figure out what's right for your app
- Don't build what's on the shelf

Why distributed systems?

Because you're *out of options*.

Thank you!

Slides

`www.lvh.io/DistributedSystems101`