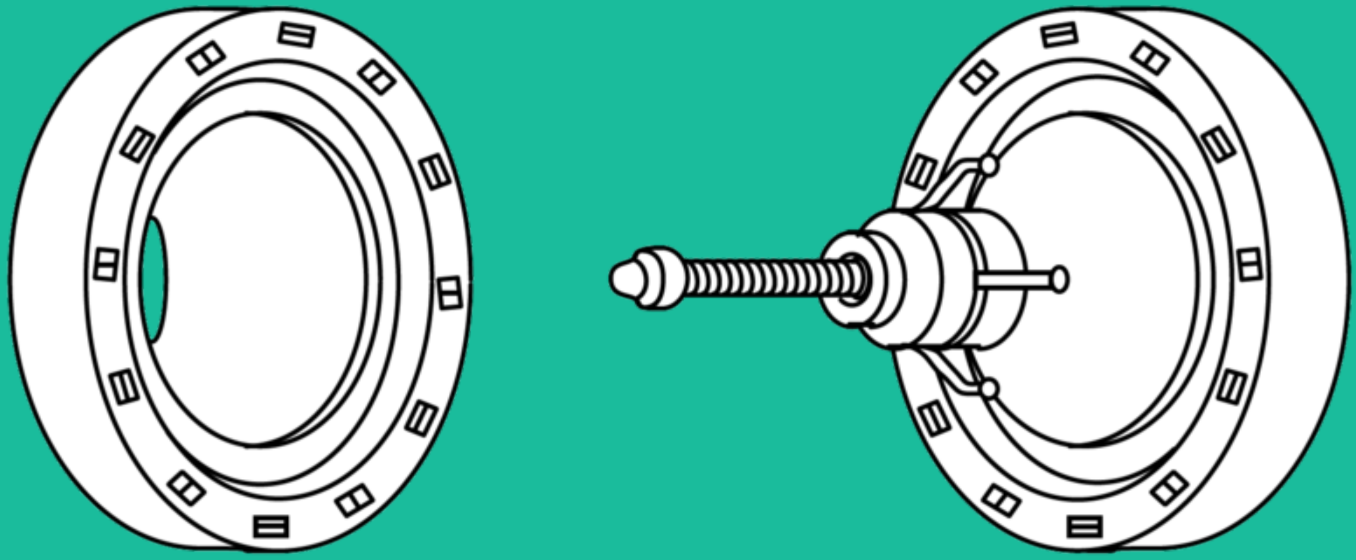


Chapter 8. Resource Management & Coordination

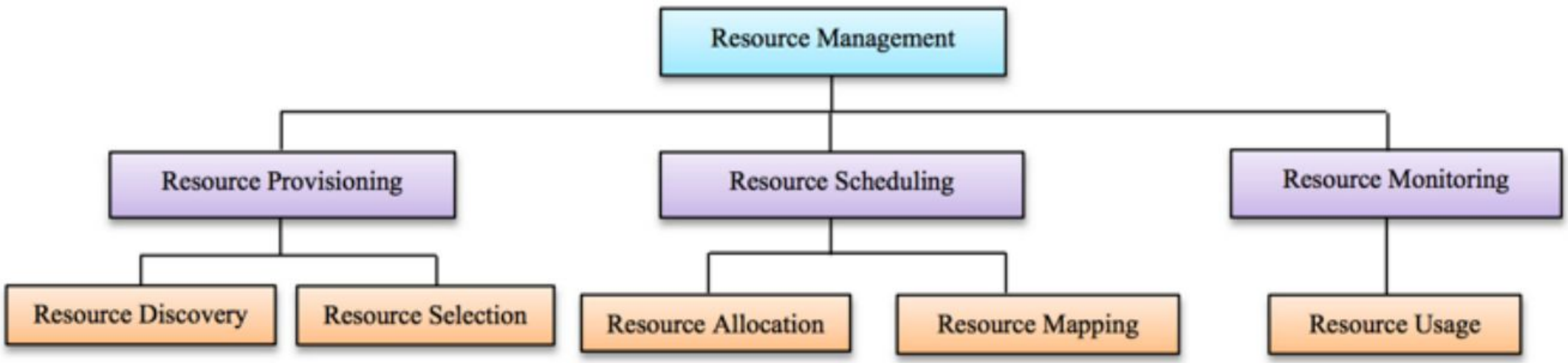
Bilkent University | CS443 | 2020, Spring | Dr. Orçun Dayıbaş

Cloud Resources

- **Computation**
  - Processor, Memory, Algorithms, APIs
- **Storage**
  - Hard/Flash drive, SW (Object store, DFS, etc.), DBs
- **Communication**
  - Physical (Routers, switches, cables, etc.)
  - Logical (Bandwidth, delay, protocols, etc.)
- **Power/Energy**
  - UPS, HVAC systems, etc.
- **Security**
  - Trust, Authentication, Integrity, Privacy

Cloud Resources

- **Resource management**
  - Provisioning
    - Simply provide it (conf. mng., deployment, etc.). Ex: Ansible
  - Scheduling
    - Utilize resource pool (hard to optimize). Ex: Titus, Kubernetes (Autoscaling)
  - Monitoring
    - Measure to manage. Ex: Prometheus



source: https://www.researchgate.net/publication/316494357

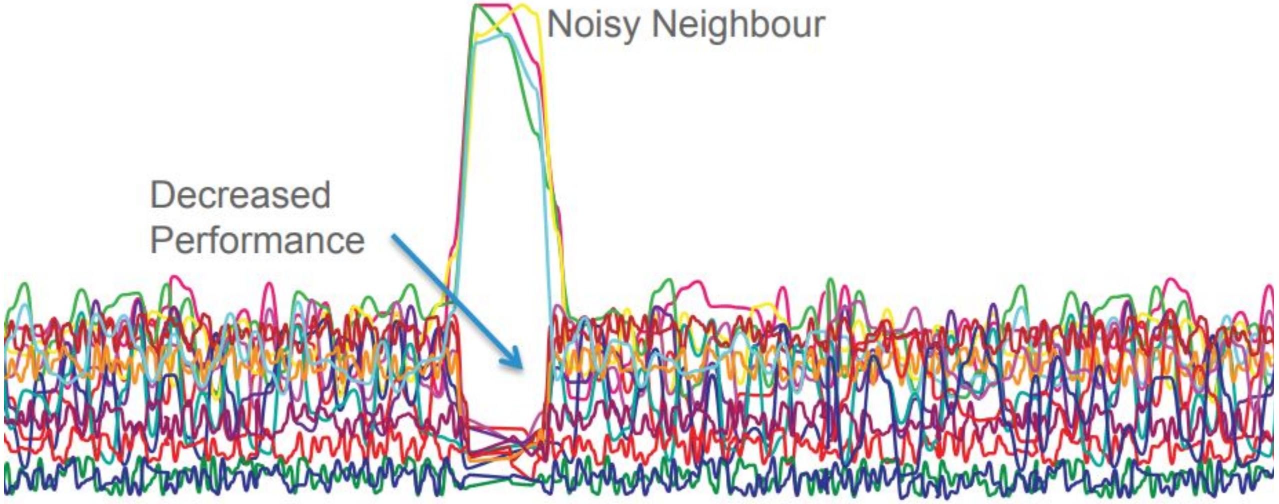
Cloud Resources

- **Everything is SW & every SW is containerized**
  - XaaS, Software-defined X
  - Elasticity → Automation of everything
  - Resource management ~= container orchestration
- **Container Orchestration Process**

Scheduling	Resource Management	Service Management
Placement	Memory	
Replication/ Scaling	CPU	Labels
Resurrection	GPU	Groups/ Namespaces
Rescheduling	Volumes	Dependencies
Rolling Deployment	Ports	Load Balancing
Upgrades	IPs	Readiness Checking
Downgrades		
Collocation		

Cloud Resources

- **Noisy Neighbors**
    - Noisy neighbor is a phrase used to describe a cloud computing infrastructure co-tenant that monopolizes bandwidth, disk I/O, CPU and other resources, and can negatively affect other users' cloud performance.
    - The noisy neighbor effect causes other containers/applications that share the infrastructure to suffer from uneven cloud network performance.



https://www.slideshare.net/Databarracks/controlling-performance-in-the-cloud-taking-charge-of-your-hosting-environment

Data Replication

- **Redundancy**
  - Cloud resources (individually or as a whole like hybrid cloud)
    - Remember ANSI/TIA-942 standard tiers (see chapter-6)
  - It is perfectly OK for
    - Computation, Communication, Power or even storage
  - But redundant “meta”data → Inconsistent system
    - We need to solve that part
- **Data replication techniques**
  - Gossip/multicast protocols
    - Epidemic broadcast trees, bimodal multicast, SWIM, HyParView, etc.
    - Do not solve inconsistency problem
  - Consensus protocols
    - Paxos, Raft, Zab, etc.
    - Solves inconsistency problem

Distributed Consensus

- **Distributed systems**
  - Modern systems/solutions are distributed
  - Distributed systems are harder to implement
    - Lack of global knowledge: all you have exchanged messages, up-to-date?
    - Time: Clock skew, msg. order (delay/duplicate messages)
    - Consistency: Concurrent operations, conflict, consistent state
    - Failures: detecting and recovering
  - Remember chapter-2 (slide #22 to be exact)

- **Definition**
  - "A collection of independent computers that appear to its users as one computer" A.T.
  - Three characteristics
    - The computers run concurrently
    - The computers fail independently
    - The computers don't share a global clock

Distributed Consensus

- **Definition**
  - A distributed consensus ensures a consensus of data among nodes in a distributed system or reaches an agreement on a proposal
  - The real world applications include clock synchronization, opinion formation, smart power grids, state estimation, control of UAVs (and multiple robots/agents in general), load balancing, blockchain and others
- **Consensus algorithms/protocols**
  - Paxos Family
    - Multi-paxos, EPaxos, WPaxos, Cheap/Fast Paxos, etc.
  - Raft, Zab
  - Bitcoin/cryptocurrency ecosystem???
    - Nakamoto Consensus
    - Proof-of-work

Distributed Consensus

- **Protocols & Implementations**

System	Protocol	Implementation	Usage
Google GFS	Multi-Paxos	Chubby	Lock Service
Google Spanner	Multi-Paxos	Chubby	
Google Borg	Multi-Paxos	Chubby	Configuration, Master election
Apache HDFS	Zab	ZooKeeper	Failure detection, Active NameNode election
Apache Giraph	Zab	ZooKeeper	Coordination, Configuration, Aggregators
Apache Hama	Zab	ZooKeeper	Coordination
CoreOS	Raft	etcd	Service Discovery
OpenStack	Zab	ZooKeeper	Service Discovery
Apache Kafka	Zab	ZooKeeper	Coordination, Configuration
Apache BookKeeper	Zab	ZooKeeper	Coordination, Configuration

source: https://muratbuffalo.blogspot.com/2015/10/consensus-in-wild.html

Paxos

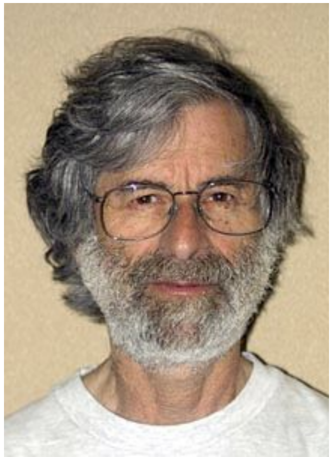
- **Definition**
  - Paxos is a family of protocols for solving consensus in a network of unreliable processors
- **Background**
  - ACM Transactions on Computer Systems
    - Submitted: 1990, Accepted: 1998

Γωνδα is the new cheese inspector

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state machine approach to the design of distributed systems.



Paxos

- **Background**
  - 10 years later, Leslie Lamport revised the original paper

Paxos Made Simple

Leslie Lamport

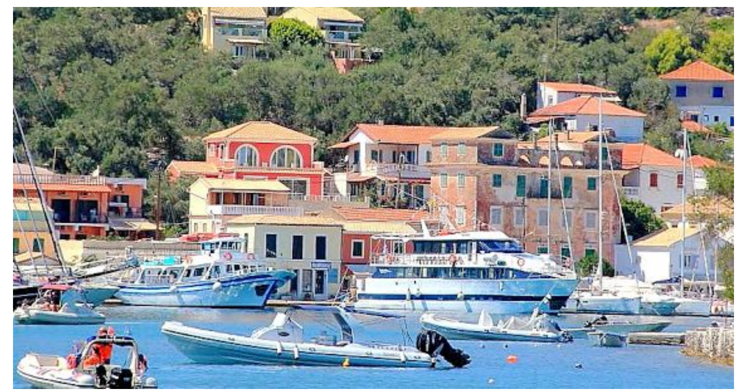
01 Nov 2001

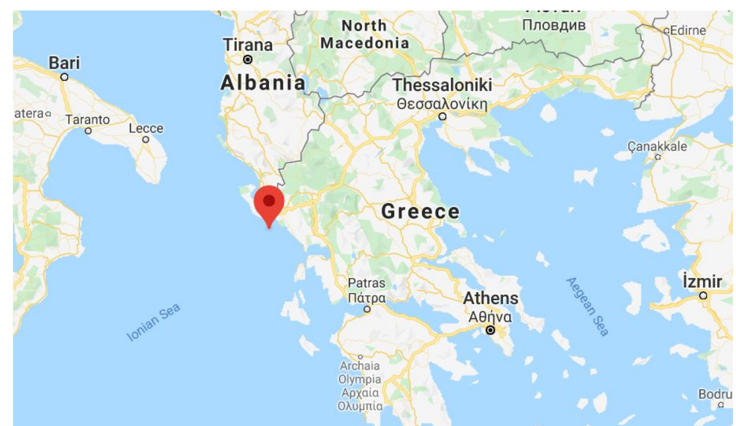
Abstract

The Paxos algorithm, when presented in plain English, is very simple.

1 Introduction

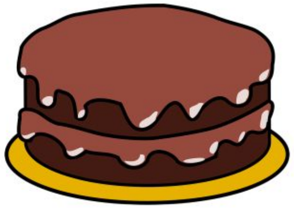
The Paxos algorithm for implementing a fault-tolerant distributed system has been regarded as difficult to understand, perhaps because the original presentation was Greek to many readers [5]. In fact, it is among the simplest and most obvious of distributed algorithms. At its heart is a consensus algorithm—the "synod" algorithm of [5]. The next section shows that this consensus algorithm follows almost unavoidably from the properties we want it to satisfy. The last section explains the complete Paxos algorithm, which is obtained by the straightforward application of consensus to the state machine approach for building a distributed system—an approach that should be well-known, since it is the subject of what is probably the most often-cited article on the theory of distributed systems [4].

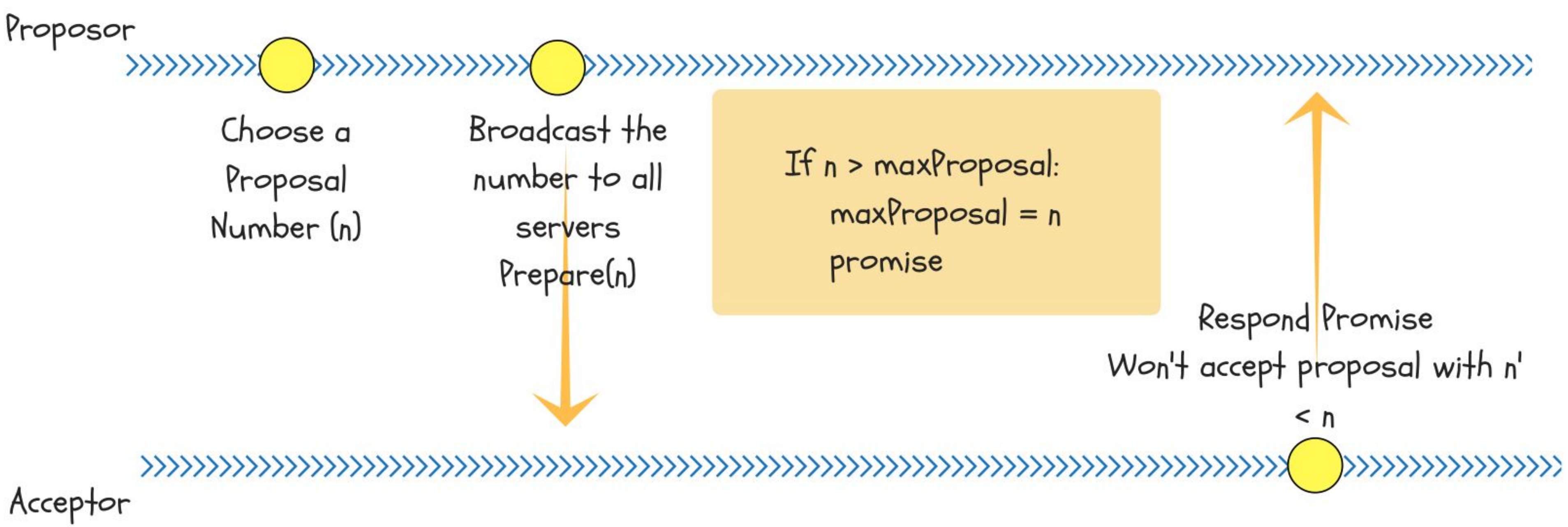




Paxos

- **Basic Paxos**
  - Assume group of people deciding on cake flavor to order
  - Prepare to propose & propose
  - Roles: Proposer, Acceptor

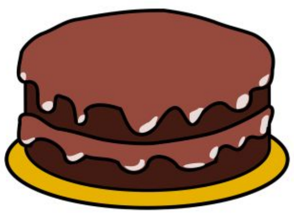


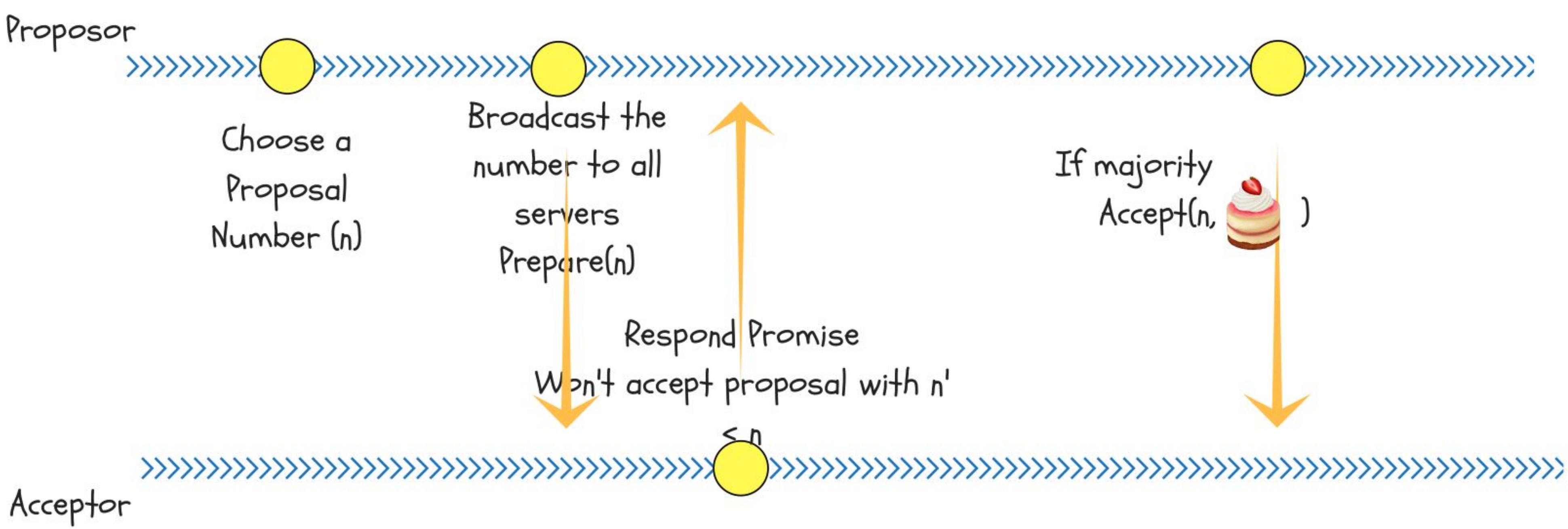


source: https://www.slideshare.net/YifanXing/consensus-algorithmspaperswelove2018sep

Paxos

- **Basic Paxos**
  - Assume group of people deciding on cake flavor to order
  - Prepare to propose & propose
  - Roles: Proposer, Acceptor

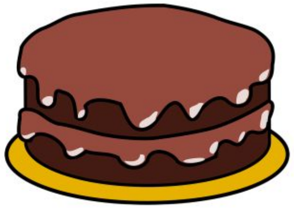


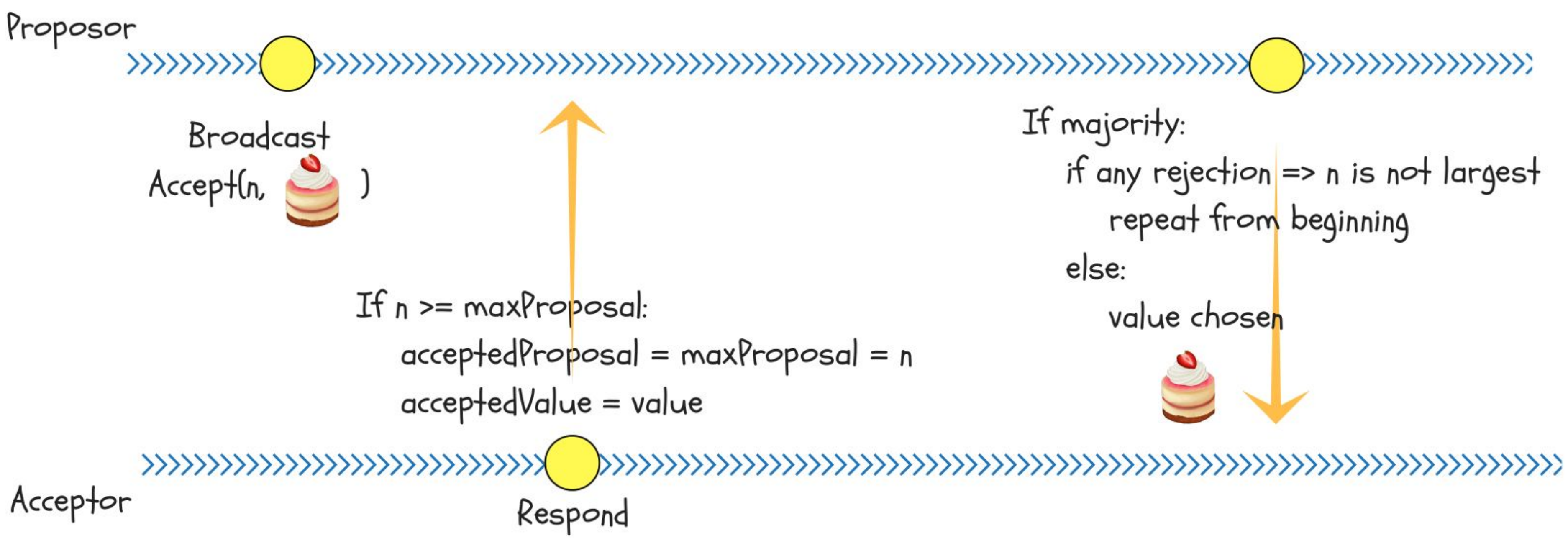


source: https://www.slideshare.net/YifanXing/consensus-algorithmspaperswelove2018sep

Paxos

- **Basic Paxos**
  - Assume group of people deciding on cake flavor to order
  - Prepare to propose & propose
  - Roles: Proposer, Acceptor

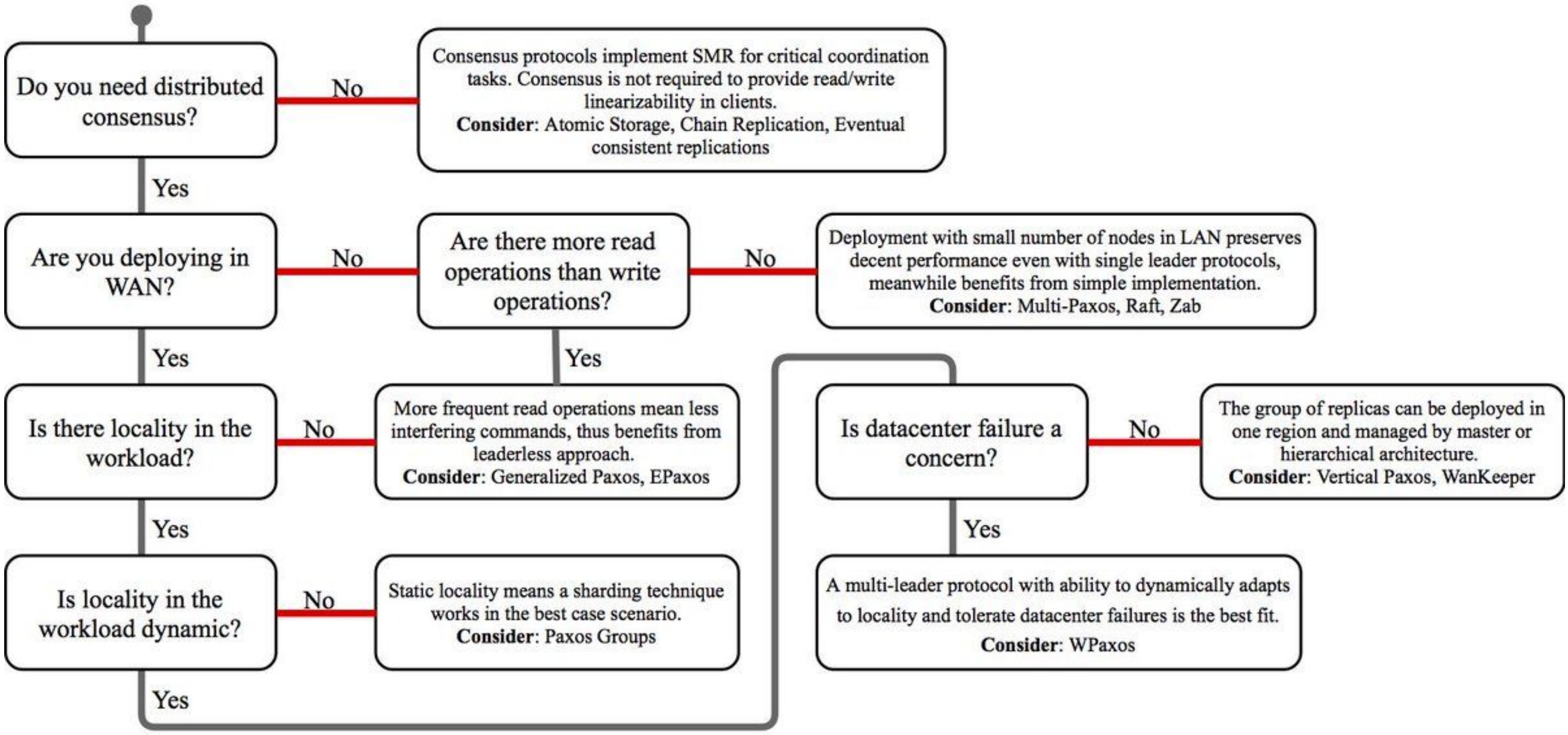




source: https://www.slideshare.net/YifanXing/consensus-algorithmspaperswelove2018sep

Choosing a Paxos Variant

http://paxos.systems/variants.html

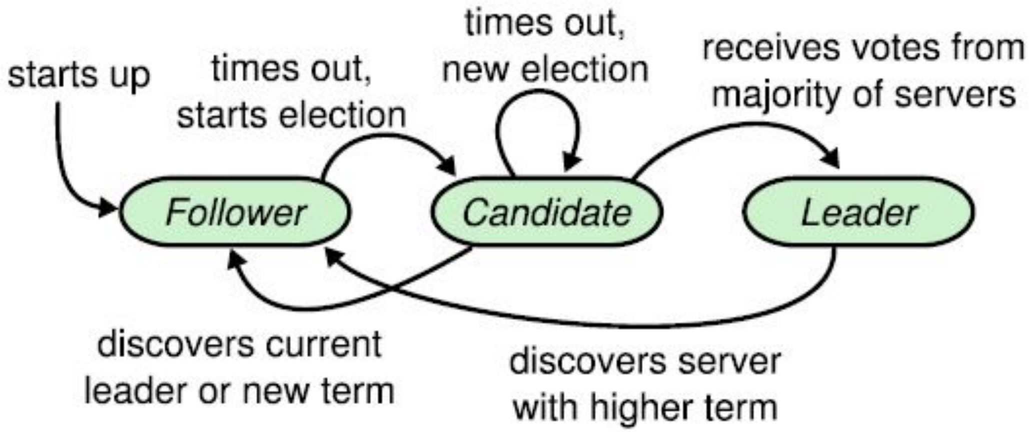


Raft

- **Definition**
  - **R**eliable, **R**eplicated, **R**edundant, **A**nd **F**ault-**T**olerant
  - A consensus algorithm designed as an alternative to Paxos. It was meant to be more **understandable** than Paxos by means of separation of logic, but it is also formally proven safe and offers some additional features
- **Goal**
  - Easier to be implemented (understandable)
  - The system is fully operational as long as **a majority of the servers are up**
- http://thesecretlivesofdata.com/raft/

Raft

- **Raft decomposes consensus into 3 sub-problems**
  - **Leader Election:** A new leader needs to be elected in case of the failure of an existing one
  - **Log replication:** The leader needs to keep the logs of all servers in sync with its own through replication
  - **Safety:** If one of the servers has committed a log entry at a particular index, no other server can apply a different log entry for that index
- **State changes**



Raft

- **Terms**
  - Raft divides time into “terms” of arbitrary length, each beginning with an election. If a candidate wins the election, it remains the leader for the rest of the term. If the vote is split, then that term ends without a leader
  - The term number **increases monotonically**
  - Each server stores the current term number which is also **exchanged in every communication**
- **Leader election**
  - The leader periodically sends **a heartbeat** to its followers to maintain authority. A leader election is triggered when a follower times out after waiting for a heartbeat from the leader
  - This follower transitions to the candidate state and increments its term number

Raft

- **Leader election (cont.)**
  - After voting for itself, it issues RequestVotes RPC in parallel to others in the cluster. Three outcomes are possible:
    - The candidate receives votes from the majority of the servers and becomes the leader. It then sends a heartbeat message to others in the cluster to establish authority.
    - If other candidates receive AppendEntries RPC, they check for the term number. If the term number is greater than their own, they accept the server as the leader and return to follower state. If the term number is smaller, they reject the RPC and still remain a candidate.
    - The candidate neither loses nor wins. If more than one server becomes a candidate at the same time, the vote can be split with no clear majority. In this case a new election begins after one of the candidates times out.
  - Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from a fixed interval (e.g., 150 – 300ms).

Raft

- **Log replication**
  - When a leader gets a client request, it adds it to its own log as a new entry. Each entry in a log:
    - Contains the client specified command
    - Has an index to identify the position of entry in the log (the index starts from 1)
    - Has a term number to logically identify when the entry was written
  - It needs to replicate the entry to all the follower nodes in order to keep the logs consistent.
    - The leader issues AppendEntries RPCs to all other servers in parallel. The leader retries this until all followers safely replicate the new entry.
    - When the entry is replicated **to a majority of servers** by the leader that created it, it is considered **committed**.
    - All the previous entries, including those created by earlier leaders, are also considered committed. The leader executes the entry once it is committed and returns the result to the client.
    - The leader maintains the highest index it knows to be committed in its log and sends it out with the AppendEntries RPCs to its followers. Once the followers find out that the entry has been committed, it applies the entry to its state.

- **Log replication (cont.)**
  - When sending an AppendEntries RPC, the leader includes **the term number and index of the entry** that immediately precedes the new entry. If the follower cannot find a match for this entry in its own log, it rejects the request to append.
    - This consistency check lets the leader conclude that whenever AppendEntries returns successfully from a follower, they have identical logs until the index included in the RPC but the logs of leaders and followers may become inconsistent in the case of leader crashes.
  - The leader tries to find the last index where its log matches that of the follower, deletes extra entries if any, and adds the new ones.

> In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log.

Raft

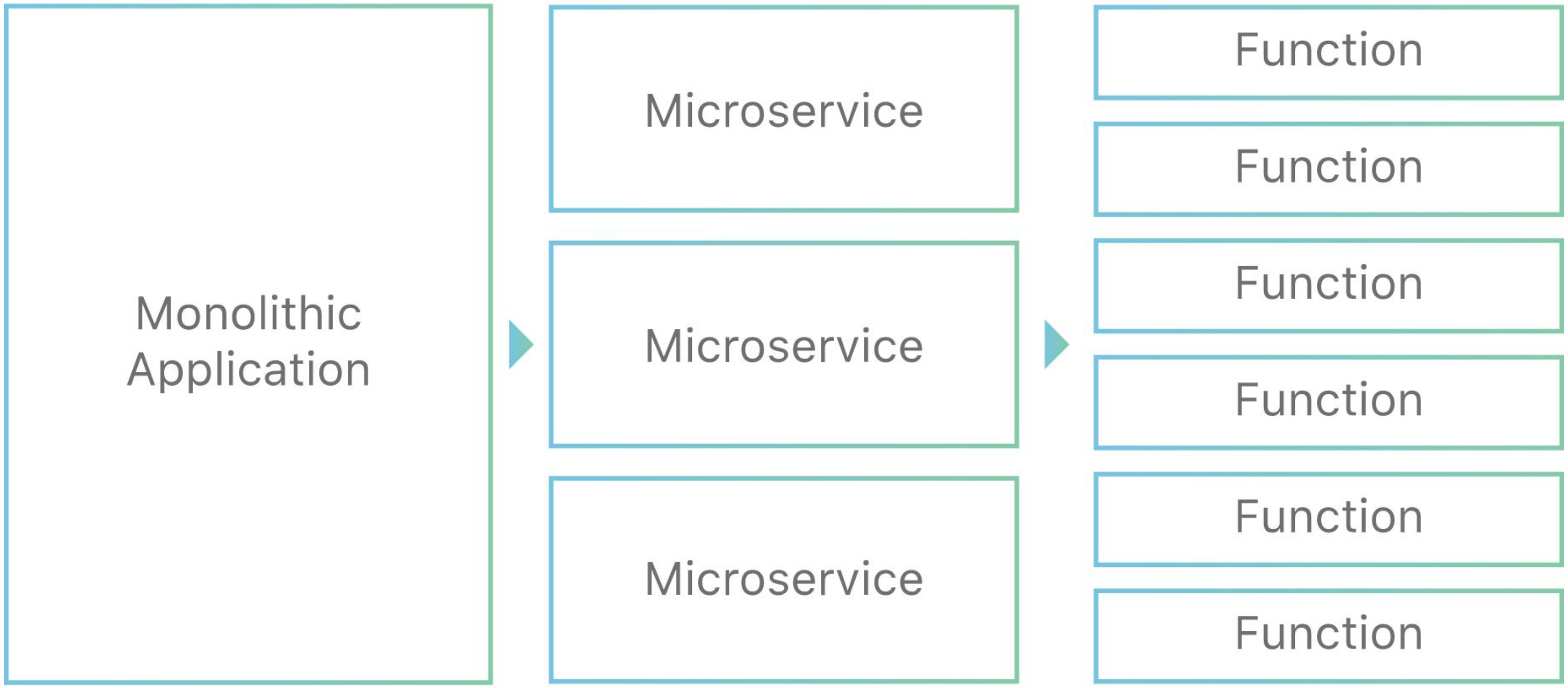
- **Log replication (cont.)**
  - The leader maintains a nextIndex for each follower, which is the index of the next log entry the leader will send to that follower.
    - When a leader first comes to power, it initializes all nextIndex values to the index just after the last one in its log.
  - Whenever AppendRPC returns with a failure for a follower, the leader decrements the nextIndex and issues RPC again.
    - Eventually, nextIndex will reach a value where the logs converge. AppendEntries will succeed when this happens and it can remove extraneous entries (if any) and add new ones from the leaders log (if any). Hence, a successful AppendEntries from a follower guarantees that the leader’s log is consistent with it.
    - With this mechanism, a leader does not need to take any special actions to restore log consistency when it comes to power. It just begins normal operation, and the logs automatically converge in response to failures of the AppendEntries consistency check. A leader never overwrites or deletes entries in its own log.

Distributed Consensus

- **Recap**
  - Consensus on what?
    - Use coordination of servers on “meta”data of system
    - Do not abuse by using it in every occasion (yes, ZooKeeper is KV store but it is designed for a specific purpose): https://www.confluent.io/blog/distributed-consensus-reloaded-apache-zookeeper-and-replication-in-kafka/
  - Implementing consensus algorithms
    - Don’t do that
    - Resilient against issues?
    - Worth it to be resilient? (Scoping problem)
      - Byzantine Fault tolerant Paxos/Raft
  - Designing consensus algorithms
    - Consistency, Reliability, Complexity
    - Understandability

Serverless

- **Functions**
  - From monoliths to microservices, from VMs to containers
  - Is it all done? No…



Serverless

- **Definition**
  - Serverless is a cloud computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of machine resources.Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity.
  - Generally, Serverless = FaaS + BaaS
- **Pros & Cons**
  - Scale to zero/infinite is possible (Google: “from prototype to production to planet-scale”) → Max. elasticity
  - Cost-effective since there is no fee for idle times. Flip side: infrequently-used code may suffer from greater latency (performance issues)
  - Lack of standards → vendor lock-in

Serverless

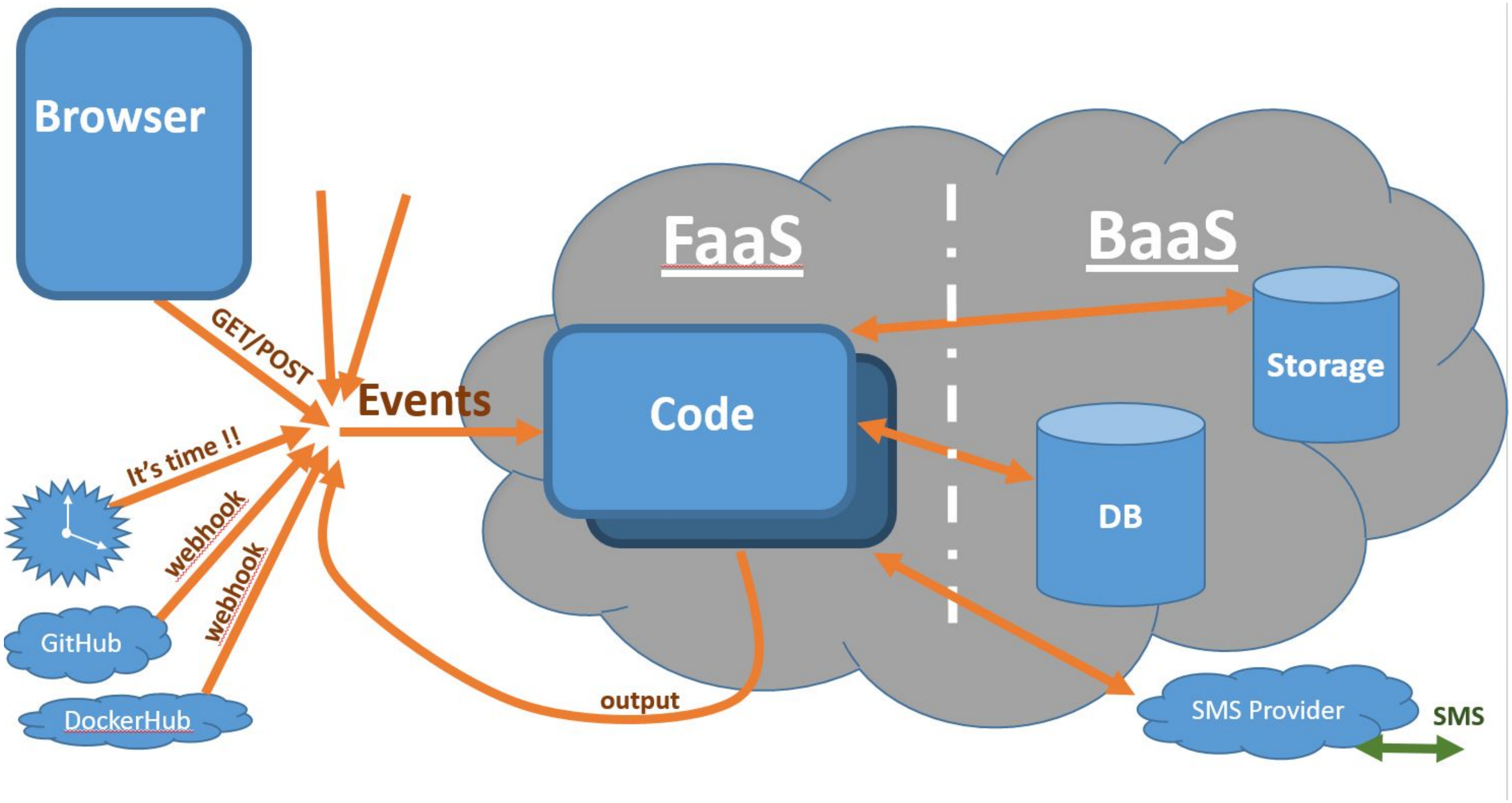
- **FaaS**
  - FaaS is the concept of serverless computing via serverless architectures. Software developers can leverage this to deploy an individual “function”, action, or piece of business logic. They are expected to start within milliseconds and process individual requests and then the process ends.
  - Major providers (see http://serverlesscalc.com/)
    - AWS Lambda (Amazon)
    - Azure Functions (Microsoft)
    - Cloud Functions (Google)
    - IBM Cloud Functions (IBM)
    - Pivotal Function Service (VMware)
  - PaaS vs. FaaS
    - PaaS simplifies dev./deployment process of applications and they run on server like a typical app. once they deployed
    - FaaS provides the ability to deploy a single function (part of application) and **scale to zero** is possible since the rest is managed by the provider

Serverless

- **BaaS (or MBaaS)**
  - BaaS is a model for providing web/mobile app developers with a way to link their applications to backend cloud storage and APIs exposed by backend applications while also providing features such as user management, push notifications, and integration with some services (via SDK/API).
  - Major providers
    - AWS Amplify (Amazon)
    - Firebase (Google)
    - Azure Mobile Apps (Microsoft & Xamarin)
    - Mobile Cloud Service (Oracle)
    - Mobile Application Platform (Red Hat)
  - In basic terms, BaaS/MBaaS is the use of 3rd party services/applications (in the cloud) to handle the server-side logic and state.

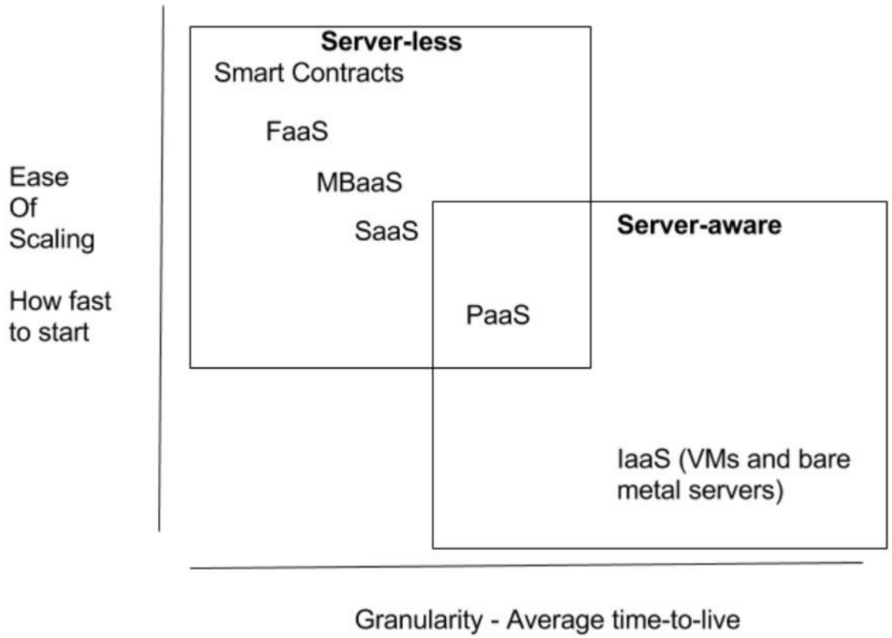
Serverless

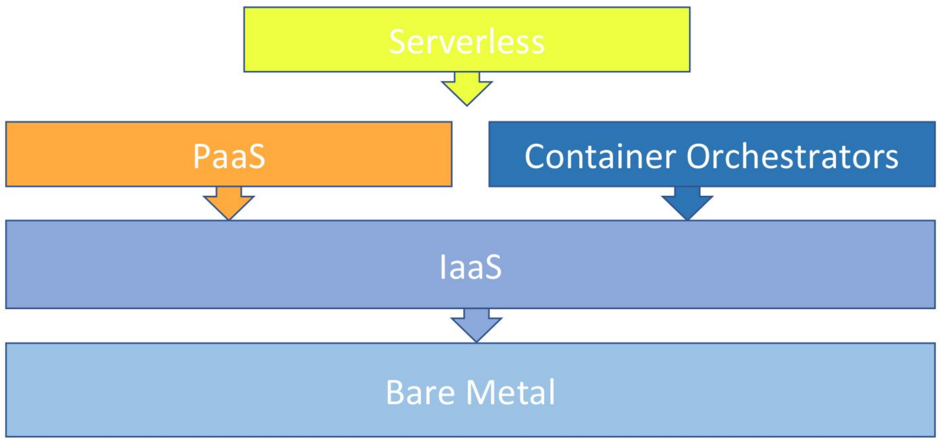
- **Event-driven Architecture**



Serverless

- **Use cases**
  - Serverless platforms are for short-running, stateless computation and event-driven applications which scales up and down instantly and automatically.
  - General characteristics
    - latency tolerant, event-driven, short-lived, periodic
    - simply, where it doesn't make sense to pay for always-on services





Serverless

- **Use cases**

good for

short-running
stateless
event-driven

Microservices

Mobile Backends

Bots, ML Inferencing

IoT

Modest Stream Processing

Service integration

not good for

long-running
stateful
number crunching

Databases

Deep Learning Training

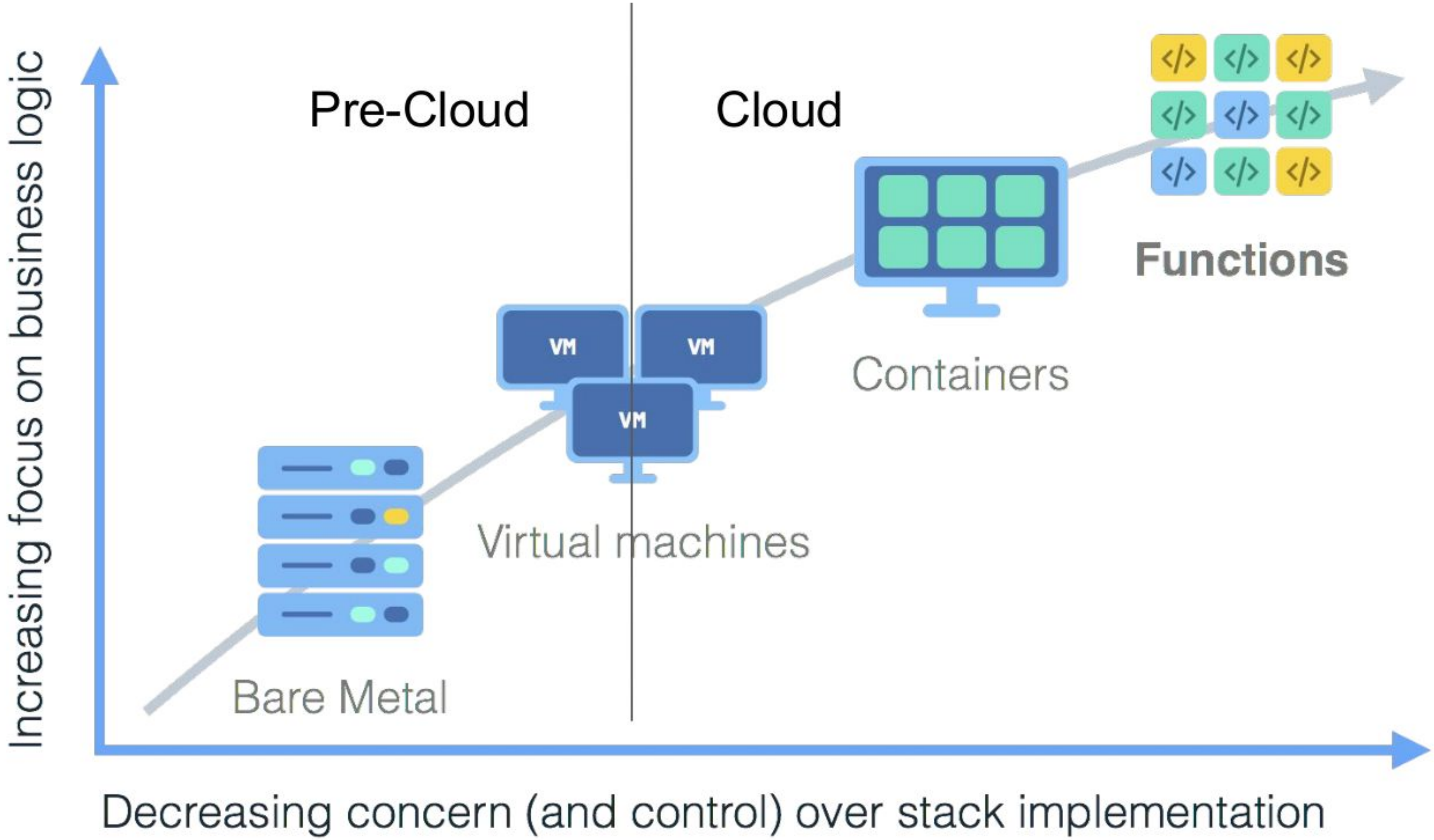
Heavy-Duty Stream Analytics

f(x) Numerical Simulation

Video Streaming

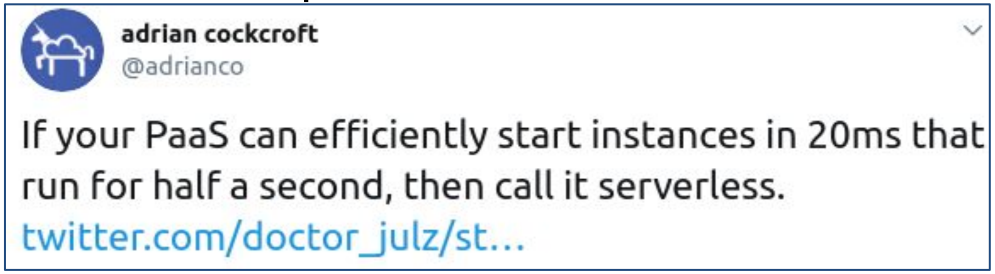
Serverless

- **Focus: Business logic vs. Tech stack**



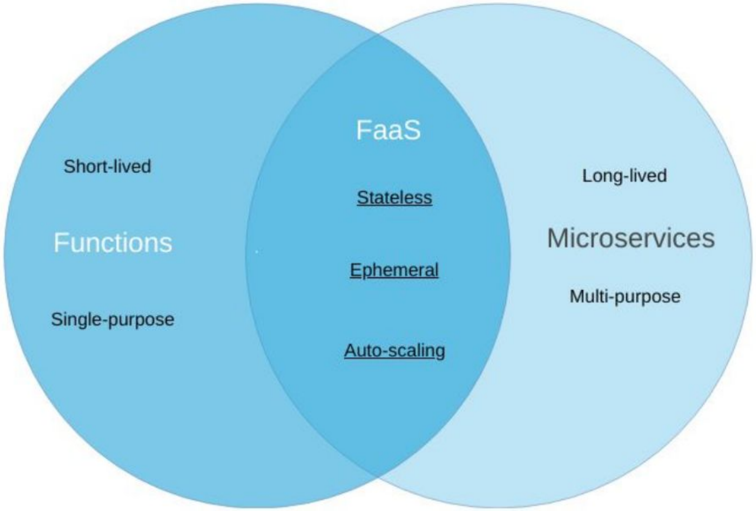
Serverless

- **Recap**
  - Beware hype & marketing, it's not a magic (like anything else)
    - In the end, the container is created and destroyed by algorithms used in FaaS platforms and the operational team have no control over that.



(link)

  - It is not a general purpose solution
    - FaaS & Microservices will co-exist
    - Not for everyone (see use cases)
  - Standardization is important
    - CNCF Serverless Work Group
      - https://github.com/cncf/wg-serverless/
      - Meeting minutes



Q/A