Beyond Off-the-Shelf Consensus

Dr. Rebecca Bilbro  January 2021

Question #1

Think about the app you're currently working on.

How much consistency does it require?

01 Eventual consistency is good enough for us.

02 We require strong consistency.

03 I have no idea 😳

Question #2

Think about the app you're currently working on.

How concerned are you about compliance with CCPA, GDPR, LGPD, etc. ?

01 We added that cookie banner; wasn't that enough?

02 It's on our roadmap, but we haven't tackled it yet.

03 We have total visibility and control over the region(s) where user data is stored and replicated.

Question #3

Think about the app you're currently working on.

How well does your app support i18n/l10n?

01 We track geographic deployments and data replication to guarantee consistent UX around the world.

02 We haven't started thinking about global markets yet.

03 I think our frontend uses gettext and CLDR.

Table of contents

Dr. Rebecca Bilbro

- Founder & CTO, Rotational Labs, LLC
- Adjunct Faculty, Georgetown University
- Applied Text Analysis with Python, O'Reilly
- Creator & Maintainer, Scikit- Yellowbrick

@rebeccabilbro

What if data systems were a little *smarter*?

rotational.io

01

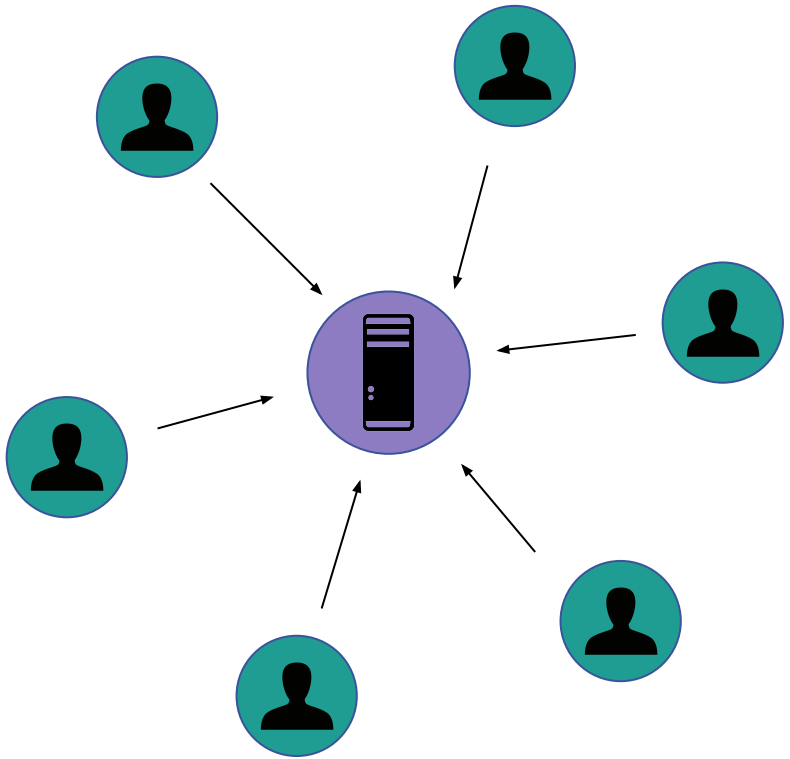
What is Consensus?

Clocks, quorums, and distracted parliamentarians

Context: A Single Server

Picture a simple data storage system where clients can:

- `GET(key) → value`
- `PUT(key, value)`
- `DEL(key)`

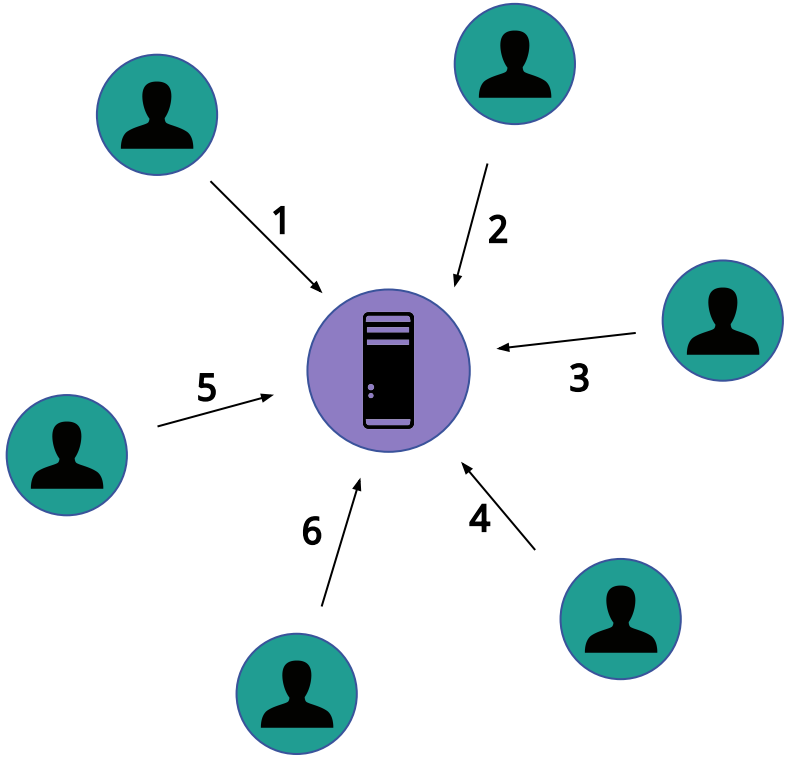


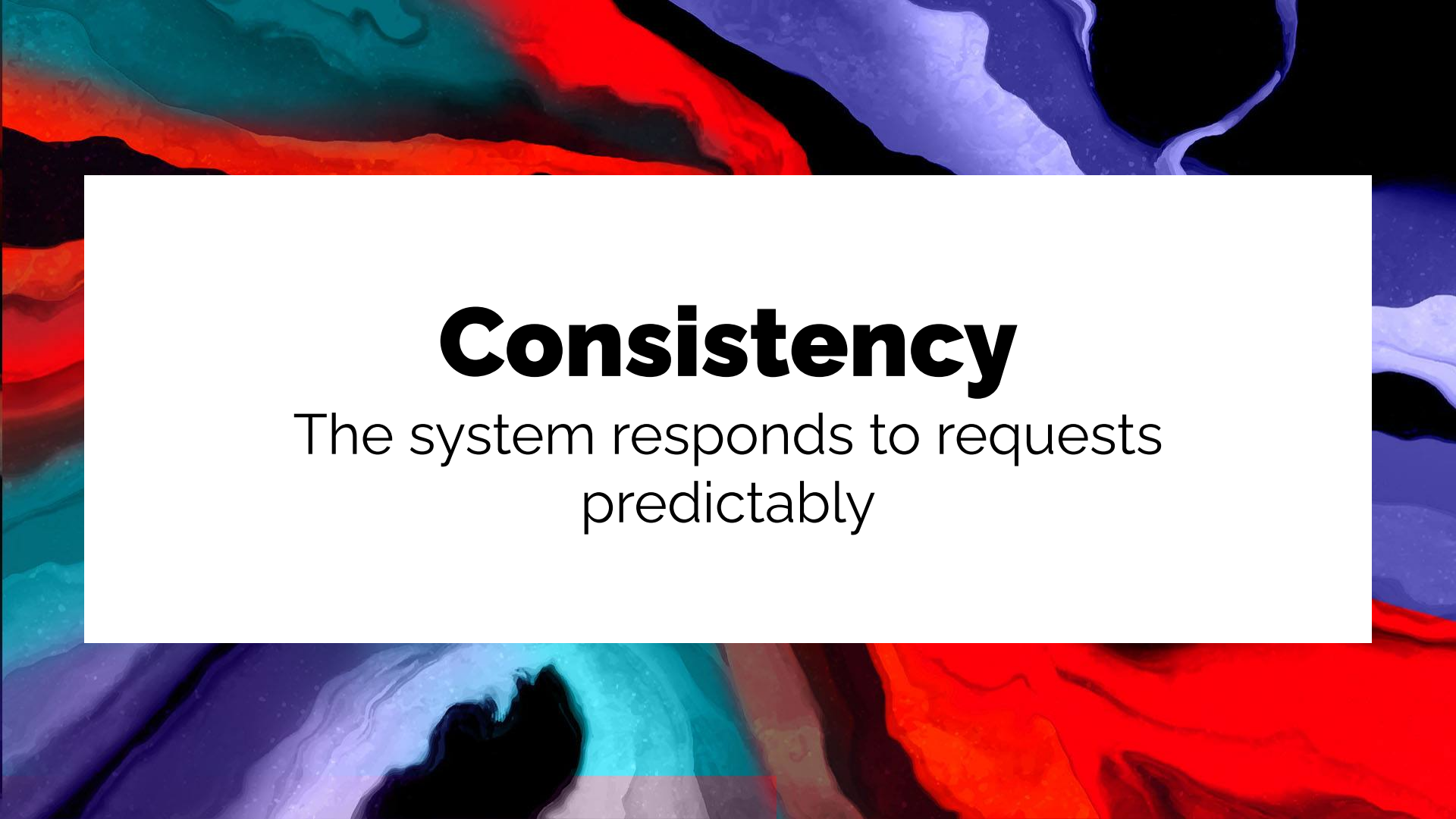
Context: A Single Server

The order of the operations determines the responses the system gives.

In a single server system, the server can determine any order it wants.

It is always *consistent.*




Consistency
The system responds to requests predictably

Failure: A Single Server

Failure can occur for a lot of reasons, from crashes to network outages and is a routine.

In the single server system, when the server fails the entire system becomes *unavailable*.

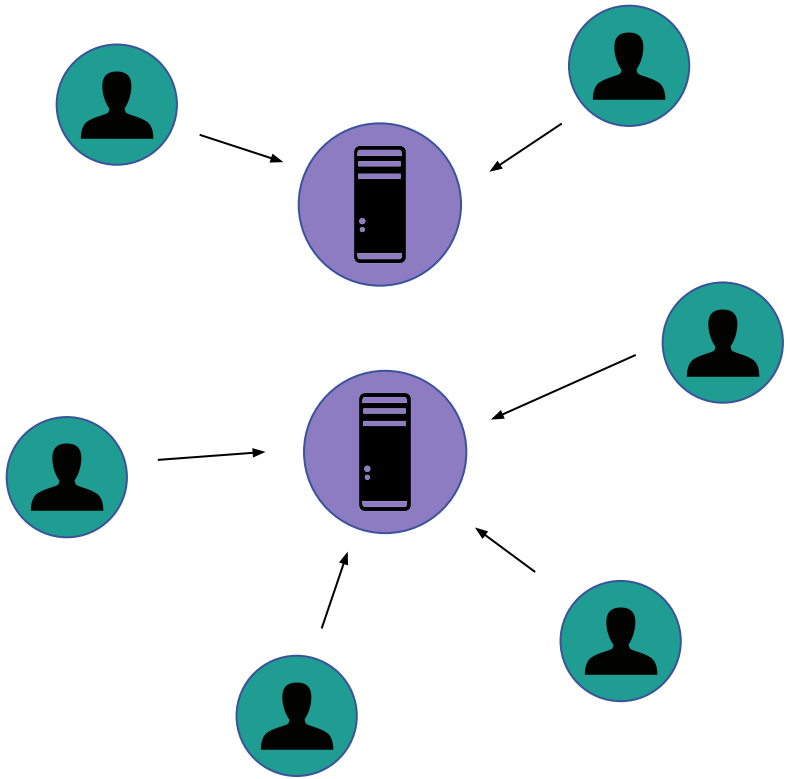
Worse, data loss might occur for any information stored on volatile memory.

Context: A Distributed System

A system is *distributed* when it contains more than one server that must communicate.

If one server in the system fails, it doesn't necessarily become unavailable because other servers can answer requests.

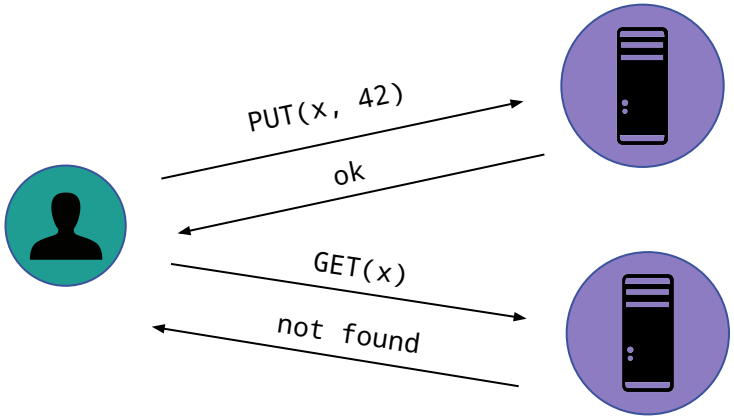
If data is *replicated* we can also avoid data loss.



Inconsistency

When multiple servers participate in the system, they need to communicate in order to ensure they remain in the same state.

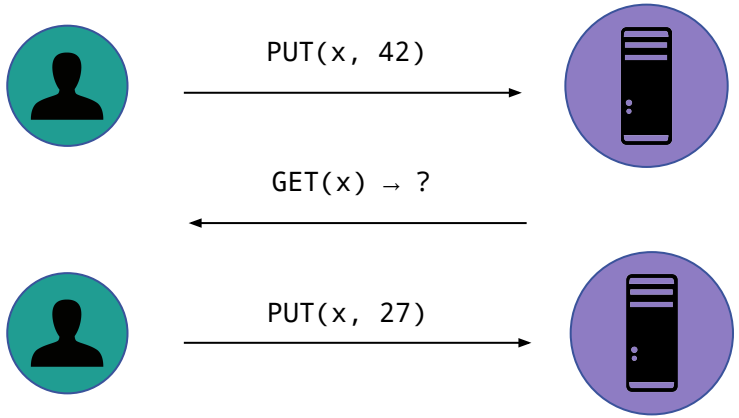
Communication takes time (*latency*) and the more servers in the system, the more time it takes to synchronize.



Concurrency

Because of delays in communication, it is possible for two clients to perform operations *concurrently.* In other words, from the system's perspective, they happen at the same time.

The order of these operations determines the response.



Consistency Levels

Synchronization and ordering increase the amount of time it takes to make requests, however consistency can be relaxed to improve performance.

Strong Consistency: any GET request is guaranteed to return the most recent PUT request.

Causal Consistency: any PUT request will only be delayed by dependent keys, not all keys.

Eventual Consistency: in the absence of PUT requests, the system will eventually become consistent.

Monotonic Reads: every GET returns a value that is more up to date.

Consistency Levels

Most data systems that exist today are eventually consistent, preferring performance over a low likelihood of inconsistency.

Strong Consistency: any GET request is guaranteed to return the most recent PUT request.

Causal Consistency: any PUT request will only be delayed by dependent keys, not all keys.

Eventual Consistency: in the absence of PUT requests, the system will eventually become consistent.

Monotonic Reads: every GET returns a value that is more up to date.

Consistency Levels

However, many applications require **strong** consistency in addition to performance but have to rely on centralized systems.

Strong Consistency: any GET request is guaranteed to return the most recent PUT request.

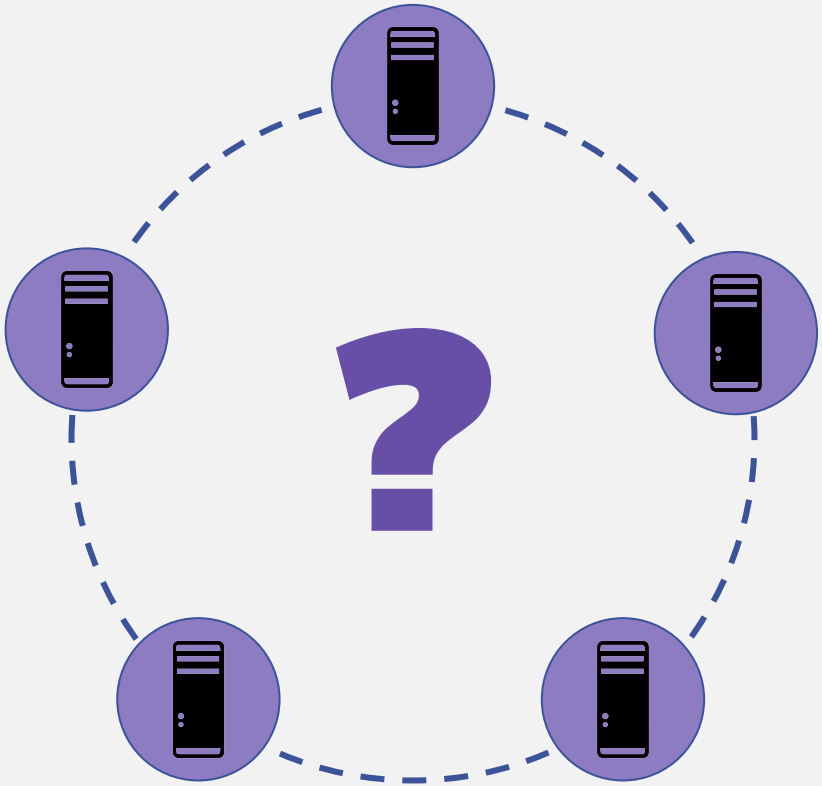
Causal Consistency: any PUT request will only be delayed by dependent keys, not all keys.

Eventual Consistency: in the absence of PUT requests, the system will eventually become consistent.

Monotonic Reads: every GET returns a value that is more up to date.

Distributed Consensus

Fault Tolerant Decision Making for a Network of Replicas.

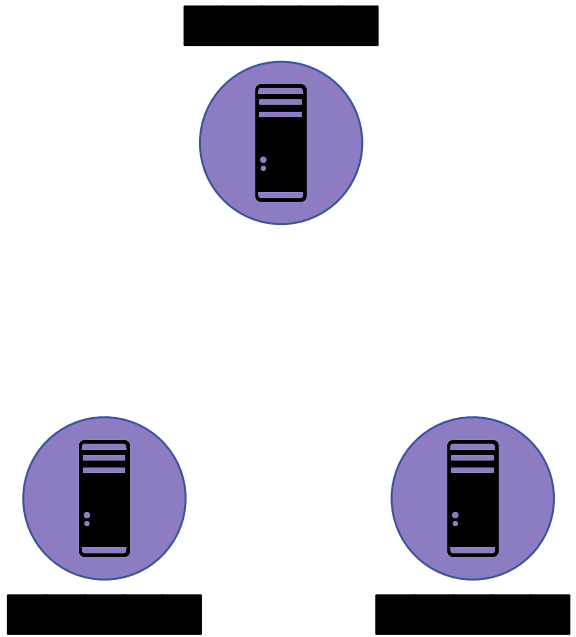


Paxos Consensus

Every server is a state machine that can apply commands in a single order.

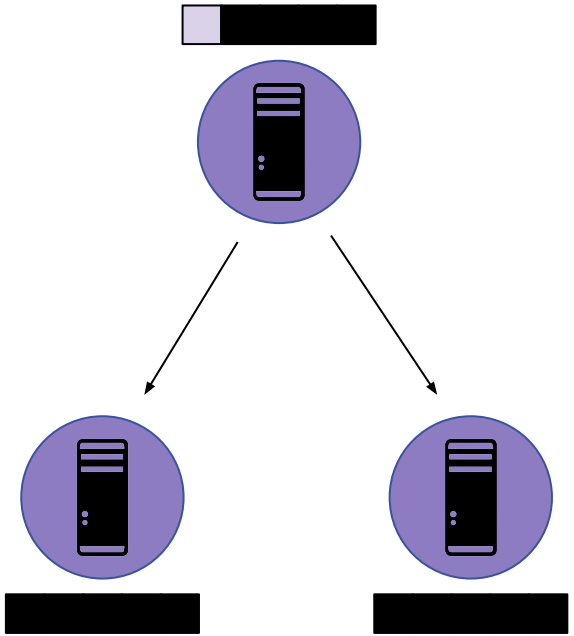
Each server maintains a log of the operations (e.g. GET or PUT), and applies those operations when the entry in the log is **committed.**

Committing entries happens by majority vote as follows.



Paxos Consensus

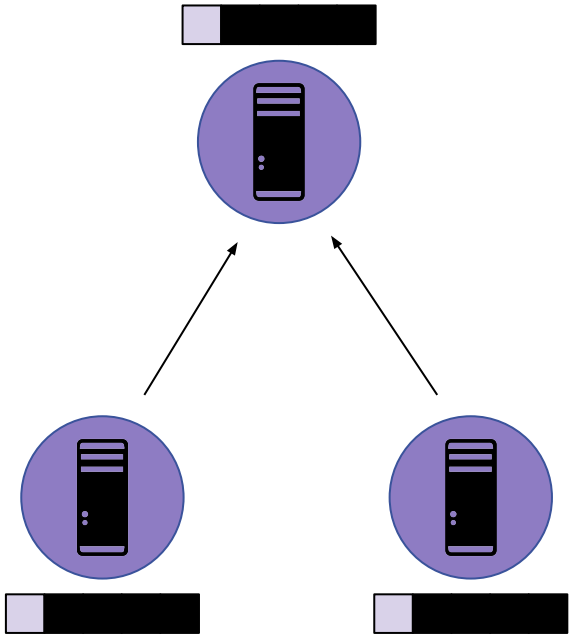
When a client makes a request, the server requests a slot in the log to apply a command, the “prepare” phase.



Paxos Consensus

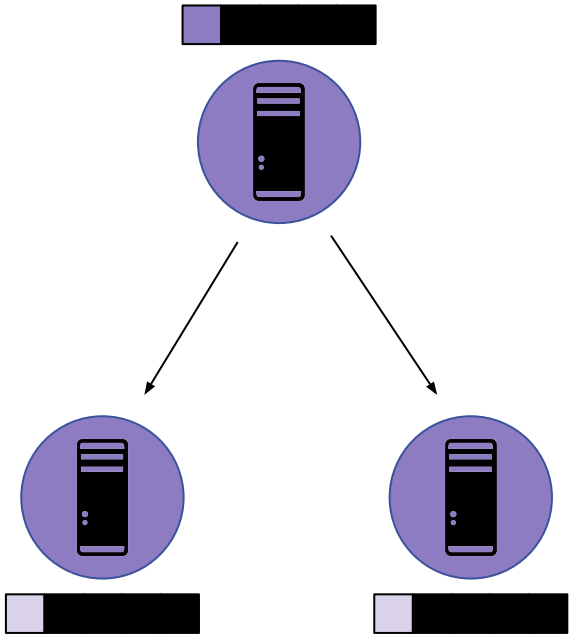
When a client makes a request, the server requests a slot in the log to apply a command, the “prepare” phase.

If the other servers have that spot free, they will reserve the slot for the requesting server.



Paxos Consensus

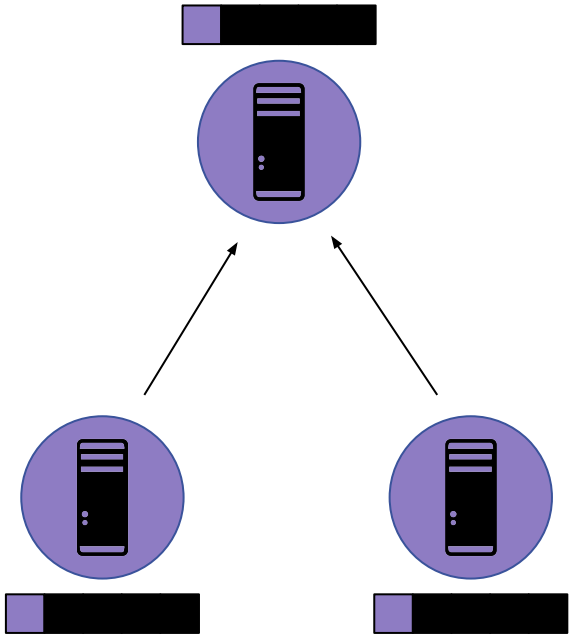
If a majority of servers responds to the prepare phase, the originating server, will send the command to be applied to the log in that spot, the accept phase.



Paxos Consensus

If a majority of servers responds to the prepare phase, the originating server, will send the command to be applied to the log in that spot, the accept phase.

If a majority of servers reply to the accept phase, the entry in the log is **committed.**

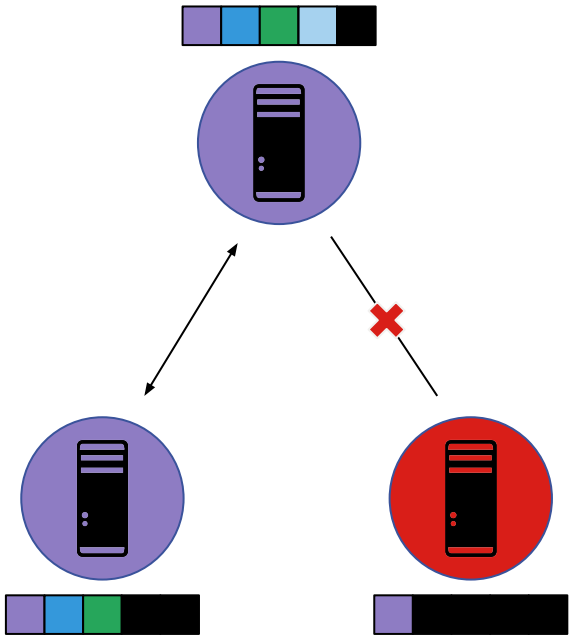


Paxos Consensus

Even if servers fail, so long as a majority of servers are still running, decisions can be made.

Once the server returns, it can be brought back up to date by the other servers.

Rules for responding to the prepare and accept phases ensure that there will only ever be one log order.

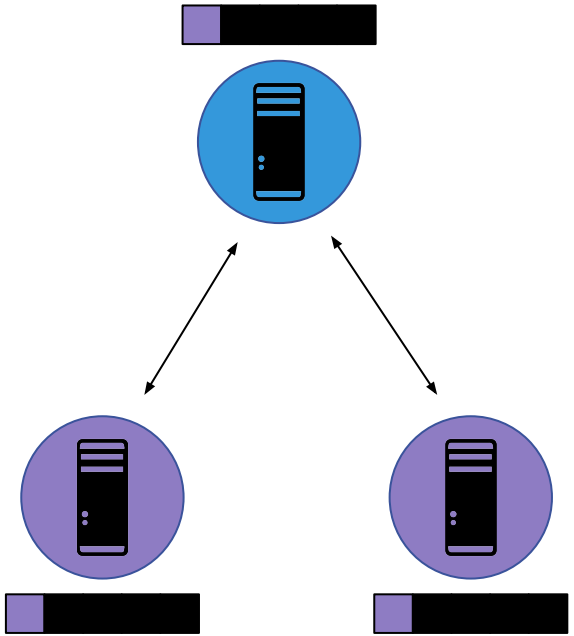


Leader Optimization: Raft & Multi-Paxos

An optimization where the prepare phase is performed once, ahead of time by electing a designated leader.

Heartbeats are used to determine if the leader has died and a new leader must be elected.

This results in faster responses to clients, but small outages during elections.

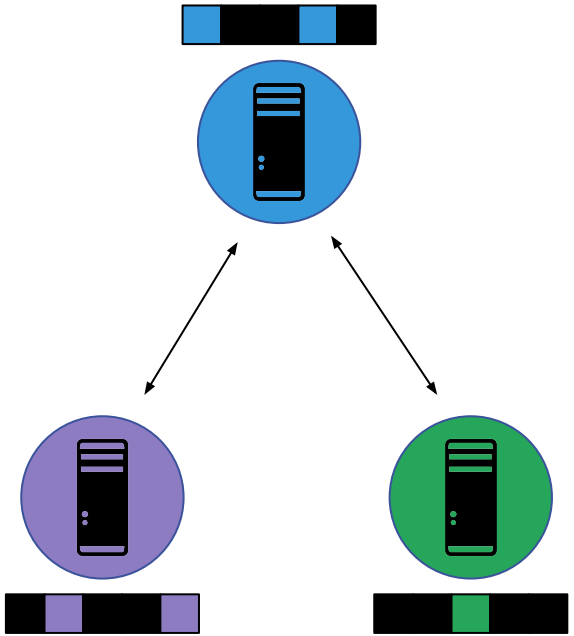


Ballot Optimization: Mencius

To avoid a prepare phase, slots are granted to leaders in a predetermined fashion (e.g. round-robin).

Clients tend to broadcast to multiple leaders in order to apply the command to the next available slot.

Good for dense workloads with continuous accesses. Compaction and forwarding also help manage “empty” log slots.

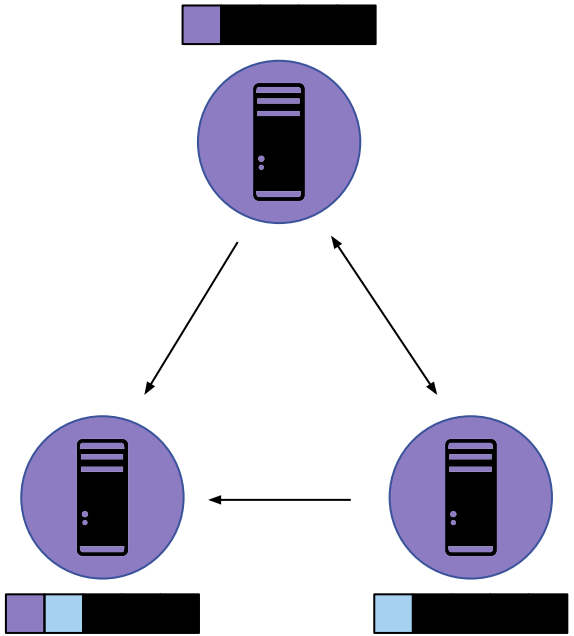


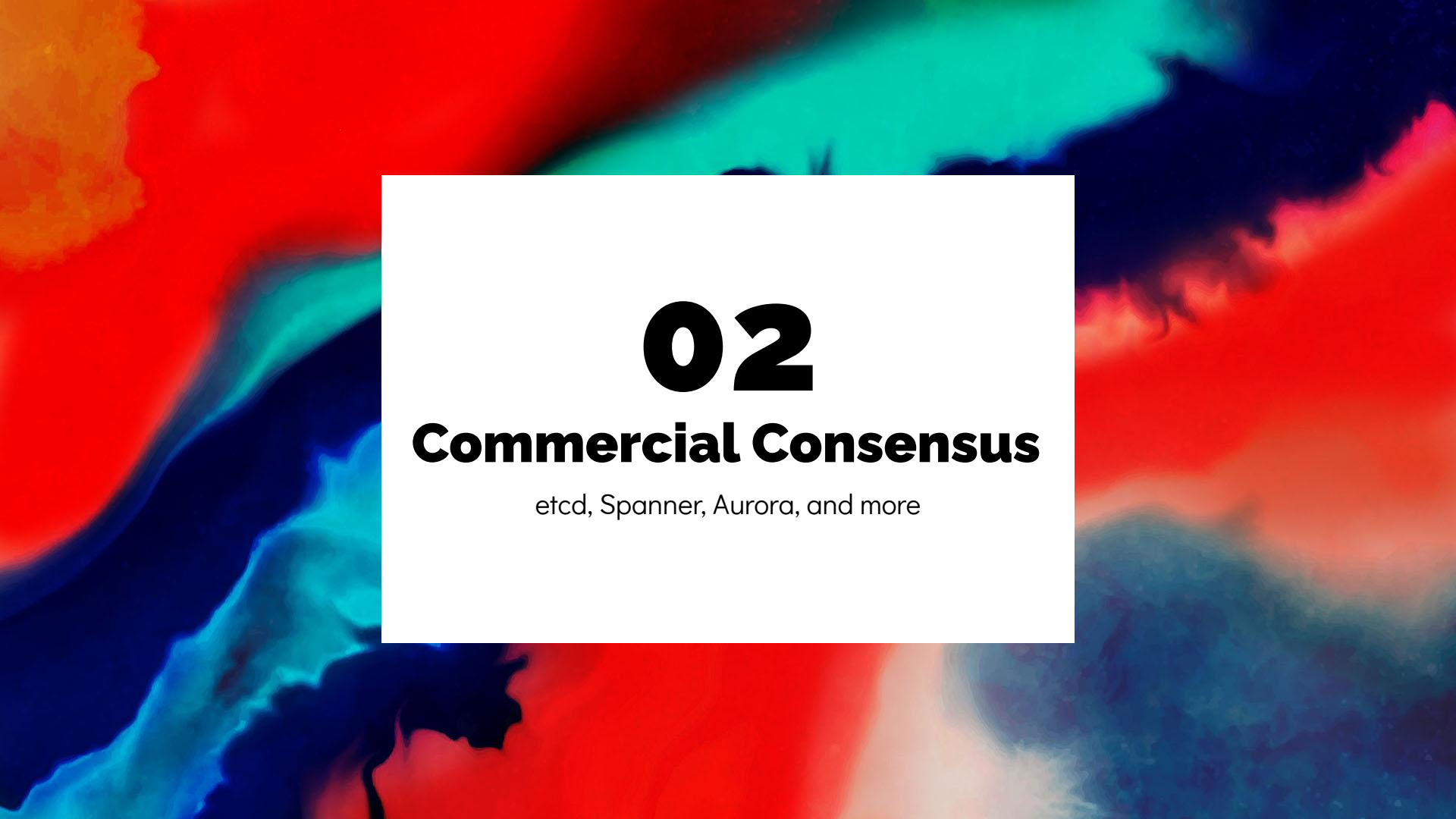
Optimistic Consensus: Fast Paxos, ePaxos

Attempt to commit a command in the first phase, known as the “fast path”. During the fast path commit, conflict detection is applied.

If a conflict is detected, then perform regular 2 phase Paxos (slow path).

If conflicts are rare, most accesses will be fast path. But conflicts require 3 communication phases.





02

Commercial Consensus

etcd, Spanner, Aurora, and more

Chubby, Zookeeper, etcd, etcetera

2001: Lamport publishes “Paxos Made Simple”

2006: Google develops Chubby, a distributed locking service based on Paxos, to rescue the GFS

2010: Apache Zookeeper offers an open source version of Chubby using a Paxos-variant called ZAB

2013: CoreOS releases etcd, based on Raft, to manage a cluster of Container Linux.

2014: Google launches k8s using etcd for the configuration store.

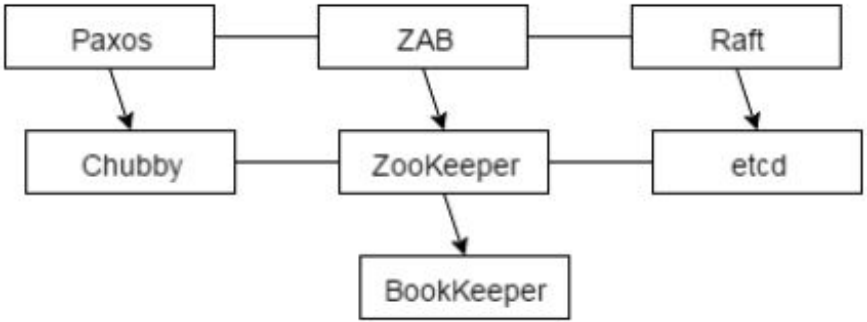
Google

APACHE ZooKeeper™

etcd

kubernetes

Chubby, Zookeeper, etcd, etcetera



Patterns	Frequency	Data Volume	API	Paxos system use	Better substitute
SR	High	Large	Hard	Bad	Replication Protocol
LR	High	Medium	Hard	Bad	Replication Protocol
SS	Low	Small	Hard	OK	Distributed Locks
BO	Low	Depends	Easy	OK	
SD	Low	Small	Easy	Good	
GM	Low	Small	Easy	Good	
LE	Low	Small	Easy	Good	
MM	Medium	Medium	Easy	Good	Distributed Datastore
Q	High	Large / Medium	Hard	Bad	Kafka

"ZooKeeper... got popular and became the *de facto* coordination service for cloud computing applications. However, since the bar on using the ZooKeeper interface was so low, it has been abused/misused by many applications.

When ZooKeeper is improperly used, it often constituted the bottleneck in performance of these applications and caused scalability problems."

Ailijiang, Charapko, Demirbas (2016)

Chubby, Zookeeper, etcd, etcetera

	Systems		
Feature	**Chubby**	**ZooKeeper**	**etcd**
Filesystem API	✓	✓	✓
Watches	✓	✓	✓
Ephemeral Storage	✓	✓	✓
Local Reads	✓	✓	
Dynamic Reconfiguration		✓	✓
Observers	✓	✓	✓
Autoincremented Keys		✓	✓
Hidden Data			✓
Weighted Replicas		✓	

"Despite the increased choices and specialization of Paxos protocols and Paxos systems, the confusion remains about the proper use cases of these systems and about which systems are more suitable for which tasks."

Ailijiang, Charapko, Demirbas (2016)

Globally Distributed Databases

Cockroach DB

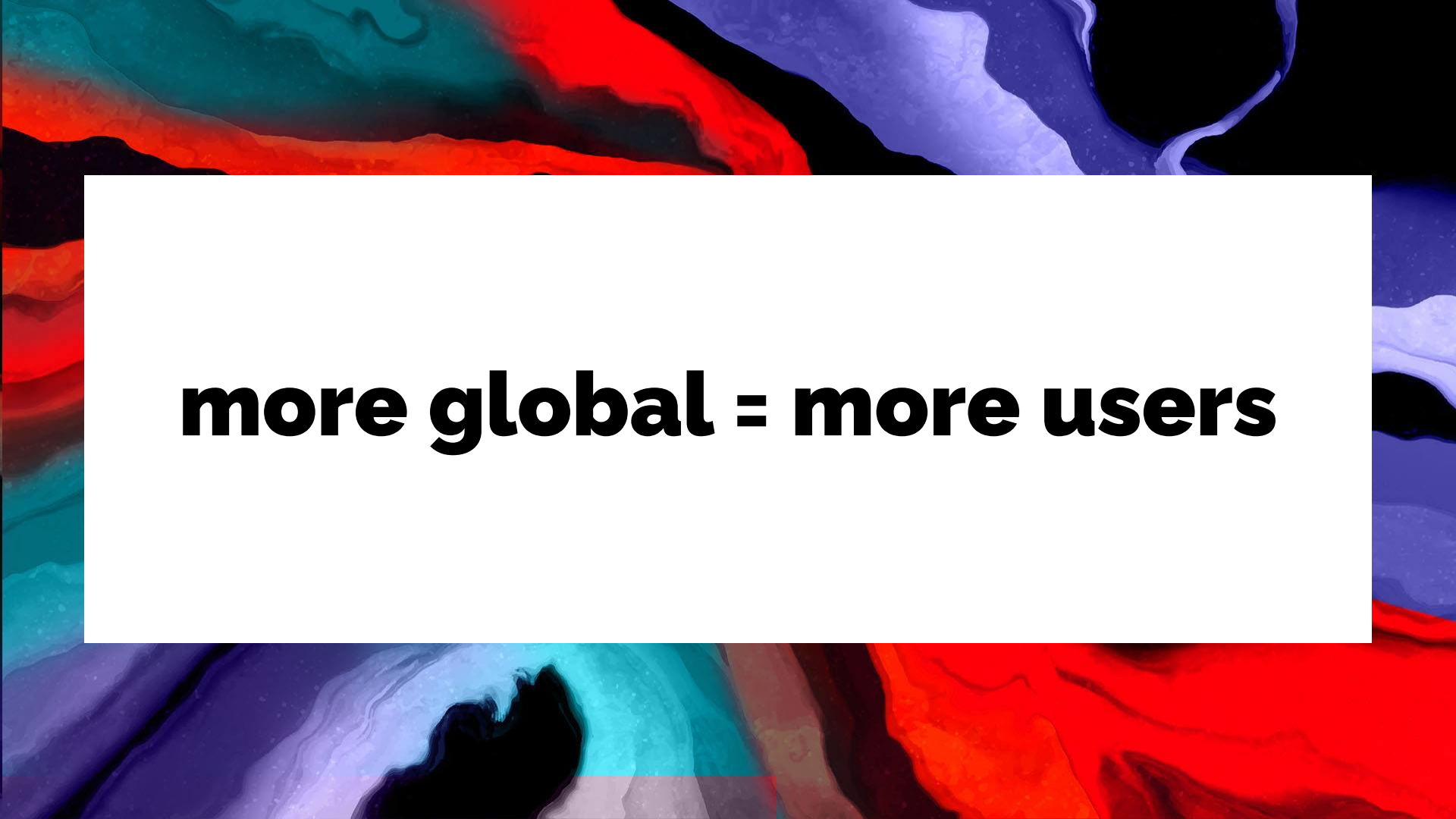
yugabyteDB

Google
Cloud
Spanner

FAUNA

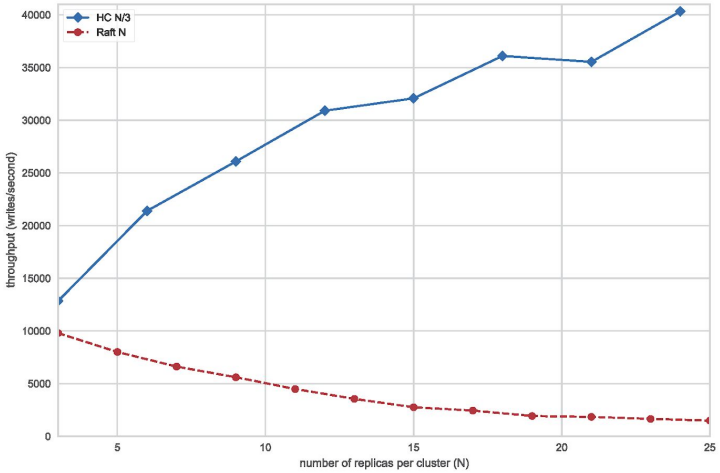
TiDB

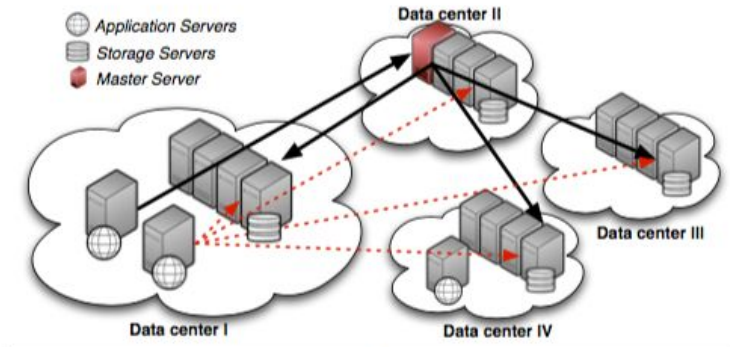
Amazon Aurora


more global = more users

But... scaling consensus is hard

- The larger the quorum, the slower it is to respond to requests.
- Things get worse when you have servers in different data centers:
  - Latency increases because of physical limits.
  - Network partitions can cut off groups of servers.
  - Servers respond more quickly to colocated clients.





Commercial cloud is designed to work best here, so hopefully that's where your users are!

Legalities of Global Systems

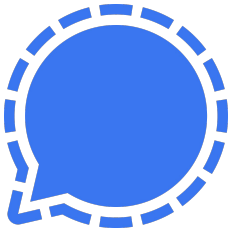
Systems are regulated in the countries where the data resides (where the servers are).

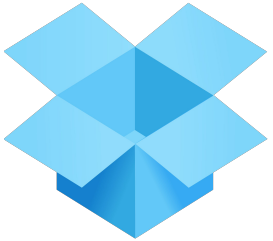
Building a global app now means navigating the complex waters of data compliance.

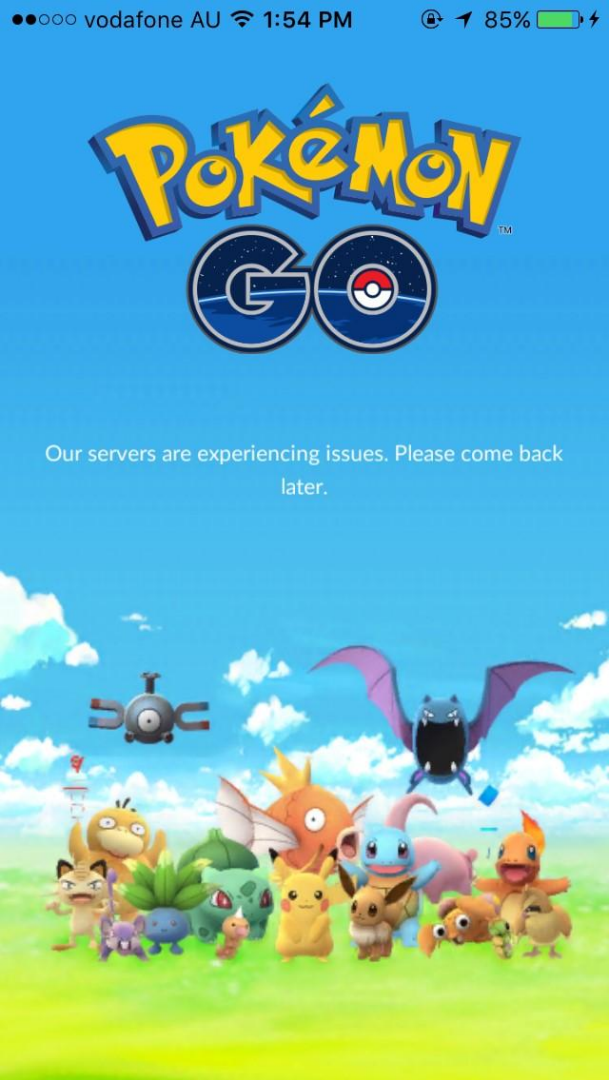
03
Growing Pains

The downsides of success

Case Studies







"The launch of the augmented reality game Pokémon Go was an unmitigated disaster. Due to extremely overloaded servers from the release's extreme popularity, users could not download the game, login, create avatars, or find augmented reality artifacts in their locales.

The game world was hosted by a suite of Google Cloud services, primarily backed by the Cloud Datastore, a geographically distributed NoSQL database. Scaling the application to millions of users therefore involved provisioning extra capacity to the database by increasing the number of shards as well as improving load balancing and autoscaling of application logic run in Kubernetes containers."

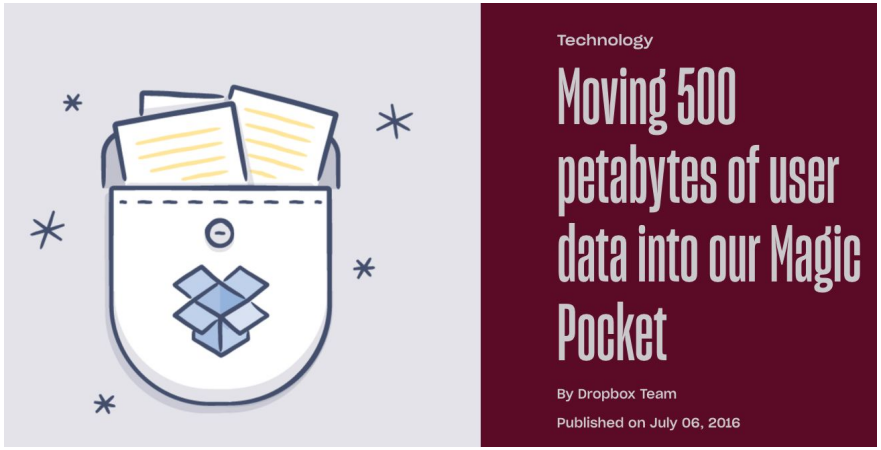
Bengfort (2019)

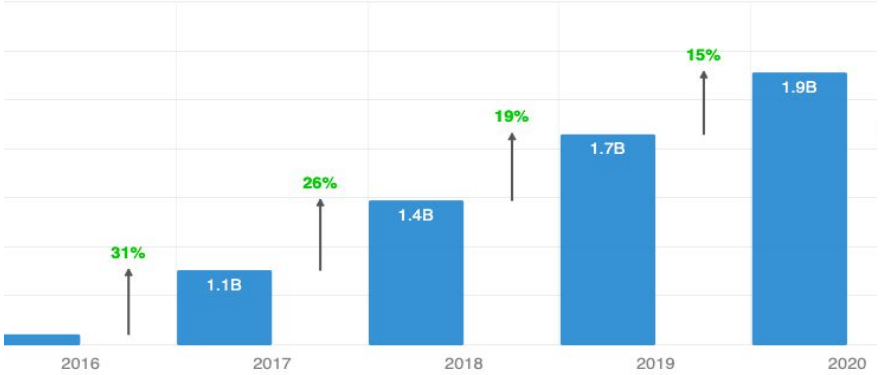
21 February 2018: 501c3 Signal Technology Foundation formed

4 Jan 2021: WhatsApp updates their privacy policy re: data sharing with Facebook

7 Jan 2021: Elon Musk tells his 60M followers to switch to Signal

13 January 2021: Signal goes from 10M users to 50M users in *under 24 hours,* bringing the service down for several days.



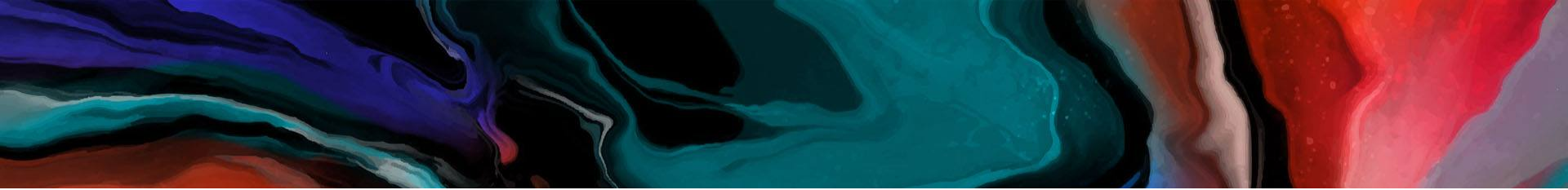


Dropbox Annual Revenue 2016-2020

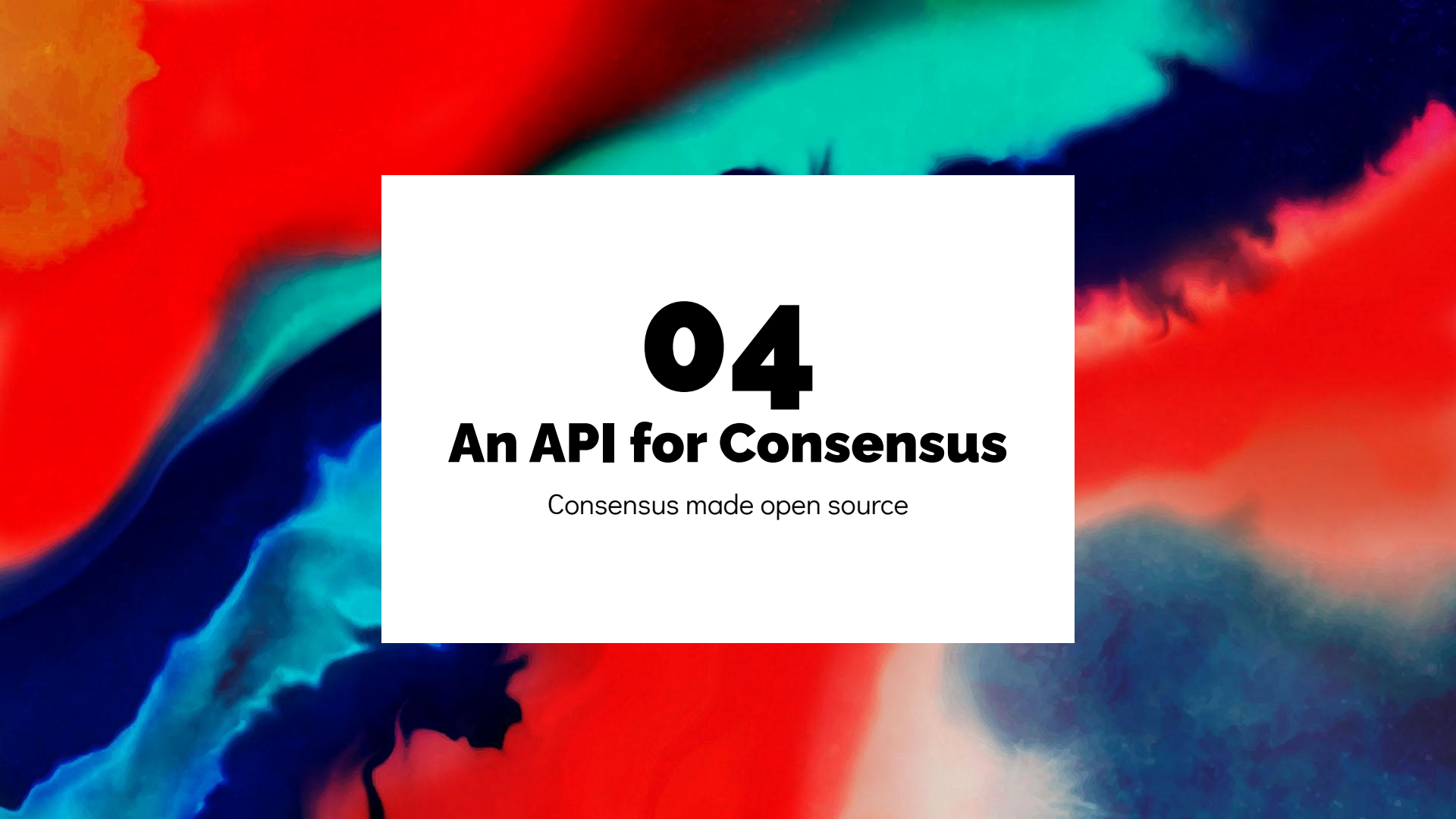
"Between February and October of 2015, Dropbox successfully relocated 90 percent of an estimated 600 petabytes of its customer data to its in-house network of data centers dubbed Magic Pocket."

"It was clear to us from the beginning that we'd have to build everything from scratch," wrote Dropbox infrastructure VP Akhil Gupta on his company blog in 2016, "since there's nothing in the open source community that's proven to work reliably at our scale. Few companies in the world have the same requirements for scale of storage as we do."

Fulton (2020)



Different systems need different consensus solutions



04

An API for Consensus

Consensus made open source

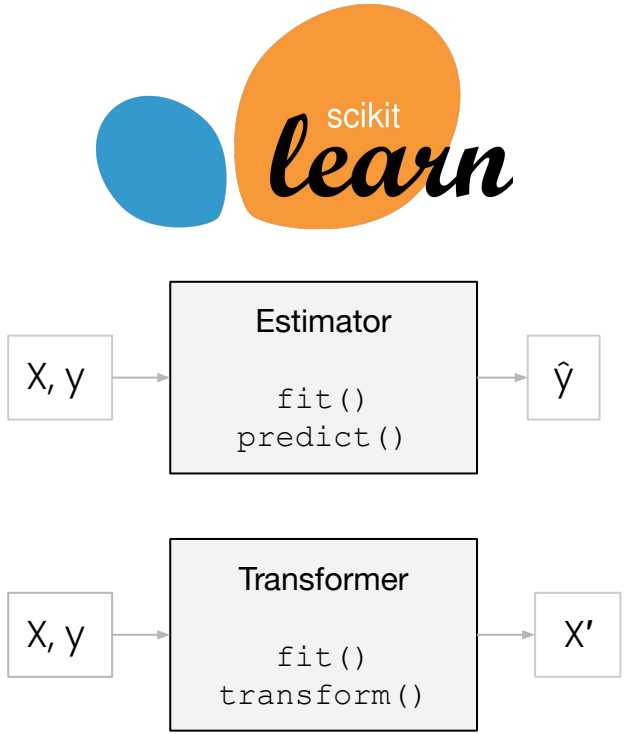
scikit-learn

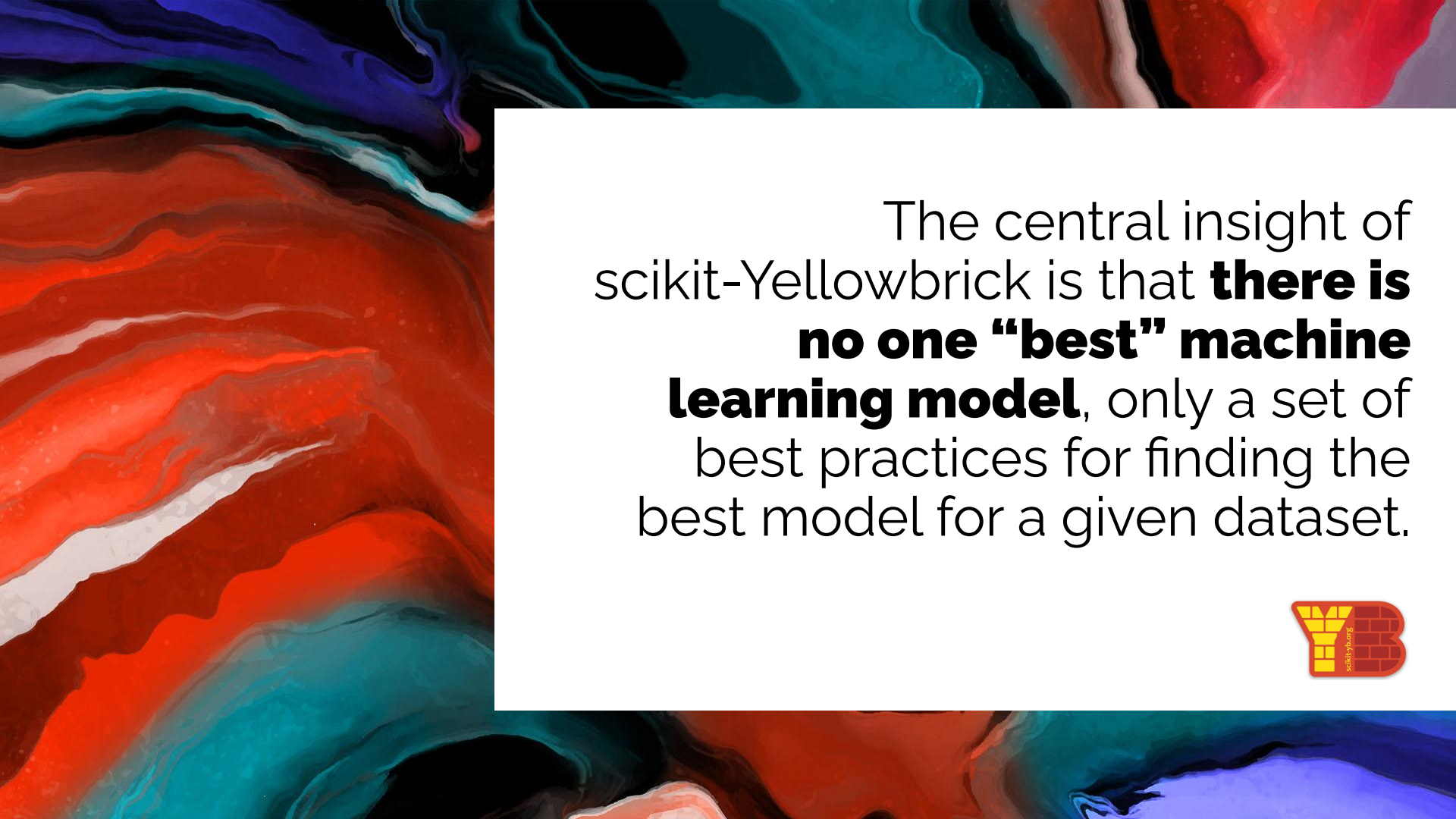
2007: Google Summer of Code project

2010: INRIA releases first open source version

2013: Authors publish “API design for machine learning software: experiences from the scikit-learn project”

2021: 2k contributors, 47k stars, used by 246k projects on Github, including scikit- Yellowbrick!



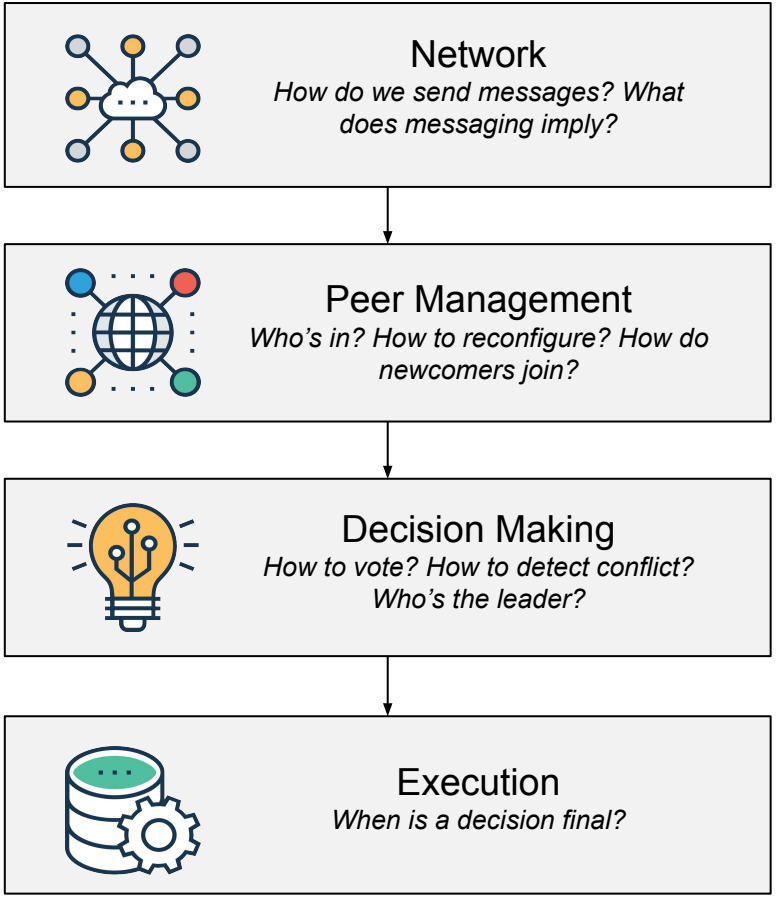

The central insight of scikit-Yellowbrick is that **there is no one "best" machine learning model**, only a set of best practices for finding the best model for a given dataset.
scikit-yb.org

Concur: An API for Consensus

```
sys := &system.New{
    QuorumSize: 7,
    Network: &message.Stream{
         Protocol: grpc,
         Volume: broadcast
    },
    PeerManagement: dynamic,
    DecisionMaking: LeadershipStrategy,
    Execution: onCommit,
}

if err := sys.Validate(); err != nil {
    return errors.New(“Invalid system: ”, err)
}

sys.Concur()
```




Want to contribute?
tinyurl.com/concurapi
Want to vent?
rebecca@rotational.io

Thank You