
Dealing with Distribution

Pros and Cons of Microservices

- In this session we'll explore the pros and cons of building a distributed system (i.e. microservices)
  - Pros: Agility, independent delivery, technology choice, scale, coolness, etc
  - Cons: Additional testing, stronger governance, anarchy, finger pointing, difficult failure modes, etc
- Most importantly microservices are a distributed system
- Distributed systems are fun, and challenging, but…

"Distributed systems are easy!"

-- No-one of a sane mind, ever

So why would we adopt microservices?

- Agility
- Delivery flexibility
- Change management
- Technology choice and independence
- Scale

So why not?

- Governance
- Delivery challenges and dependencies
- Fragility and risk
- Unnecessary for departmental scale (80% of systems)
- Computer science (hurts my brain)

We can conquer these issues...

... And reap some benefits...

... but there's a cost and no room for shortcuts


Distributed Systems

Deutsch's Fallacies of Distributed Computing (circa 1991)

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous.

Picking Apart Those Fallacies

- The network is reliable
  - Within a data centre, the network is very good
  - Failures will happen, but they will happen at seconds per month, not per hours
  - Failures are usually transient
  - WANs are more problematic
    - Transient does not apply to a submarine cutting your undersea cable!

Picking Apart Those Fallacies

- Latency is zero
  - Latency in a modern DC is very low
  - It is cheaper to access someone else's RAM across a backplane than to access your own disk
  - But API access will usually have far higher latency than local method call
  - This is critical for microservices
    - Unless you're going down the RDMA path
    - Which you're not, because coupling

Picking Apart Those Fallacies

- Bandwidth is infinite
  - 10 Gbit/sec is commodity
  - 100 Gbit/sec available
  - Infiniband is astonishing 300 Gbit/sec
  - That's a less than inter-core bandwidth
    - 80 G**Byte**/sec
  - But it's huge and now same order of magnitude as core cache connectivity

Picking Apart Those Fallacies

- The network is secure
  - It's not, especially when you don't own it
  - Oh, even if you do own it, it's probably compromised somewhere
  - So you have to work around it by making service-service interactions secure
    - And within services by securing database connections etc.
  - This increases complexity and reduces throughput

Picking Apart Those Fallacies

- Topology doesn't change
  - It will because services you depend on will be run by less competent people
  - Those services will go down
  - You shouldn’t forget to protect yourself
    - Circuit breaker pattern
      - Don’t further overload a struggling service
      - I’ve used HTTP 413 for this
  - Graceful degradation of service preferable to cascading failure

Picking Apart Those Fallacies

- There is one administrator
  - There will be at least one administrator per service
  - But your NOC will also be providing air cover
    - So the networking things are covered
  - What this really means is that you can't expect anything other than an API to bind to
    - And you need to shape that API with CDCs
  - The services you depend on will change at their administrator's whim

Picking Apart Those Fallacies

- Transport cost is zero
  - Both bandwidth and latency of network are **amazing**
  - But the cost of marshalling and unmarshalling all those application-level messages is considerable
    - Unless you go with efficient packaging mechanisms like Aeron
  - So the cost of a remote interaction is overall quite high

Picking Apart Those Fallacies

- The network is homogeneous
  - Depending on your topology, it might be
  - A single backplane is fast, high-throughput
  - A multi-DC system, a little less so

So what to do

- We've just covered off Deutsch's fallacies and found most of them still apply
- Microservices are distributed systems and distributed systems are hard
- Particularly around failures
- So let's just get back to a monolith and let the database deal with the problems, right?
  - Fortunately not
  - We don't have to deal with all the problems in distributed systems, just some of them

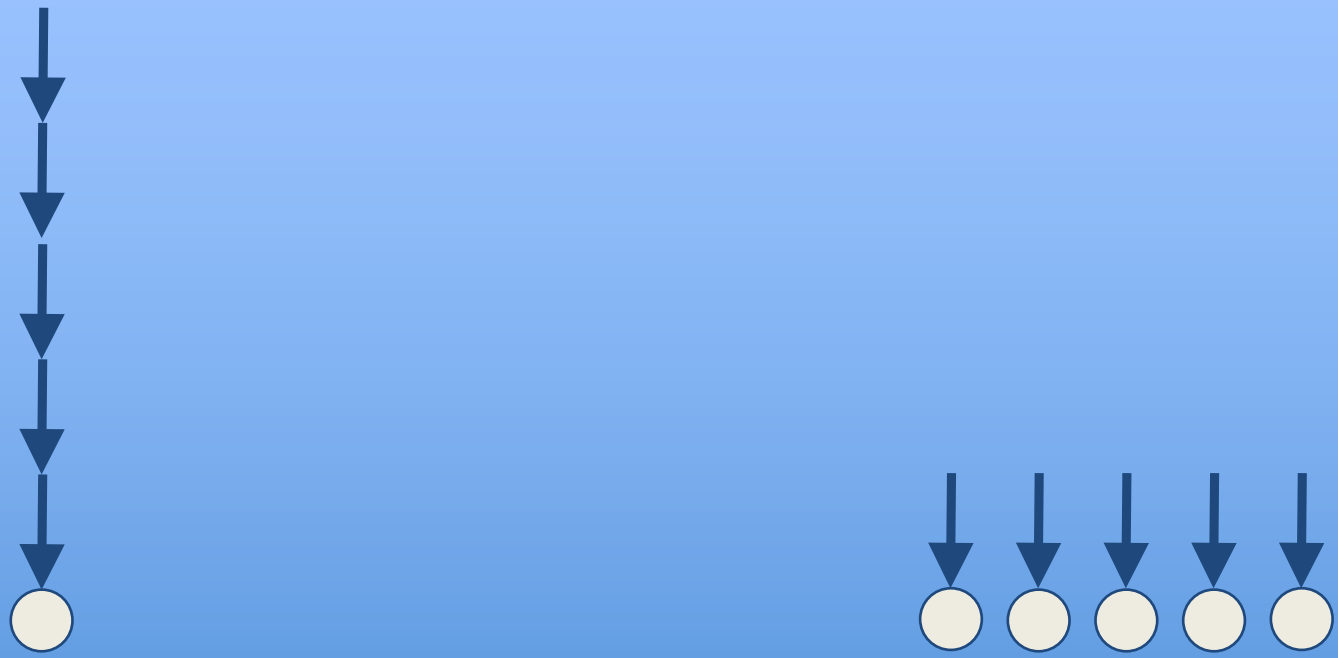
So this seems hard, why would we do it?

- Aggregate throughput
- Redundancy for fault tolerance
- High available for continuous service

- Need these both within a microservice and between mircorservices

- Let’s see why…

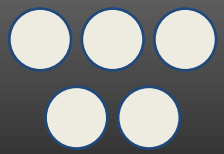
Massive Throughput



Data Redundancy

Data Redundancy

Data Redundancy



Data Redundancy

High Availability

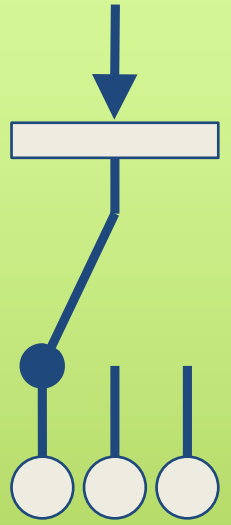
High Availability

High Availability

Error!
503: Service
Unavailable

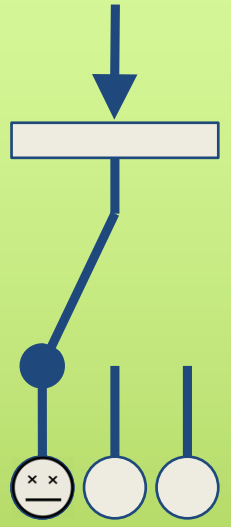
High Availability

Error!
503: Service
Unavailable



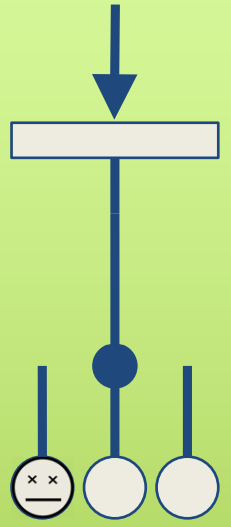
High Availability

Error!
503: Service
Unavailable



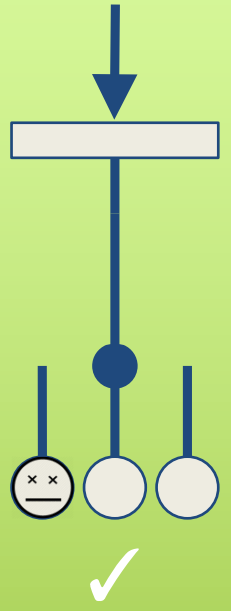
High Availability

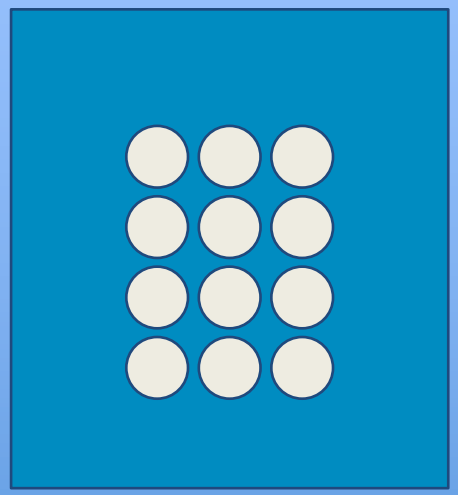
Error!
503: Service
Unavailable



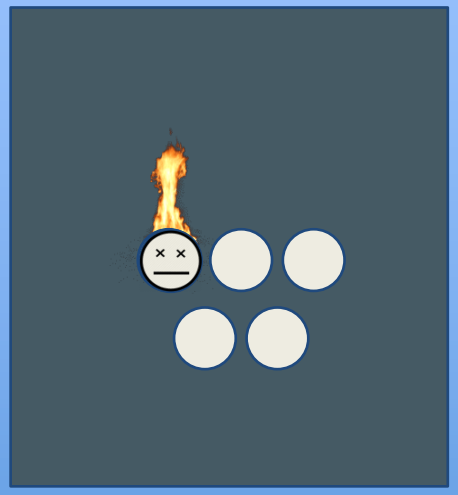
High Availability

Error!
503: Service
Unavailable

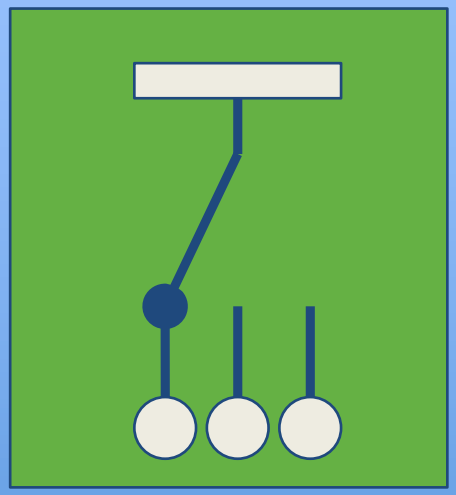




Massive Throughput



Data Redundancy



High Availability

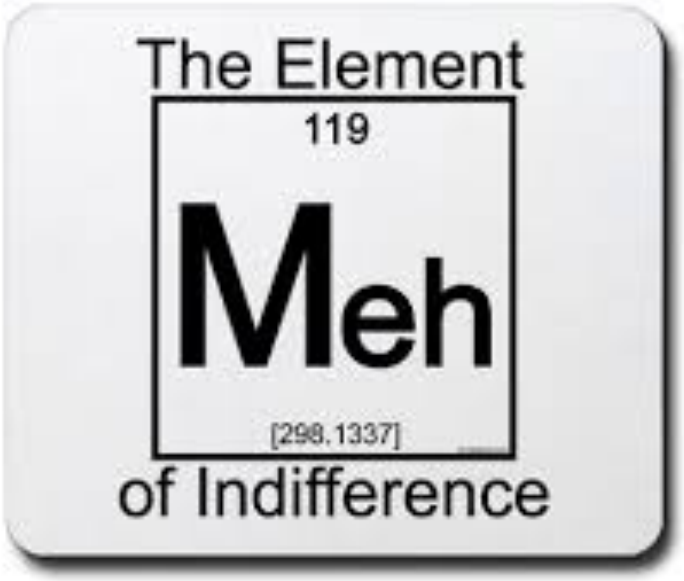

Transactions and Concensus

Single Server ACID Transactions

- ACID transactions are ***reliability*** protocol
- Within a single service they need:
  - 2 phase locking, a coordinator, to serialise any contended access to resources
- To impose ordering, safety, ACID is conservative and mechanically costly (e.g. disk flush)
- Which is why we're seeing a lot of other consistency models (e.g. eventual consistency, very weak guarantee)

ACID Transactions across service boundaries are a DOS attack

- ACID transactions become unavailability protocol
  - I lock your resources, never unlock
  - Rely on a coordinator service
    - Which I probably don't own
  - Coordinator failure compounds unavailability for multiple services



Workflows and Compensations

- Compensating actions usually deployed
- Build in order/cancel reciprocal actions into the Domain Application Protocol
- Use hypermedia to stitch these together over the network
- But how do you un-fire the missiles?
  - There may not be a perfect compensation

ACID Semantics, Not ACID Drawbacks

- ACID implementations have suffered many drawbacks
  - Blocking, contended, SPOF
- But the semantics are very appealing
  - Easy to reason about operations in a system
- Consensus protocols are not dead
  - Paxos, raft are non-blocking
  - Multi-Paxos, Multi-Raft scale well if you domain partitions easily
- Transactions redux
  - Avoid coordination, then even ACID transactions are fast

OK, so not so “Restful” but…

Apache Kafka

A high-throughput distributed messaging system.

Apache Kafka is publish-subscribe messaging rethought as a distributed commit log.

Distributed Commit Log?

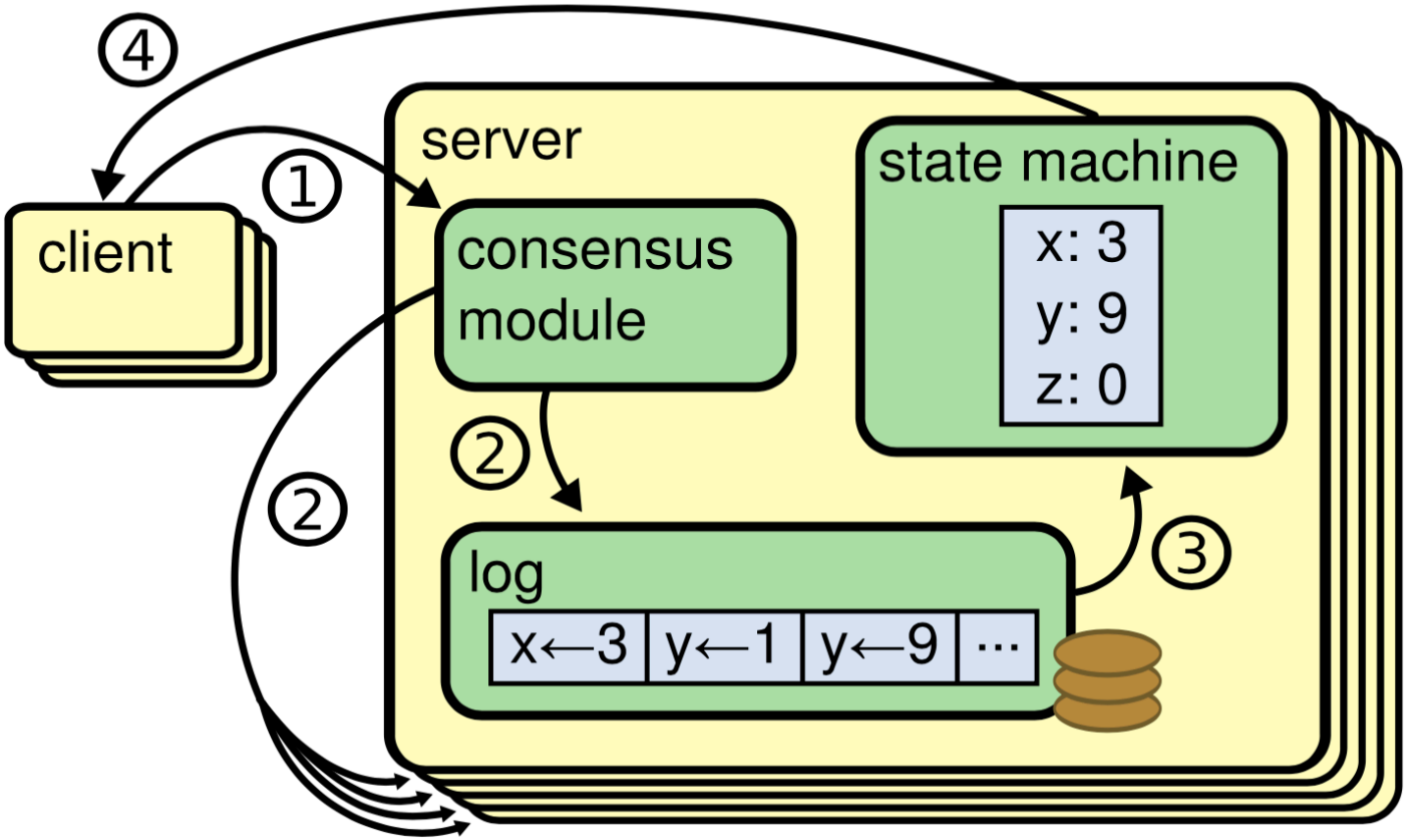
- This is deep distributed systems stuff being surfaced to the application layer as integration middleware
- It could be a viable alternative to event feeds for some scenarios (latency?)
- So what is it anyway?

Raft: A Humane Protocol for Distributed Commit Logs

- Diego Ongarro, PhD Stanford 2014
- Similar in intent to Paxos
  - But actually understandable by mere humans
  - This is important: difficult protocols have bugs
  - Bugs are multiplied when software is distributed
- As it happens, we use Raft in Neo4j
  - And historically used Paxos too

- Bonus question: why call it Raft?

Distributed Consensus



In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

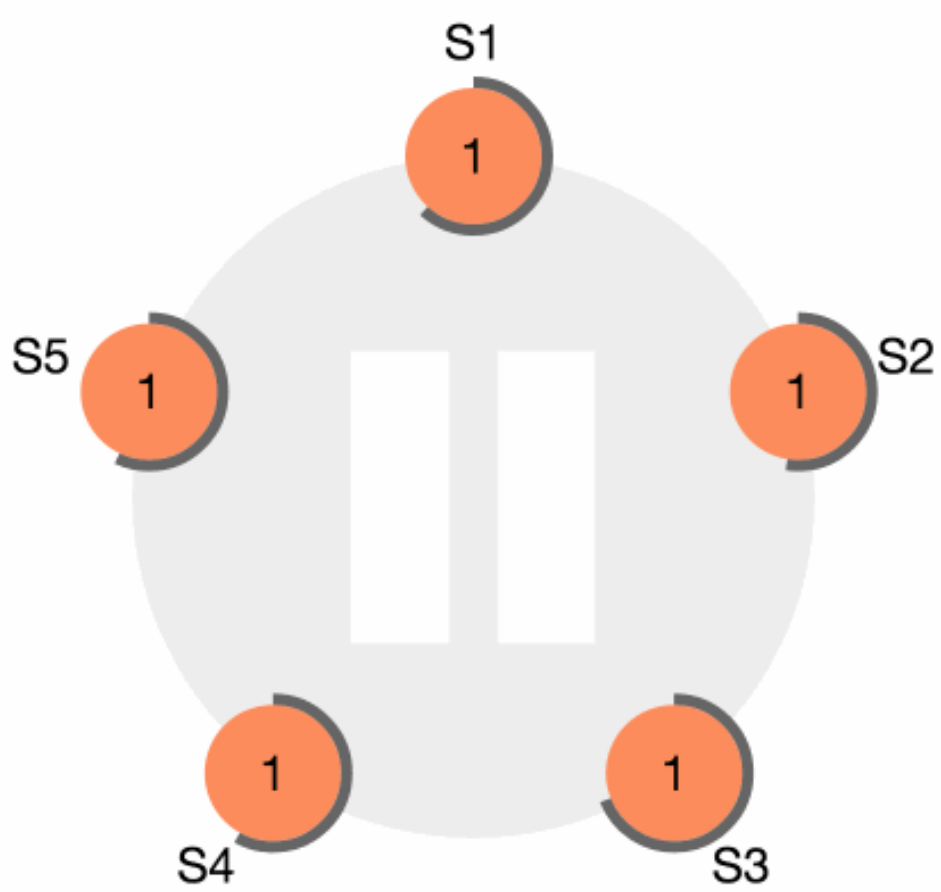
state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to

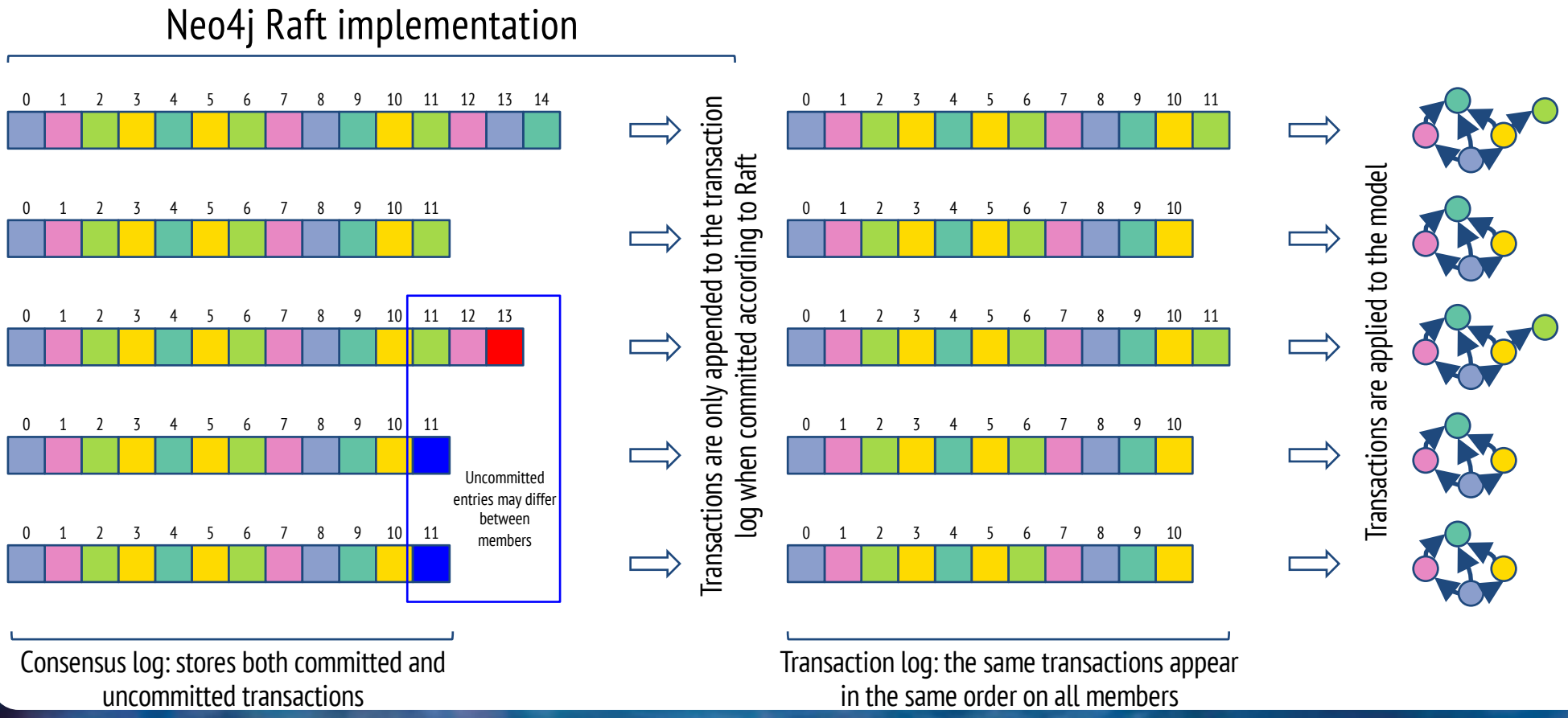
EASY TO UNDERSTAND

Raft Protocol



https://github.com/ongardie/raftscope

Consensus Log → Committed Transactions → Updated Model



Roles

- Leader: Handles client requests and drives replication.
- Followers: Responds to requests from the leader.
- Candidates: A temporary role assumed by a follower that is trying to become the leader.

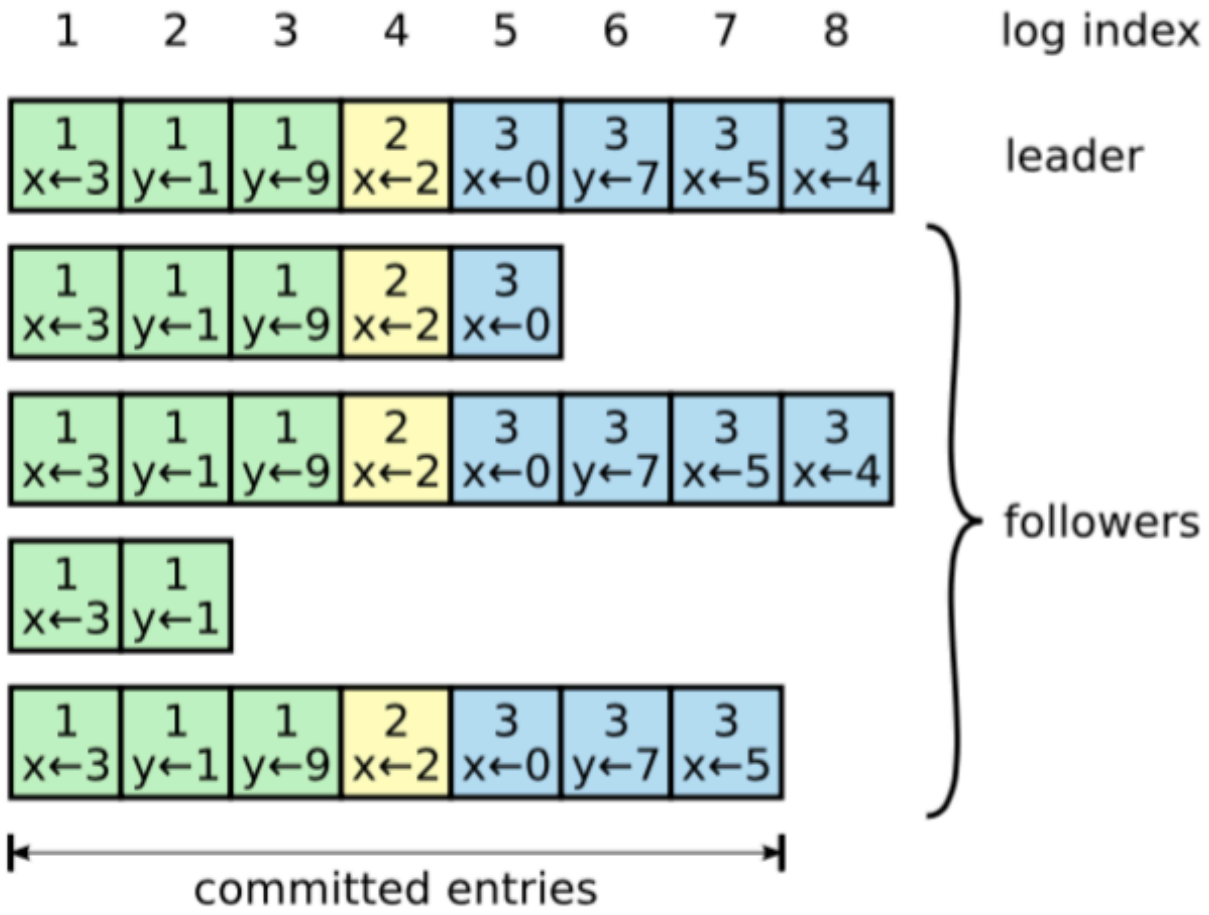
Messaging - General

- RAFT designed to work in a distributed system
  - messages can get lost, be duplicated, arrive out of order, be severely delayed, etc.
  - not designed for byzantine or other failures like malformed messages
- Small set of core messages
  - AppendEntries Request/Response
  - Vote Request/Response
  - Client Request

Appended, Committed, Applied

- Entries are appended to the end of the log of all members
  - Appended entries might be truncated
- When the leader has observed a successful append of an entry by a quorum of the members (N/2 followers) then it now considers that entry safely committed
- A committed entry can be safely applied
  - Cannot be truncated by current or future leaders

System view of logs



Properties

Election Safety

At most one leader can be elected in a given term. §3.4

Leader Append-Only

A leader never overwrites or deletes entries in its log; it only appends new entries. §3.5

Log Matching

If two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §3.5

Leader Completeness

If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §3.6

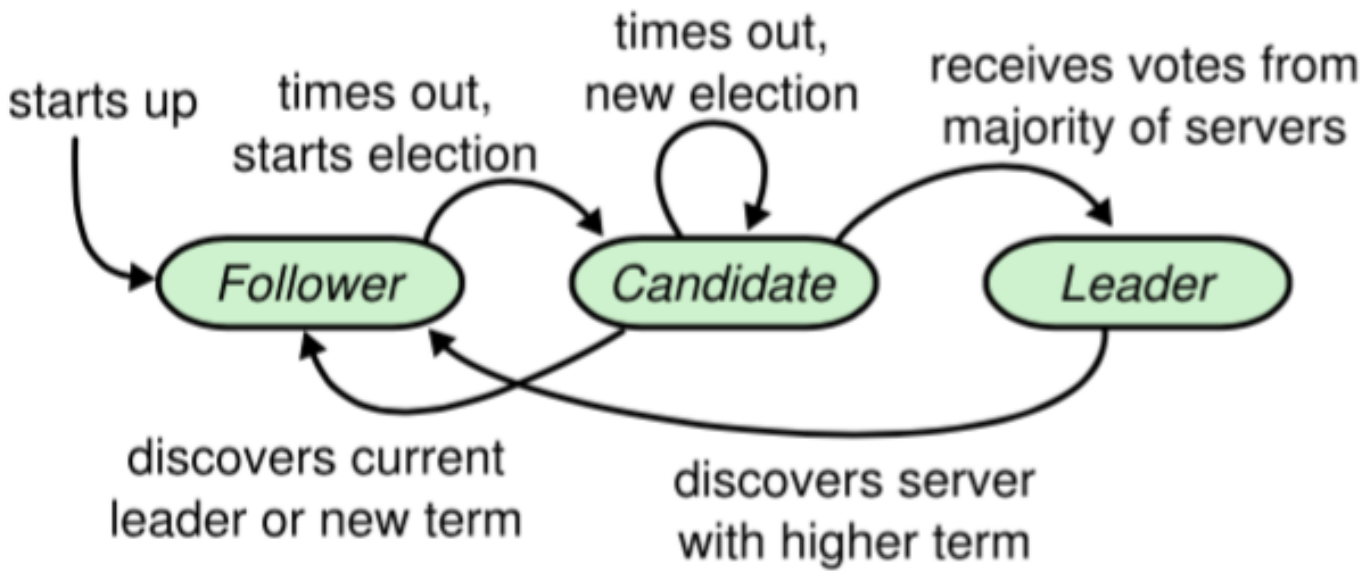
State Machine Safety

If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §3.6.3

Elections

- Every member starts out as a follower
- If a member does not hear from a leader within an election timeout then it assumes the candidate role, to try and become the new leader
- A candidate sends out VoteRequests to all other members
- The VoteRequest is sent out in a new term
- Followers respond with yes/no depending on if they considered this candidate eligible for becoming a leader
- If the candidate gets a sufficient amount of yes-votes (N/2), then it assumes the leader role in that term

Role switching



Leader eligibility

- Followers only vote for a single candidate in each term.
- Follower will not vote for candidates in older terms.
- Followers only vote for a candidate with a log which is at least as up-to-date as its own. Specifically in order:
  - Ending with the largest term
  - Ending with the largest index

Heartbeats

- The AppendEntries message is also used as the heartbeat message
  - Empty AppendEntries sent if no work to be done
  - Election timeout/2 is a sane frequency
- No heartbeats triggers a candidate to emerge
  - Not a big deal, but we'd like to do real work rather than cluster admin
- Election happens, leader may change

Split votes

- Elections are susceptible to split votes that must be resolved.
- Split votes are situations where none of the candidates become a leader
- After much consideration and tries, a simple approach using random timing was deemed the best
  - The election timeout for followers will have an amount of randomness, e.g. 150-300 ms.
- Retry until leader emerges

Membership changes

- Must avoid all situations where two leaders could co-exist in the same term
- Should retain availability
- Membership itself is just stored as another entry in the log
  - Therefore inherits the Raft safety property!
- The latest membership entry appended to the log is used
- A new membership change should not be commenced until the previous one is committed

Thesis is huge

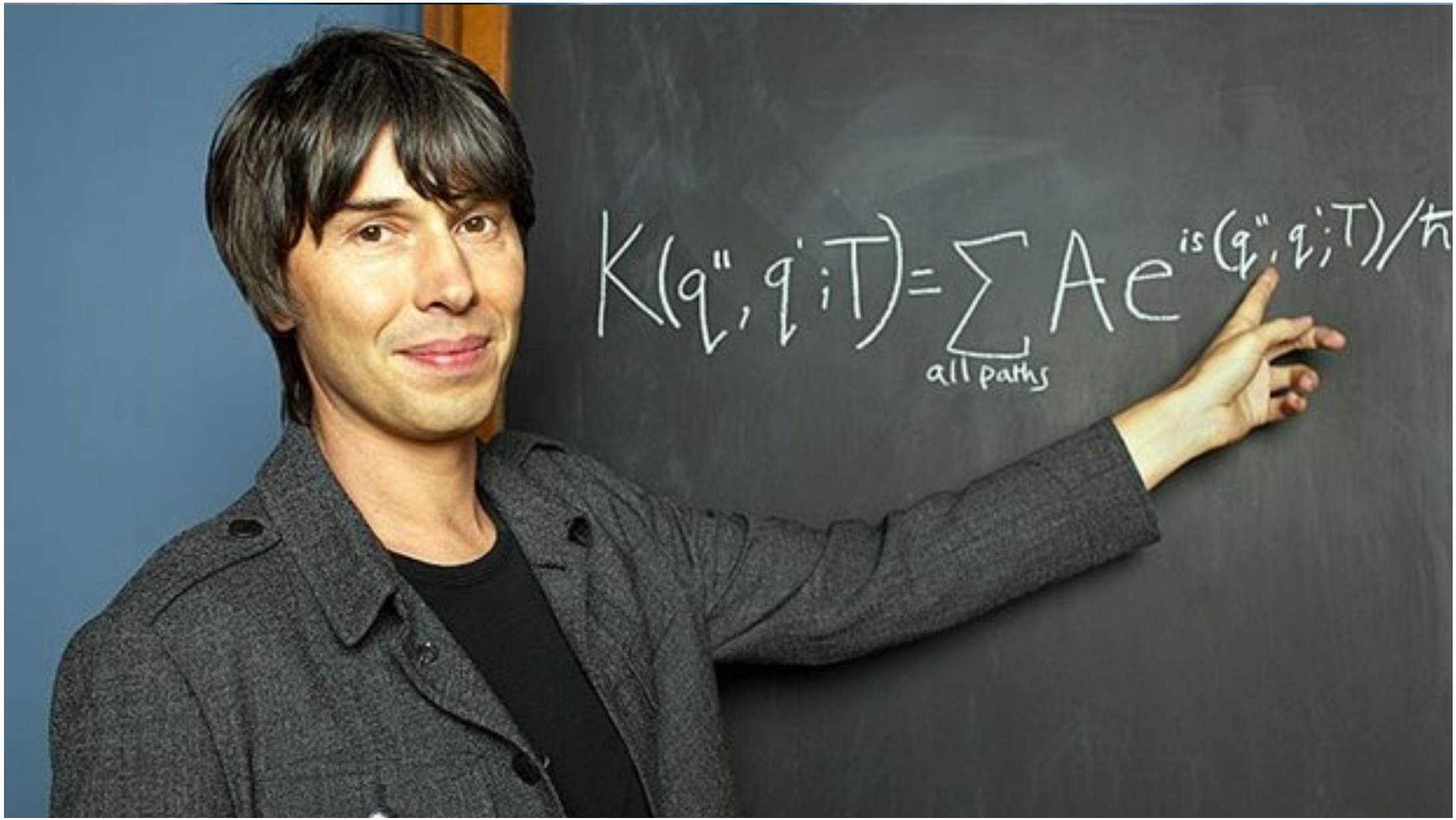
- 257 pages
- Examples of what was omitted
  - Lots of details, for example how to achieve performance
  - Formality
  - Proof of algorithms
  - Studies of it being more understandable
  - …

Raft is great but…

- We still have a contention for the log head
- We have only a single leader per Raft instance
  - Can use multi-raft if systems are sharded

- But these can be overcome, and will be overcome in middleware that we'll use in the next 10 years

- See:
  - HA Transactions (Bailis et al)
  - Linear Transactions (Sirer et al)


K(q'', q'; T) = ∑ A e^{is(q'', q'; T)/ħ}
all paths

Exercise: “But REST can’t do Transactions”

You're working in a bank...

Your customers:

Each can be individually named, e.g.

```
/customers/1
/customers/2
...
/customers/197563
```

Challenge:

For **all** the high value customers only, give them a platinum card.

Constraints:

You cannot stop the world: other business activity will continue while you're giving this perk.

Rich people gossip. If one customer gets the card but not another, the cardless customer will find out and will be annoyed enough to leave the bank.

Losing a customer = losing your job!

Summary

- We are building distributed systems which is not easy
- Naïve programming will fail and failures are vastly more unpleasant when distributed
- Transactions are dead, long live transactions
- Computer Science will impact on us, we should have a reasonable awareness of it