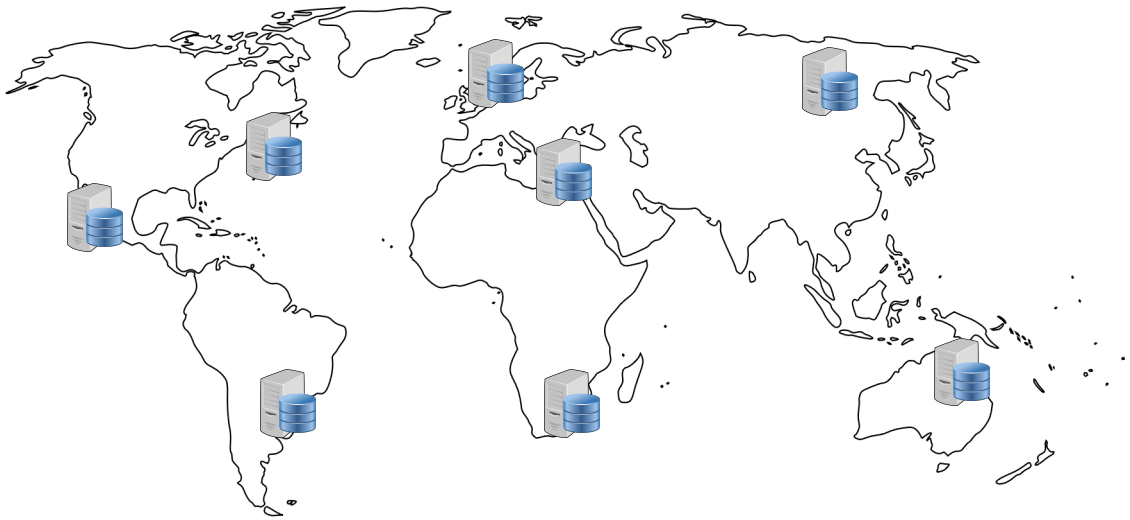
Software Architecting for Distribution

IN4315 Software Architecture



Burcu Kulahcioglu Ozkan

b.ozkan@tudelft.nl
https://burcuku.github.io/home/

TU Delft

Outline & Objectives

Why distribute?

Evolution of architectures

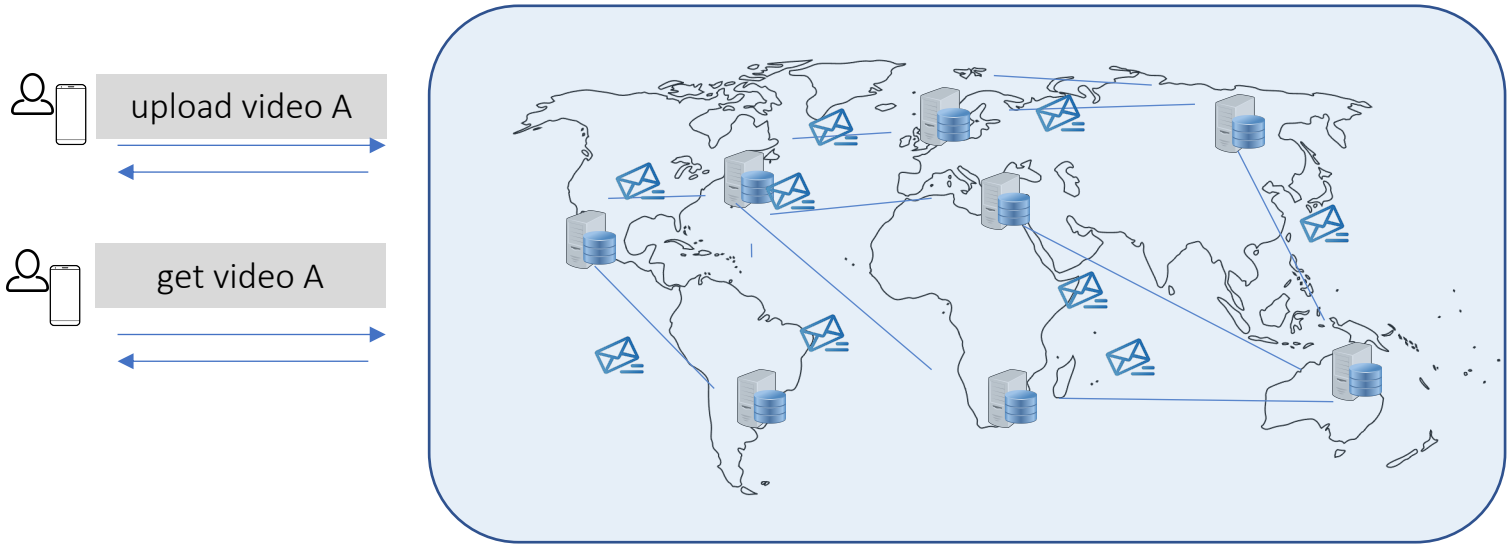
How to replicate data?

Trade-offs

What is a distributed application?

"A **distributed system** is collection of independent computers that appears to its users as a single coherent system" [Tanenbaum and van Steen, 2007]

"A **distributed application** is an application that solves a large problem by breaking it down into several tasks where each task is computed in the individual computers of the distributed system"

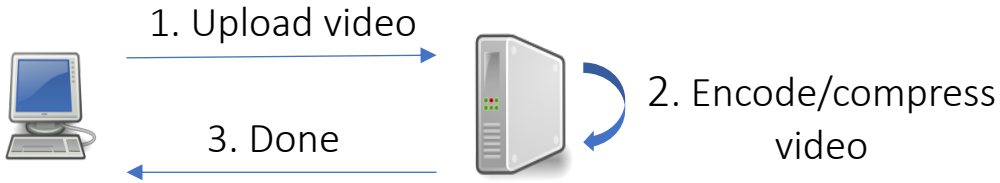


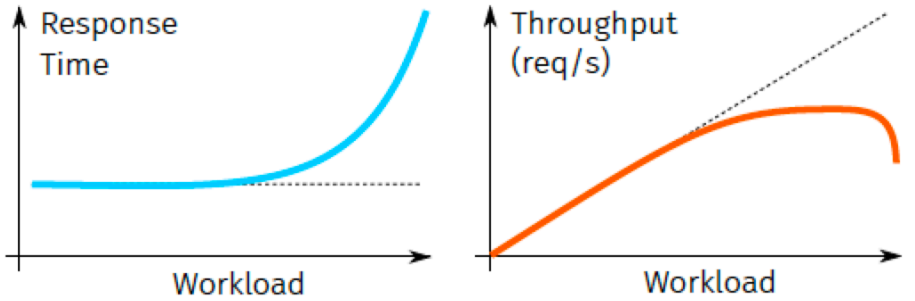
Why do we distribute?

- Performance
- Scalability
- Availability
- Fault tolerance

Why do we distribute? – Performance & scalability

- Performance
  - **Responsiveness**: how long it takes an application to respond to a request
  - **Throughput**: the number of transactions successfully executed per second



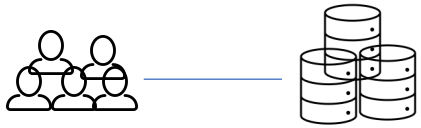


- Scalability
  - Ability to handle increase in workload

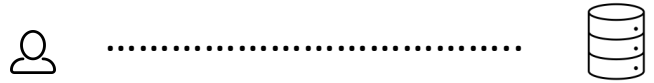
Image from: “Software Architecture”, Cesare Pautasso, 2020

Why do we distribute? – Scalability

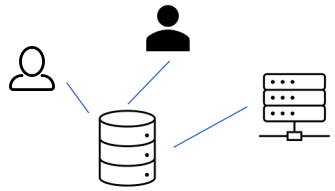
- Size scalability
  - No performance degradation when added more users or resources



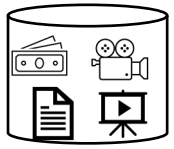
- Geographical scalability
  - No performance degradation when clients and resources may lie far apart



- Administrative scalability
  - Increasing number of organizations/users to easily share the system



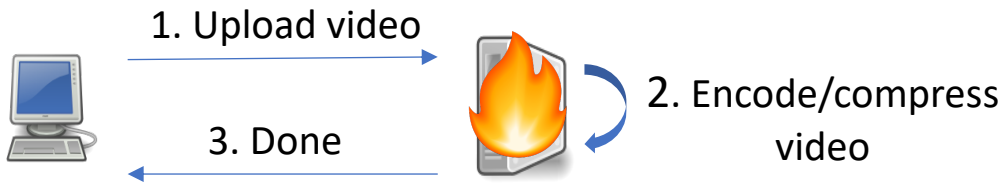
- Functional scalability
  - New features can be added easily without disrupting existing ones

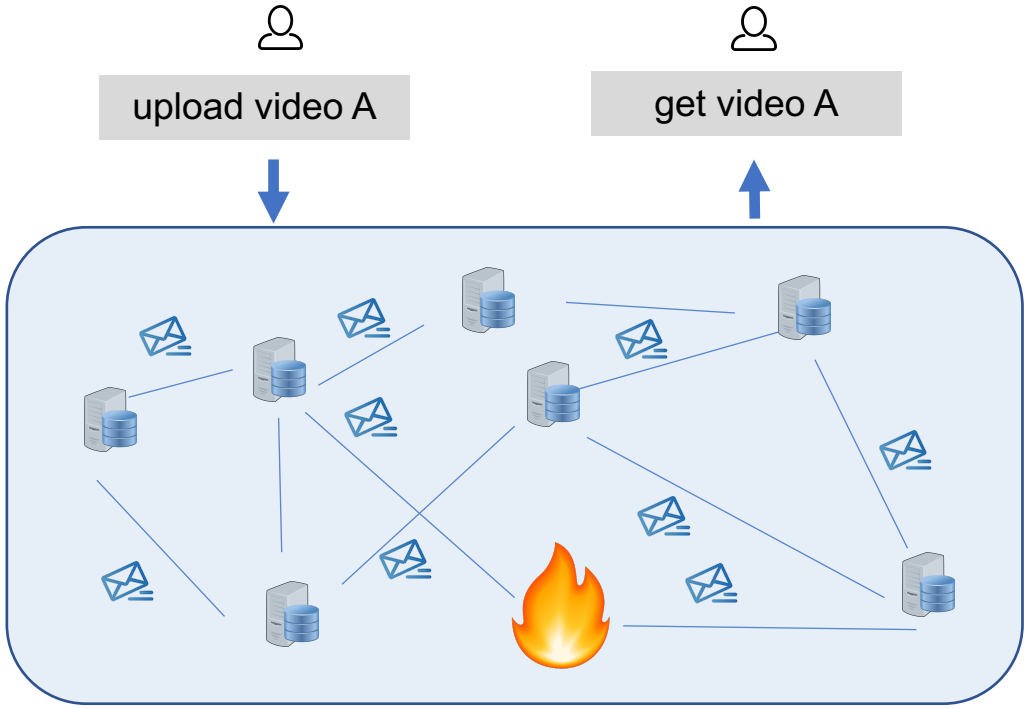


Why do we distribute? – Availability & fault tolerance

Availability:

- Is the system running?
- Is the system accessible?





Faults due to a variety of factors:

- Hardware failure
- Software bugs
- Network errors/outages

A recent availability incident (*Cloudflare API, November'20*)

An analysis of the Cloudflare API availability incident on 2020-11-02

When we review design documents at Cloudflare, we are always on the lookout for Single Points of Failure (SPOFs). Eliminating these is a necessary step in architecting a system you can be confident in. Ironically, when you're designing a system with built-in redundancy, you spend most of your time thinking about how well it functions when that redundancy is lost.

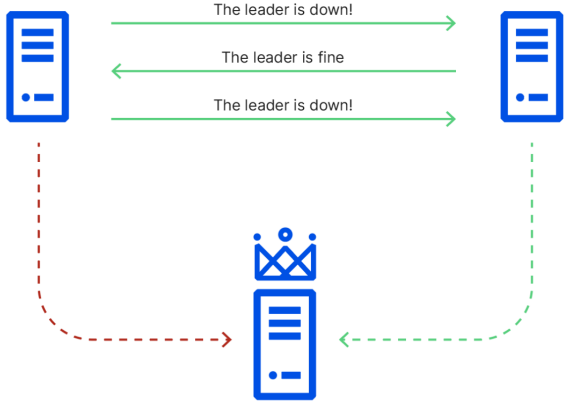
On November 2, 2020, Cloudflare had an incident that impacted the availability of the API and dashboard for six hours and 33 minutes. During this incident, the success rate for queries to our API periodically dipped as low as 75%, and the dashboard experience was as much as 80 times slower than normal. While Cloudflare's edge is massively distributed across the world (and kept working without a hitch), Cloudflare's control plane (API & dashboard) is made up of a large number of microservices that are redundant across two regions. For most services, the databases backing those microservices are only writable in one region at a time.

and many more…

Microsoft Azure, March 3

It was an initial early March six-hour outage that struck the US East data center for Microsoft's Azure cloud, limiting the availability of Azure cloud services for some North American customers. Microsoft then disclosed that a cooling system failure was the cause of the outage. Malfunctioning building automation controls caused a reduction in airflow, and the subsequent temperature spikes throughout the data center hampered the performance of network devices, rendering compute and storage instances inaccessible.

Microsoft ultimately reset the cooling system controllers, and once the temperature fell, engineers power-cycled hardware to resume services.



Summary of the Amazon Kinesis Event in the Northern Virginia (US-EAST-1) Region

November, 25th 2020

We wanted to provide you with some additional information about the service disruption that occurred in the Northern Virginia (US-EAST-1) Region on November 25th, 2020.

Amazon Kinesis enables real-time processing of streaming data. In addition to its direct use by customers, Kinesis is used by several other AWS services. These services also saw impact during the event. The trigger, though not root cause, for the event was a relatively small addition of capacity that began to be added to the service at 2:44 AM PST, finishing at 3:47 AM PST. Kinesis has a large number of "back-end" cell-clusters that process streams. These are the workhorses in Kinesis, providing distribution, access, and scalability for stream processing. Streams are spread across the back-end through a sharding mechanism owned by a "front-end" fleet of servers. A back-end cluster owns many shards and provides a consistent scaling unit and fault-isolation. The front-end's job is small but important. It handles authentication, throttling, and request-routing to the correct stream-shards on the back-end clusters.

The capacity addition was being made to the front-end fleet. Each server in the front-end fleet maintains a cache of information, including membership details and shard ownership for

Sources:
https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/
https://status.cloud.google.com/incident/zall/20013
https://aws.amazon.com/message/11201/

Outline & Objectives

Why distribute?

Evolution of architectures

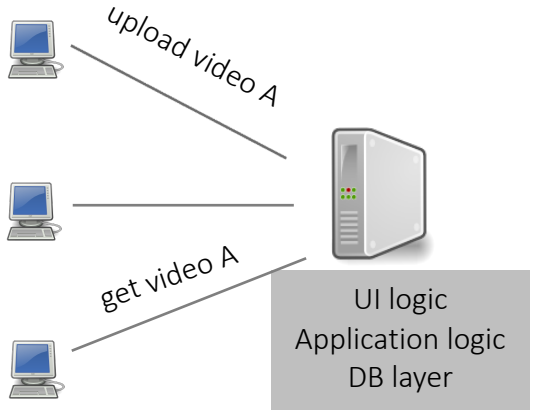
How to replicate data?

Trade-offs

- Scalability
- Performance
- Availability
- Fault-tolerance

Evolution of distributed applications – Monolithic architectures

- Early days of internet, **monolith applications**: software is developed as a single unit
- Components are interdependent in the code level
- Centralized server architecture: multiple clients share the same server

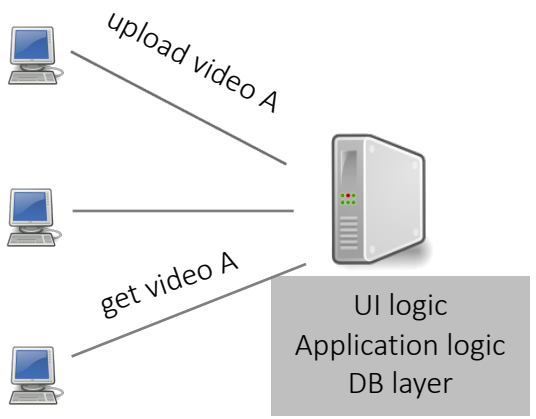


- Suitable for small teams, small projects, start-ups
- Simpler development and deployment

- Not fault tolerant
  - Single point of failure
- Not scalable
  - Increasing number of client requests?
  - Increasing complexity of the application?

Evolution of distributed applications – Monolithic architectures

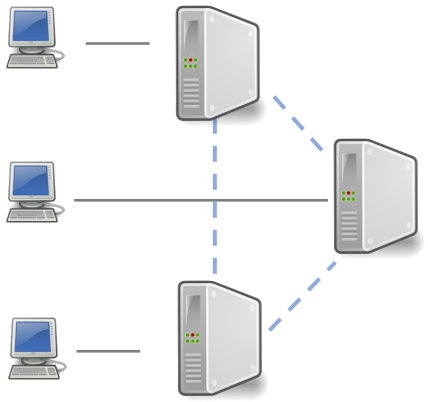
- Scale up - Vertical scaling: Increase CPU power, memory & disk space



- Not fault tolerant
  - Single point of failure
- Limits to scalability
  - The computational capacity, limited by the CPUs
  - The storage capacity, including the transfer rate between CPUs and disks
  - The network between the user and the service

Evolution of distributed applications – Monolithic architectures

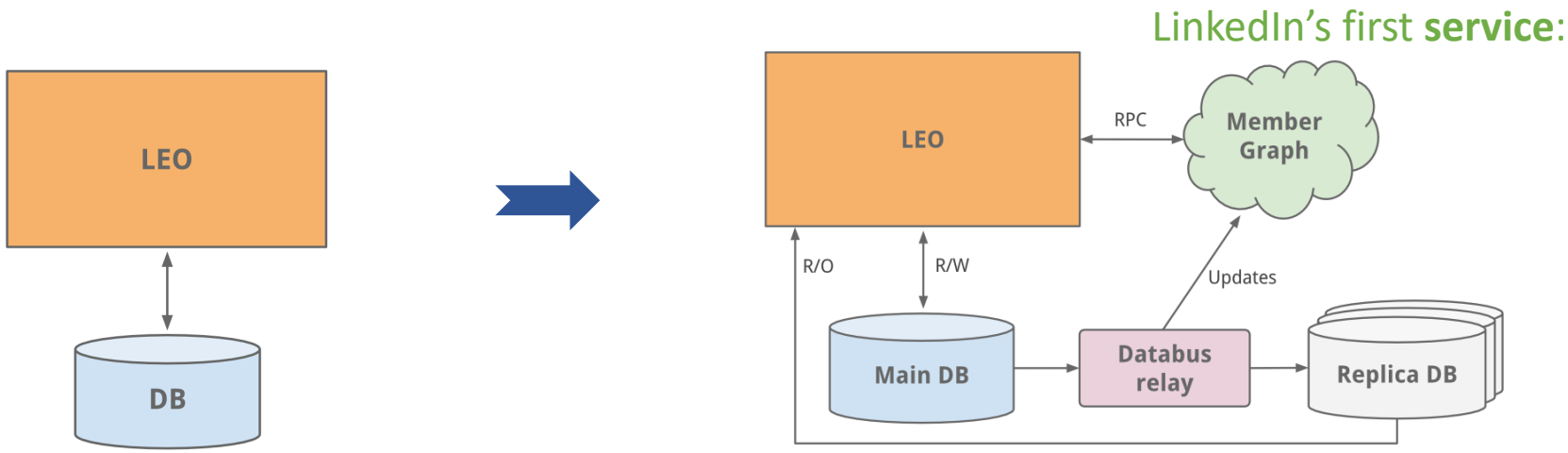
- Scale out - Horizontal scaling: Add more servers, introduce parallelism



- Fault tolerant (by replication)
- Scalable?
  - ✓ Increasing number of client requests?
  - Increasing complexity of the application?

Scaling LinkedIn – LEO Monolith

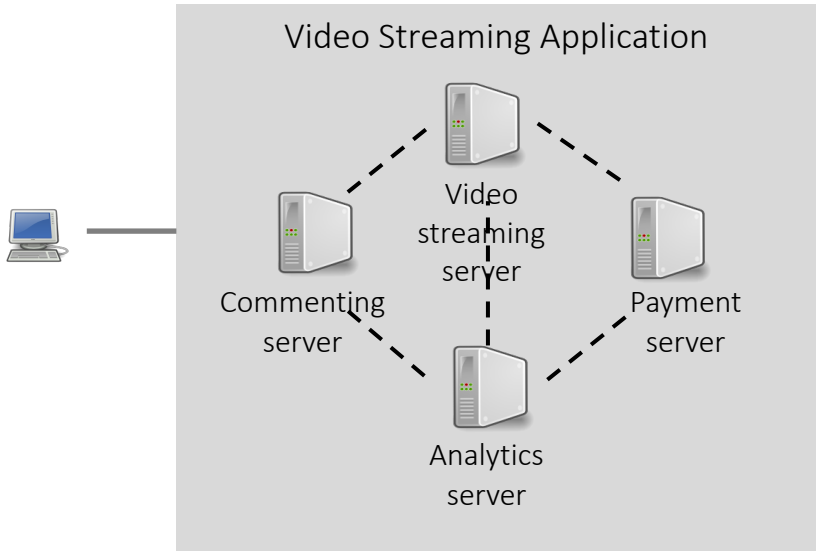
- Started as a monolith in 2003
- A system for querying membership using graph traversals: Member Graph
- Read-only replicas for scalability



https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin

Evolution of distributed systems – Service-oriented architectures (SOA)

- In the early 2000s, SOA emerged as a paradigm for distributed applications
  - Decompose the application into services, split responsibility
  - Design to share resources across services
  - Design interoperable components which communicate by a common API (e.g. SOAP)

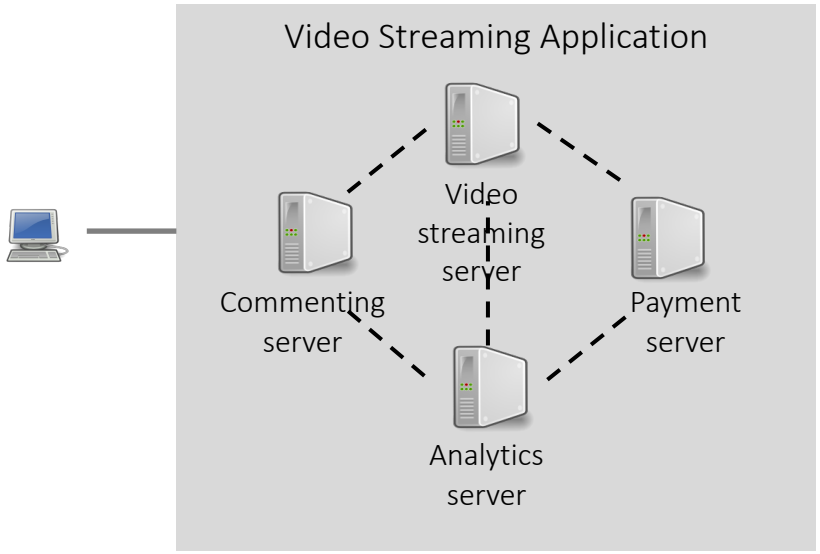


A SOA service:

- Logically represents a business activity with a specified outcome
- Is self-contained
- Is a black-box for its consumers

Evolution of distributed systems – Service-oriented architectures (SOA)

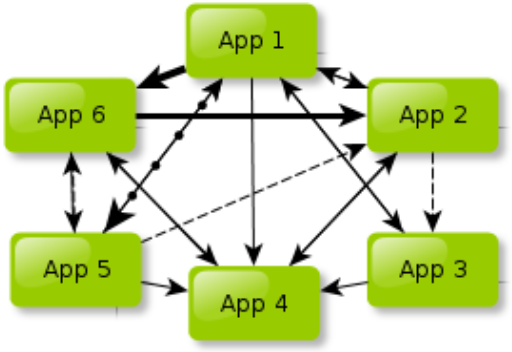
- In the early 2000s, SOA emerged as a paradigm for distributed applications
  - Decompose the application into services, split responsibility
  - Design to share resources across services
  - Design interoperable components which communicate by a common API (e.g. SOAP)

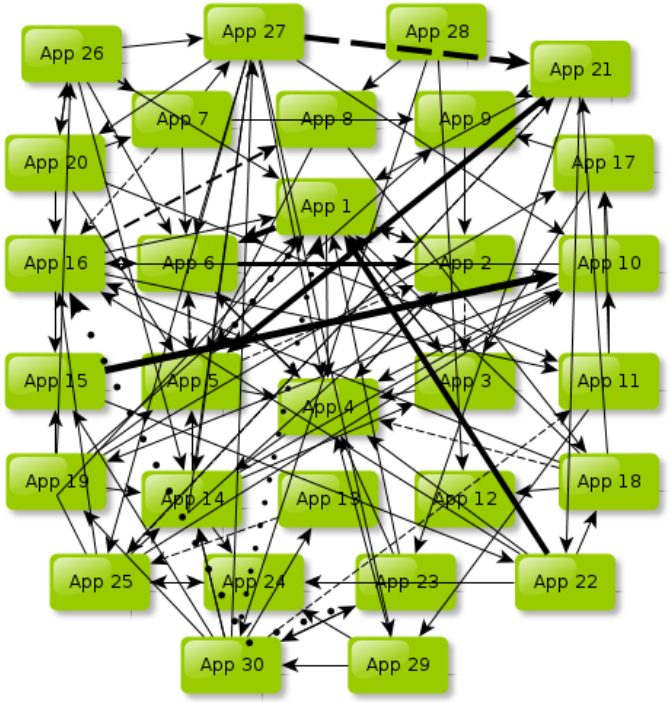


- Scalable
  - ✓ Increasing number of client requests?
  - ✓ Increasing application complexity?
  - ✓ Increasing administration complexity?
- Fault-tolerant
- Reusable
- Modular

Evolution of distributed systems – Service-oriented architectures (SOA)

- Transition from human-oriented interaction to machine-machine interaction
- Challenge: How to communicate the services?
  - **Enterprise service bus (ESB)** - an additional messaging layer removing point-to-point messaging

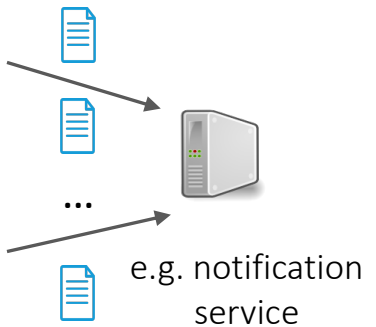




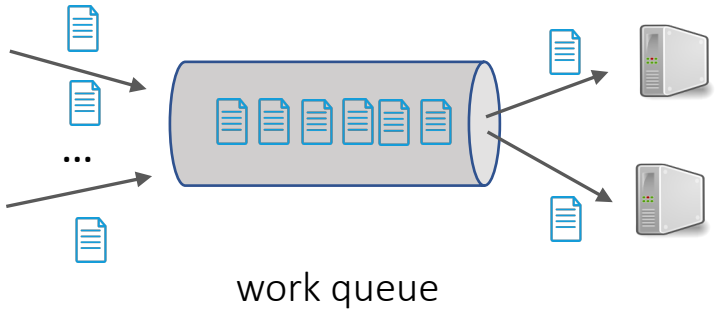
https://dzone.com/articles/apache-kafka-vs-integration-middleware-mq-etl-esb

Message Brokers

- Architectural pattern for application-level message validation, transformation, and routing
- Decouples producers and consumers
- Asynchronous communication & processing



Performance might degrade with intermittent heavy loads



The service will be able to handle the messages at its own pace

Fair dispatch:

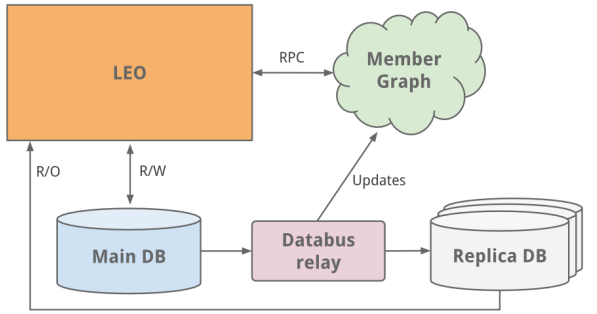
- Send messages to available consumers
- Better distribute workload

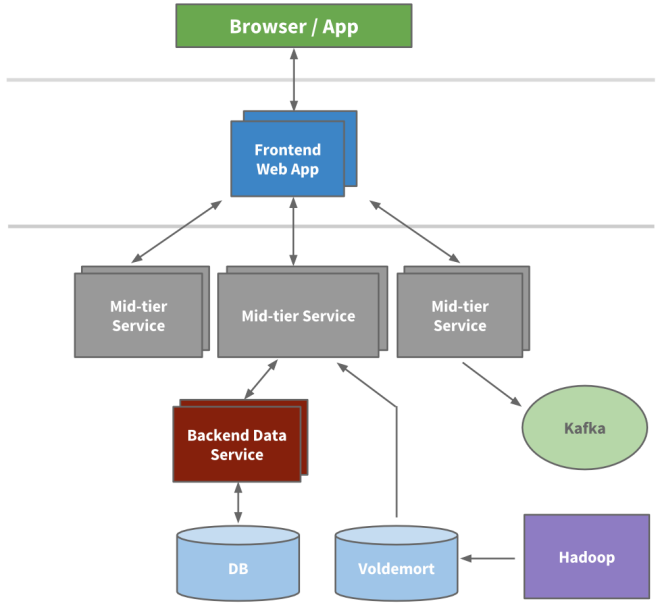
Publish/subscribe:

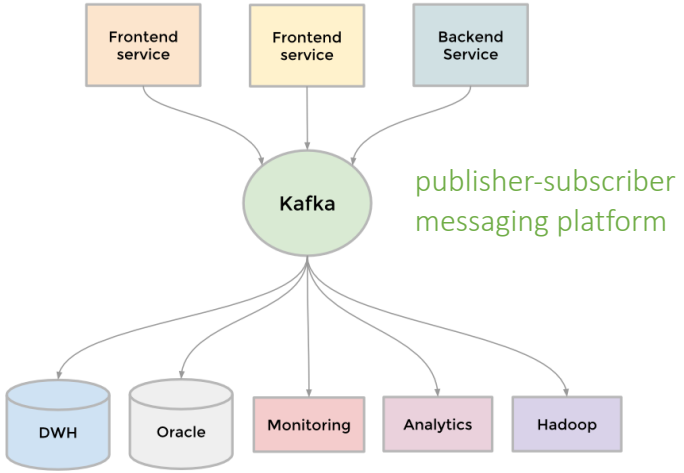
- Deliver one message to multiple consumers
- More common in microservices

Scaling LinkedIn – Service-oriented architecture

- As the site began to get more traffic LEO started going down in production
- Difficult to troubleshoot, recover, release new code
- “Kill Leo monolith” and break it up into small services



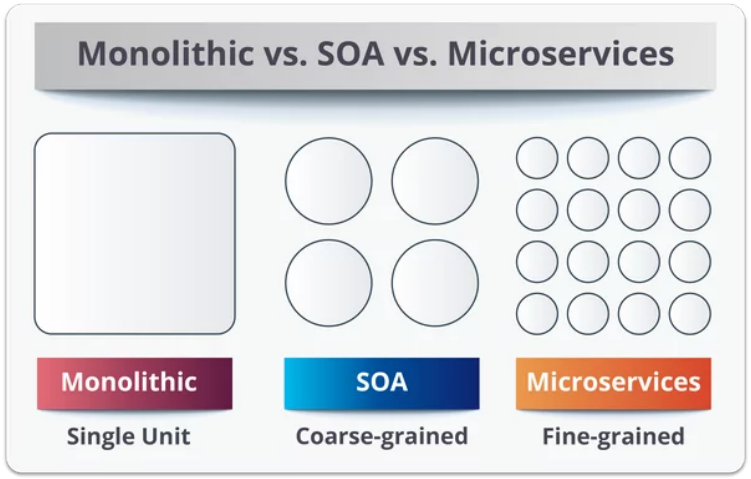


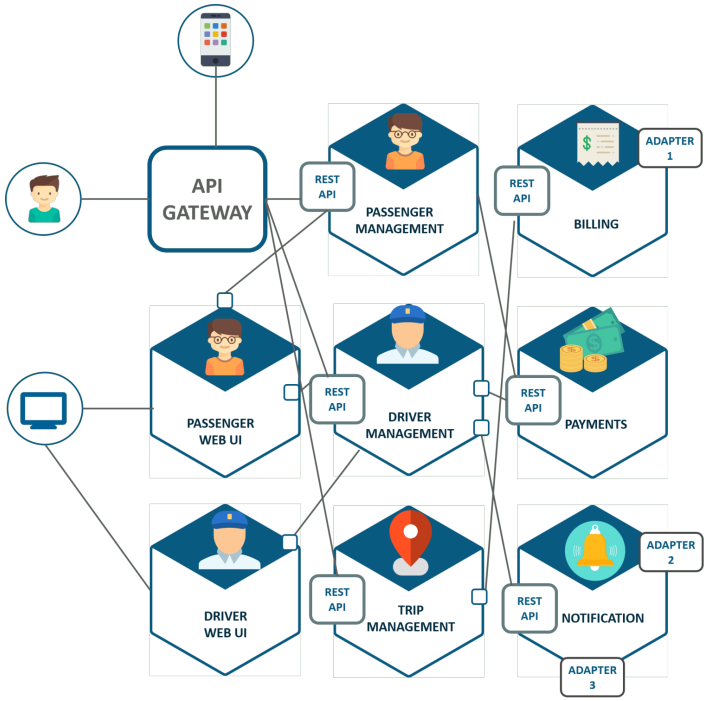


https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin

Evolution of distributed systems – Microservice architectures

- Build an application as a collection of loosely-coupled microservices
- Design to make each functionality separate as a service and a self-contained
- **Microservices are the resulting standalone services** after breaking a software application down into separate components that perform their functions





Microservice Architecture of Uber

https://www.devteam.space/blog/microservices-vs-soa-and-api-comparison
https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a

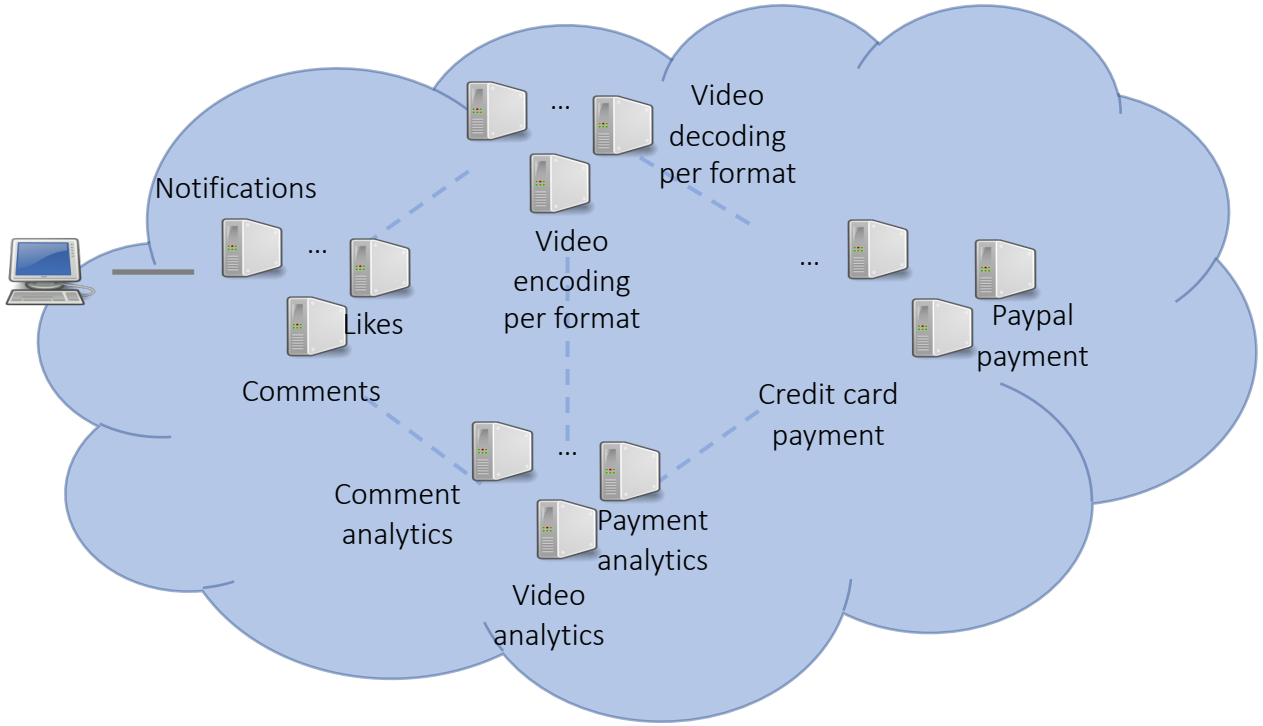
Service-oriented archirecture vs Microservice architecture

SOA	MSA
Follows "**share-as-much-as-possible**" architecture approach	Follows "**share-as-little-as-possible**" architecture approach
Importance is on **business functionality** reuse	Importance is on the concept of "**bounded context**"
They have **common governance** and **standards**	They focus on **people, collaboration** and freedom of other options
Uses **Enterprise Service bus (ESB)** for communication	Simple messaging system
They support **multiple message protocols**	They use **lightweight protocols** such as **HTTP/REST** etc.
Multi-threaded with more overheads to handle I/O	**Single-threaded** usually with the use of Event Loop features for non-locking I/O handling
Maximizes application service reusability	Focuses on **decoupling**
Traditional Relational Databases are more often used	**Modern Databases** are more often used
A systematic change requires modifying the monolith	A systematic change is to create a new service
DevOps / Continuous Delivery is becoming popular, but not yet mainstream	Strong focus on DevOps / Continuous Delivery

https://dzone.com/articles/microservices-vs-soa-whats-the-difference

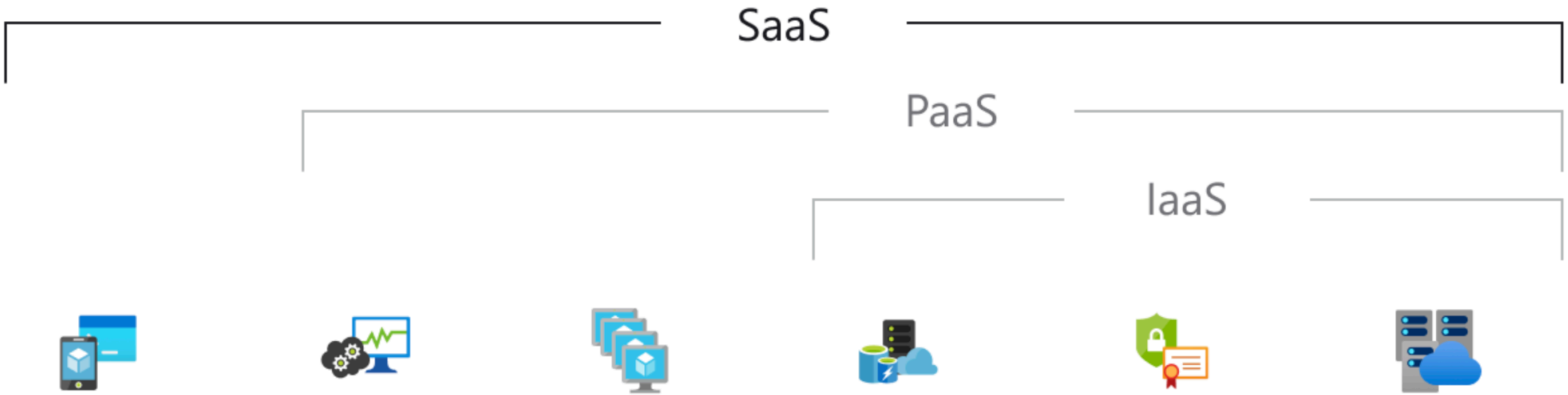
Evolution of distributed systems – Microservice architectures

- Microservices are an architectural approach to creating cloud applications
- Microservices in the cloud: Software-as-a-service
  - Hosted on a remote server, accessible over the internet
  - Users are not responsible for hardware or software updates



- Scalable
- Fault-tolerant
- Available

Cloud services and applications

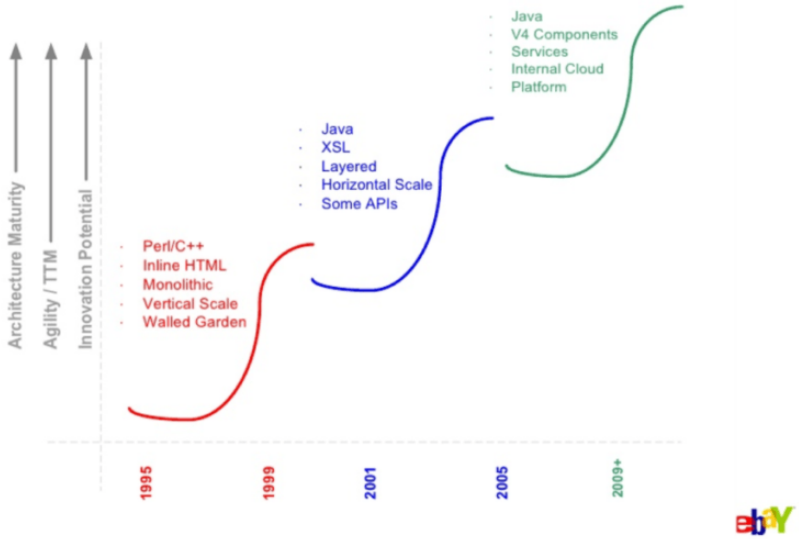


Hosted applications/apps	Development tools, database management, business analytics	Operating systems	Servers and storage	Networking firewalls/security	Data center physical plant/building

← Less effort to manage More levels of control →

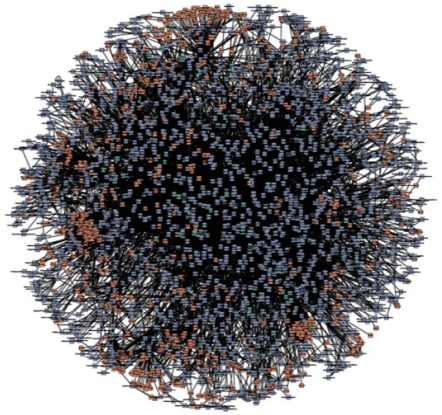
Image from: https://azure.microsoft.com

Modern systems move towards more decentralization



LinkedIn's operational setup as of 2015

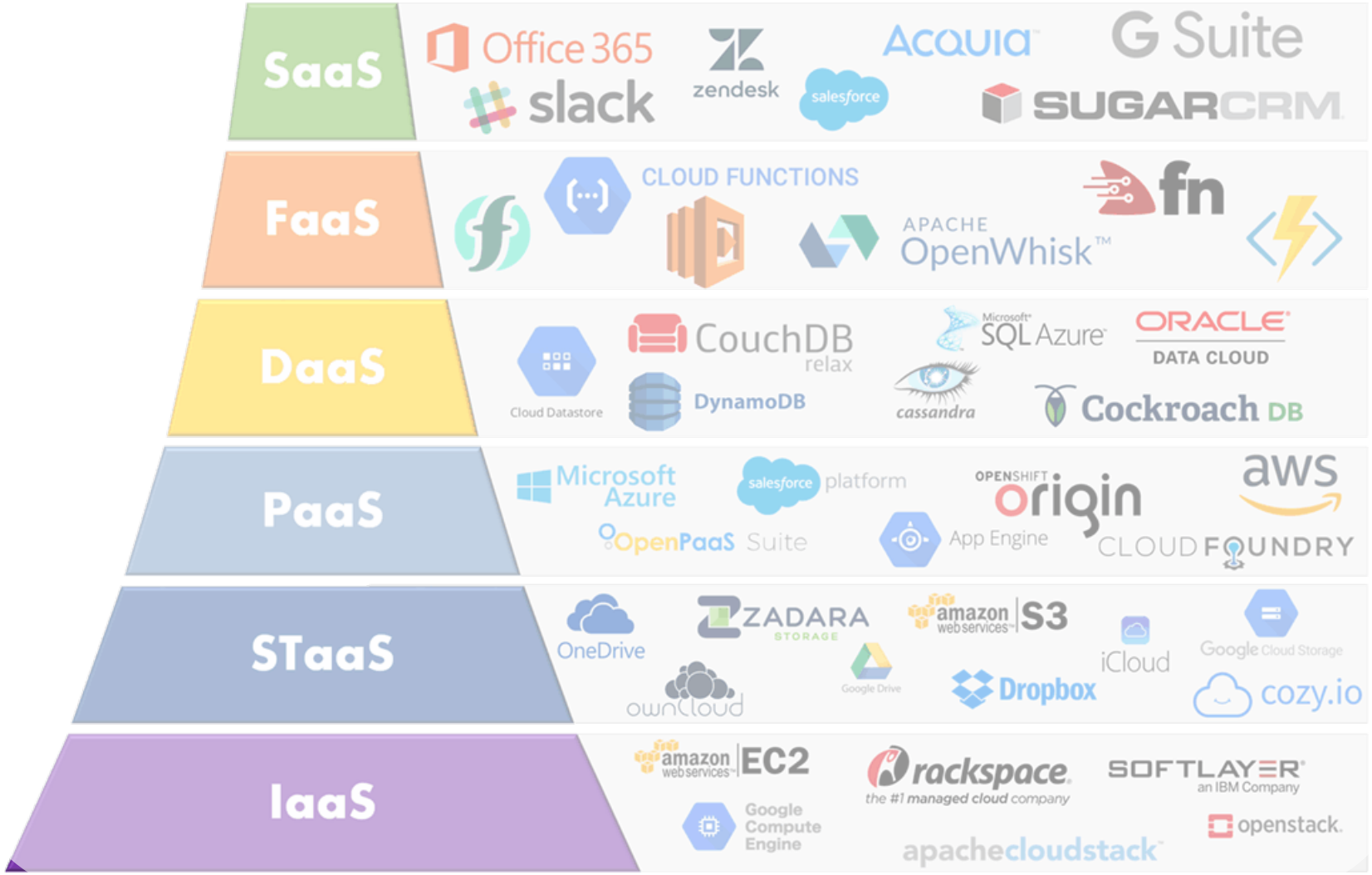
"If you go back to 2001, the Amazon.com retail website was a large architectural monolith"
Rob Brigham, Amazon AWS senior manager



Current structure of microservices at Amazon

But still... One approach does not fit all.

Cloud services and applications



As software developers,

we should know

what guarantees / choices

are provided in the services

we build our applications on.

Image from: imelgrat.me

Outline & Objectives

Why distribute?

- Scalability
- Performance
- Availability
- Fault-tolerance

Evolution of architectures

From centralized to decentralized

- Monolithic server
- Service oriented
- Microservices

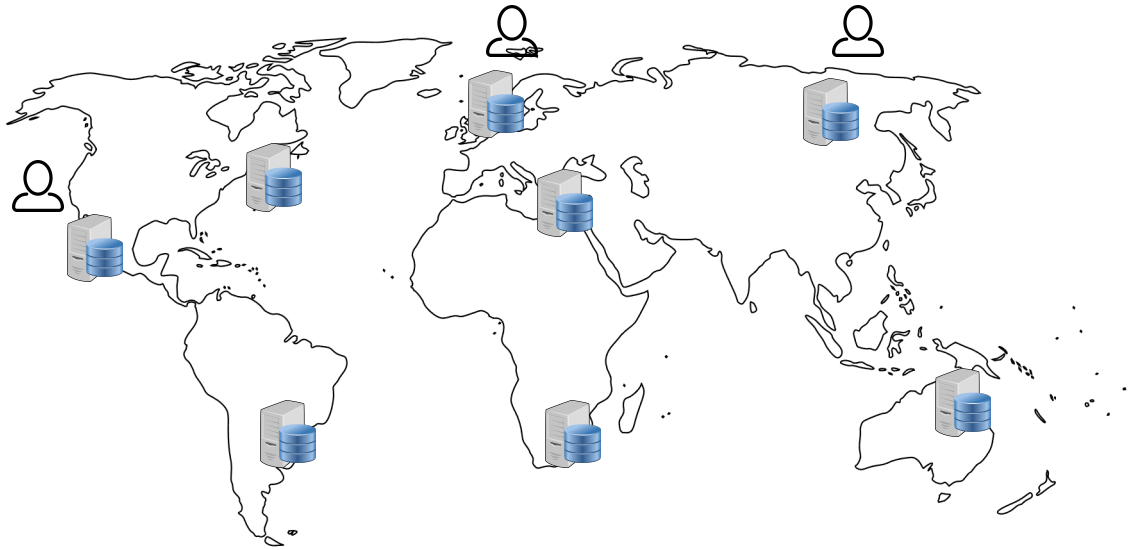
(Cloud-based services)

How to replicate data?

Trade-offs

Why replicate data?

- Size scalability
- Geographical scalability
- Availability
- Fault tolerance

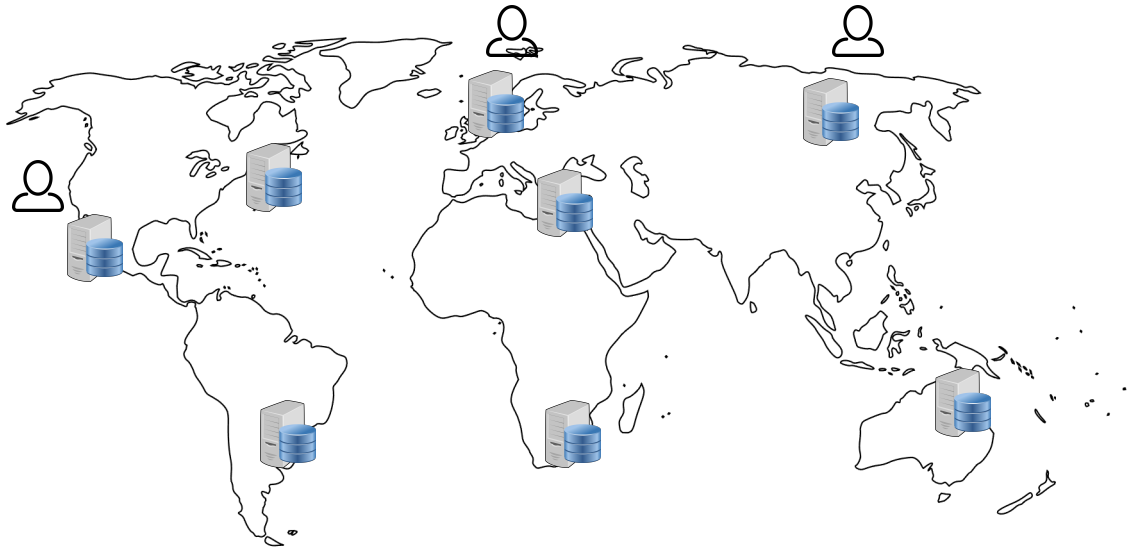


What are the cons/challenges?

- Costly
  - Computational resources
- CAP impossibility
  - Mainly between availability and consistency

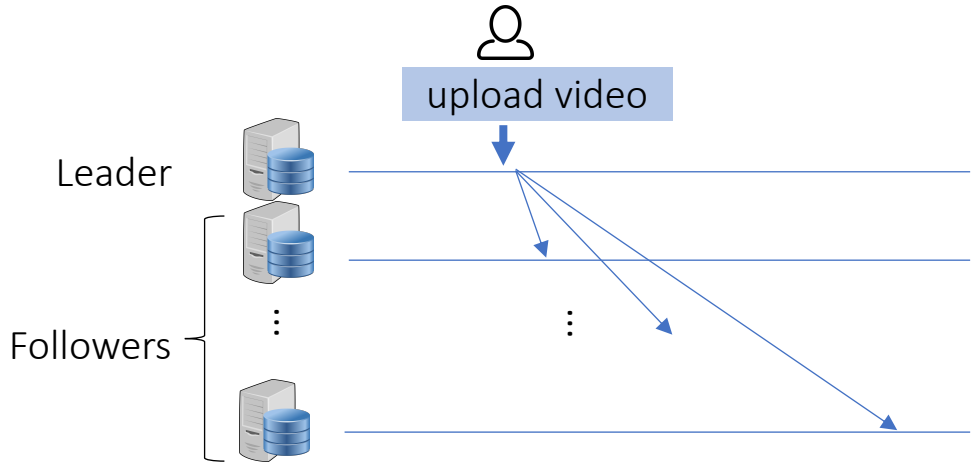
How to replicate data?

- Replication of stateless components or read-only data?
- Replication of stateful components or mutable data?
  - Single-leader
  - Multi-leader
  - Leaderless



Leader-based replication

- One of the replicas is the leader (or primary copy), the others as followers (or secondary copies)
- Write queries are only accepted on the leader, and sent to followers
- Clients can submit read queries to the leader or any of the followers

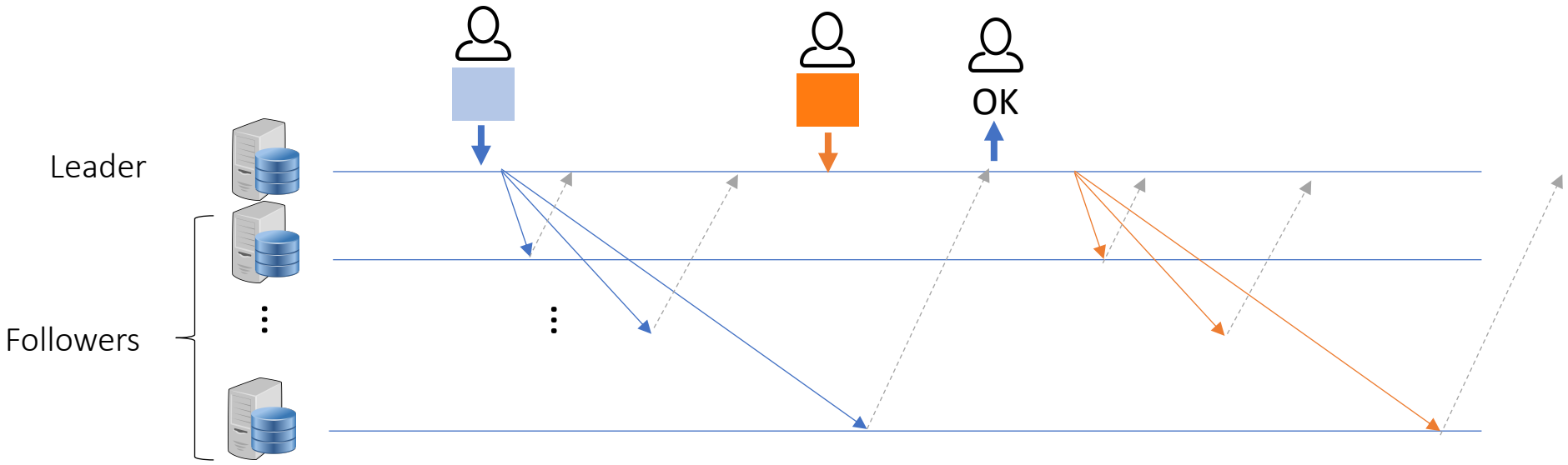


How to do the replication to followers?

- Synchronously?
- Asynchronously?

Leader-based replication - Synchronous

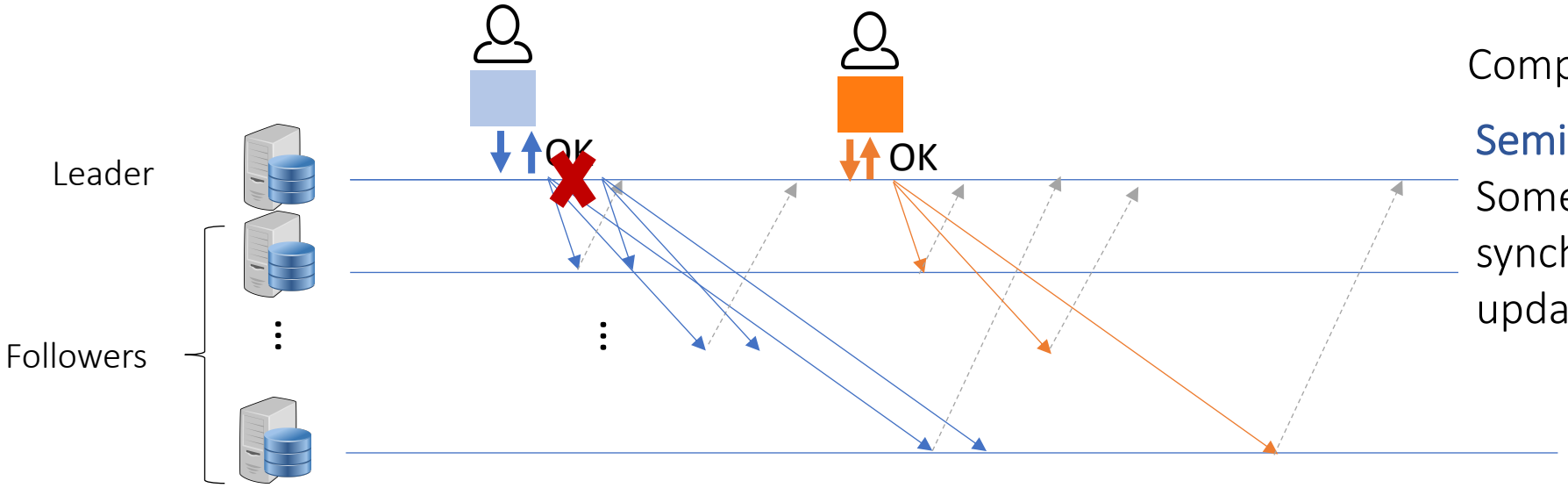
- The leader waits until the followers receive the update and before reporting success
  - ✓ A follower is has up-to-date copy
  - ✓ If the leader fails, data is still available on the follower
  - Writes are blocked if a follower is not available



impractical for all followers to be synchronous

Leader-based replication - Asynchronous

- The leader reports success and asynchronously updates the followers
  - ✓ Writes are not blocked in case of inaccessible follower
  - A follower is not guaranteed to have an up-to-date copy of the data
  - Writes are not guaranteed to be durable in case of leader failure

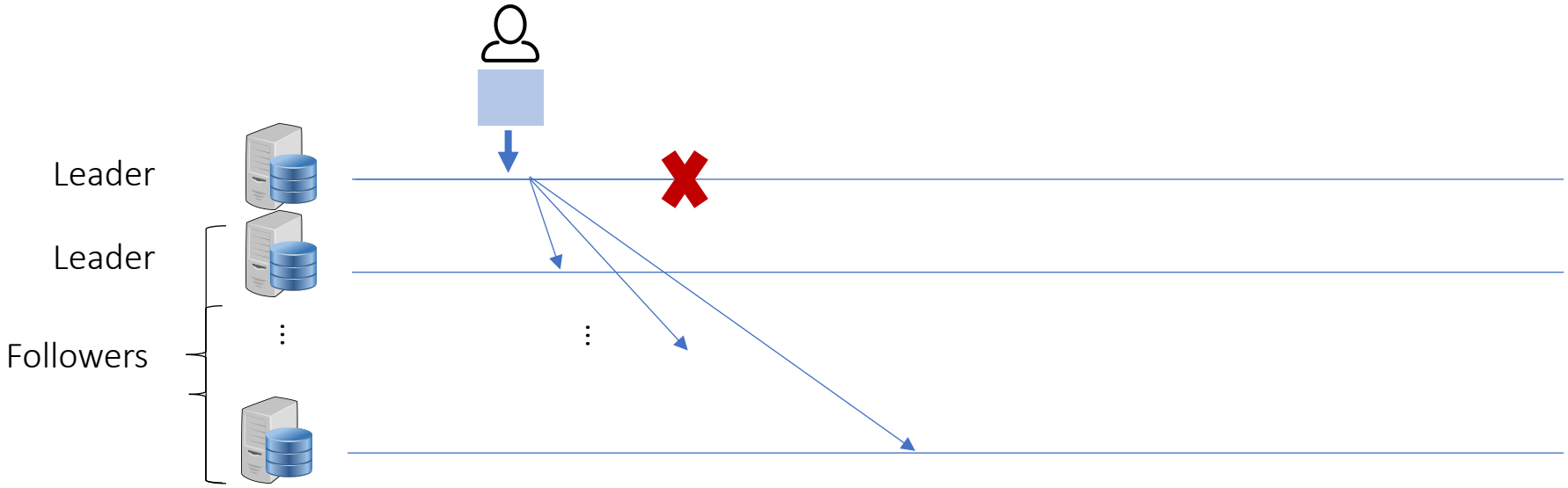


Compromise between two models:

Semi-synchronous:
Some followers are updated synchronously some are updated asynchronously

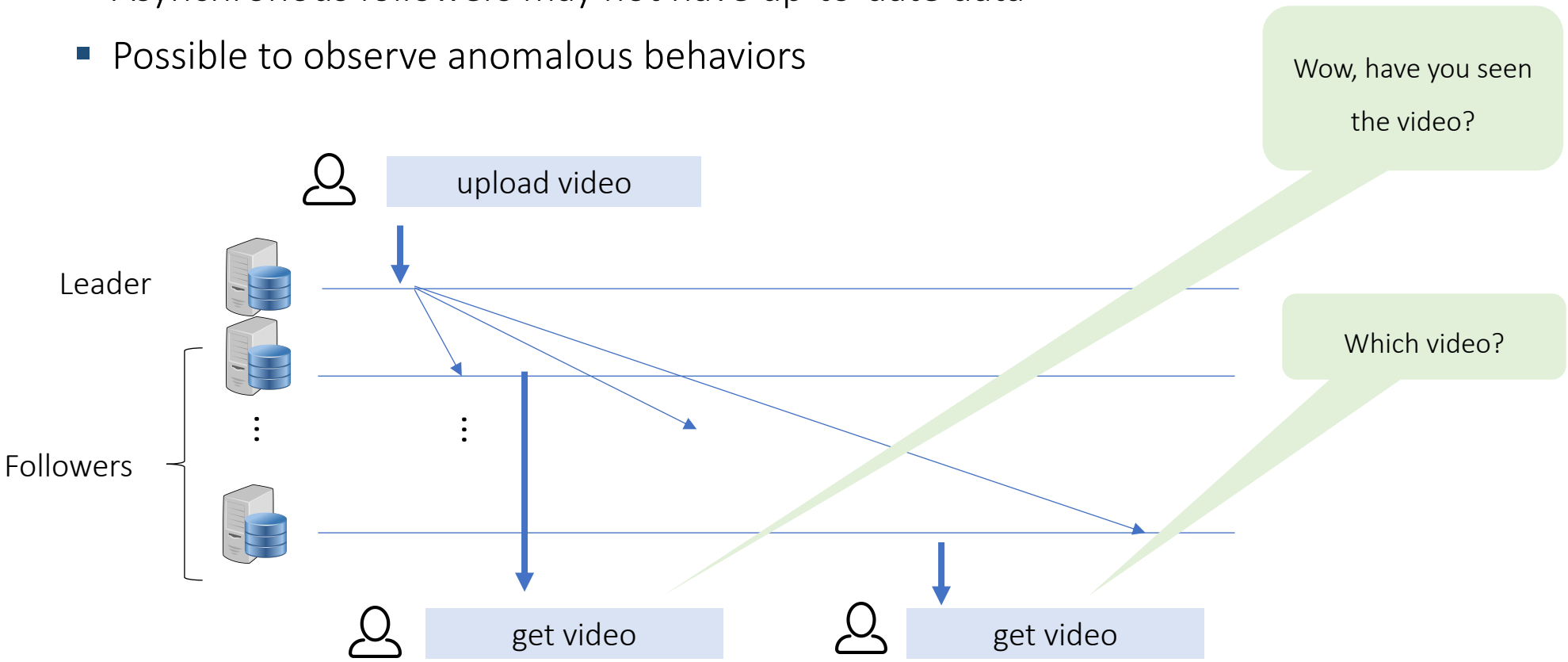
Leader-based replication – Failure Scenarios

- How to set up a new follower?
- How to handle component failures?
  - Follower failure: Catch-up recovery
  - Leader failure: Failover



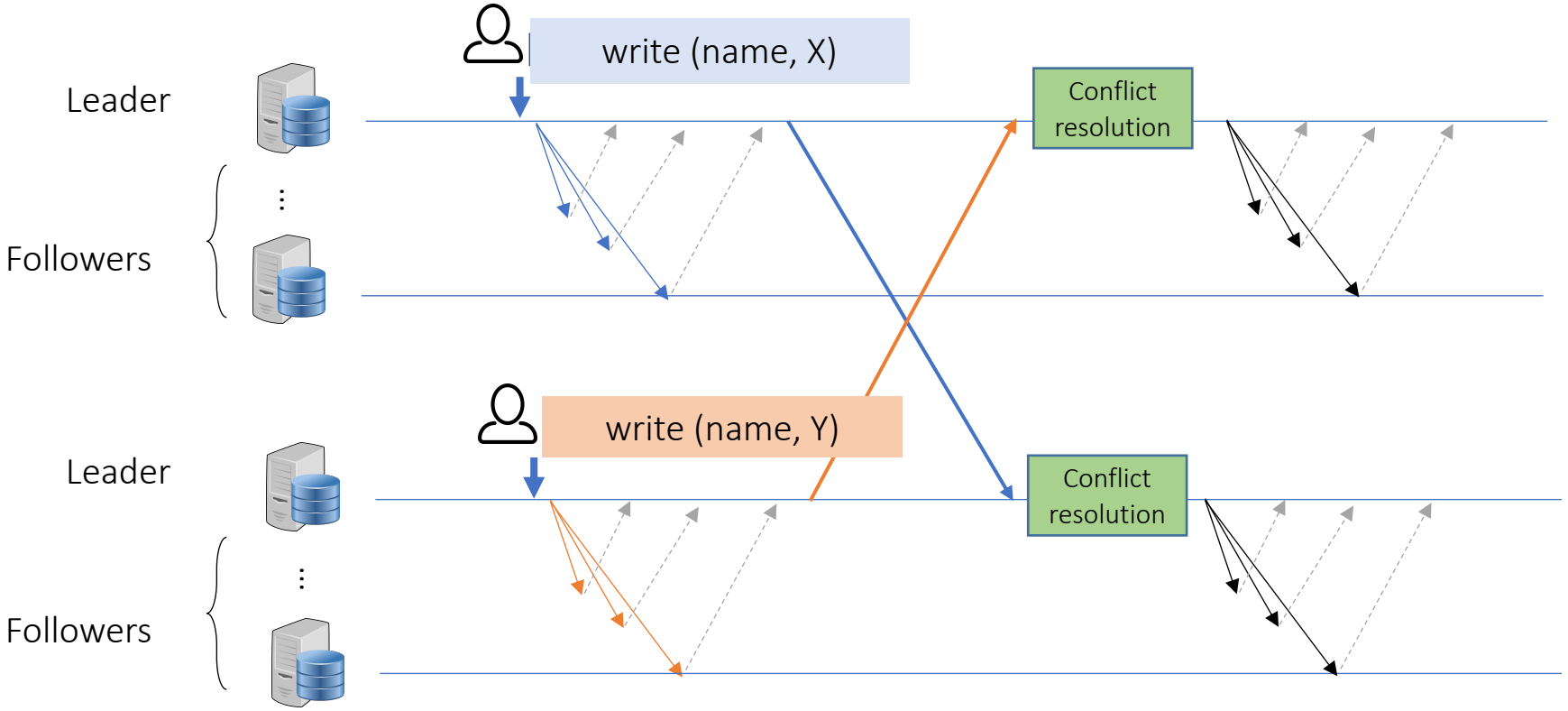
Leader-based replication – Asynchronous

- Asynchronous followers may not have up-to-date data
- Possible to observe anomalous behaviors



Multi-leader replication – Conflicting updates

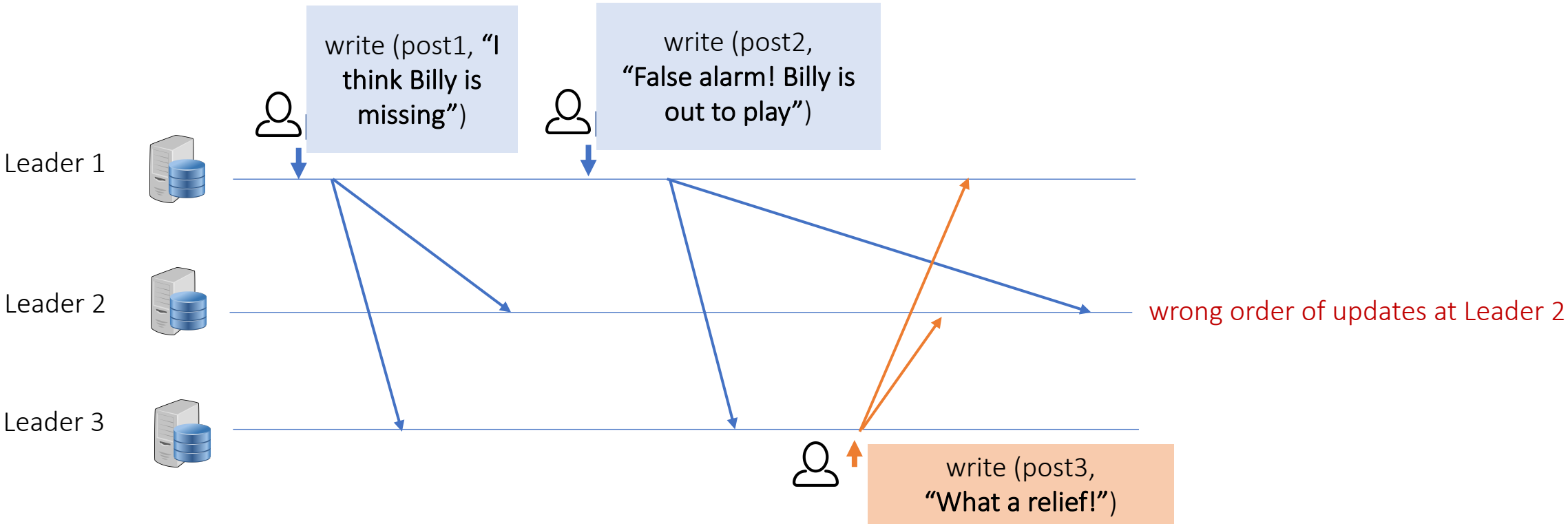
- Multiple leader nodes to accept writes
- Replication to followers in a similar way to single-leader case



Conflict resolution decides on the final value of “name”

Multi-leader replication – Ordering problems

- Writes may arrive in the wrong order to some replicas

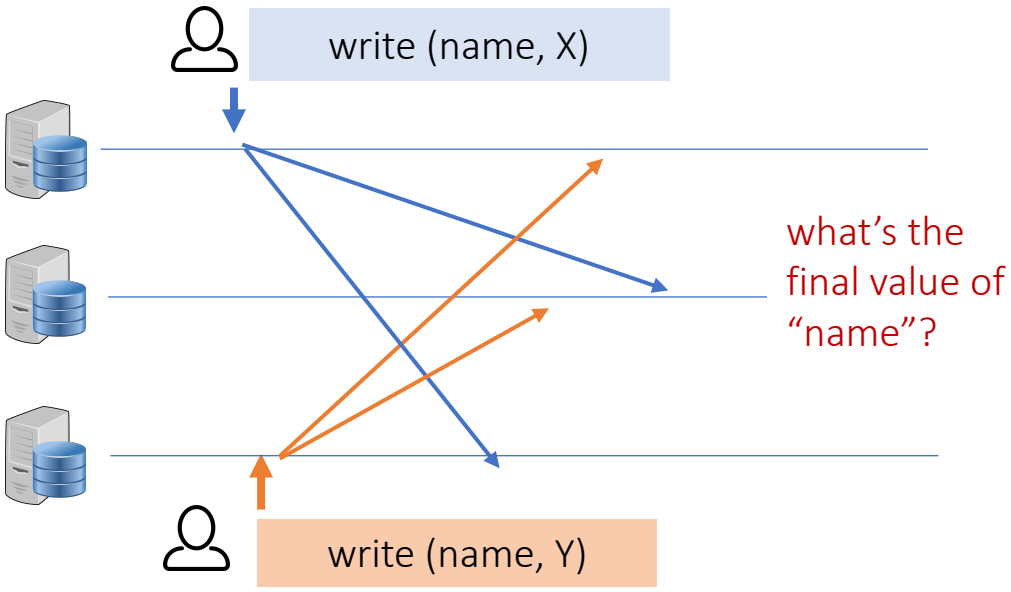


Example scenario from: "Bolt-on Causal Consistency", SIGMOD'13

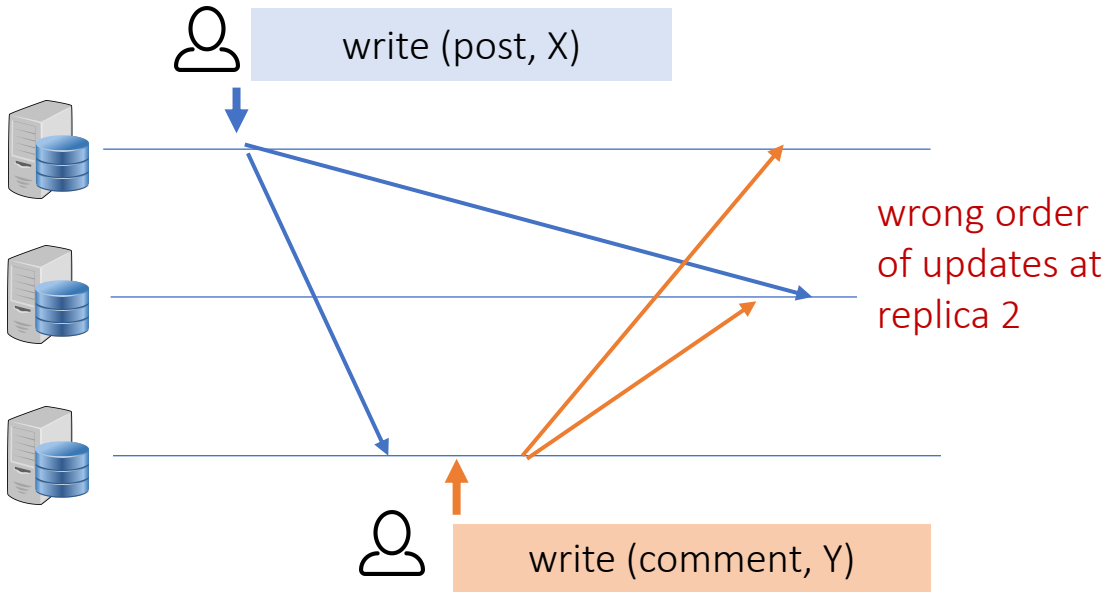
Leaderless Replication – Asynchronous reads/writes

- No leader – any replica can directly accept writes from clients
- Asynchronous replication can cause:

Conflicting concurrent updates



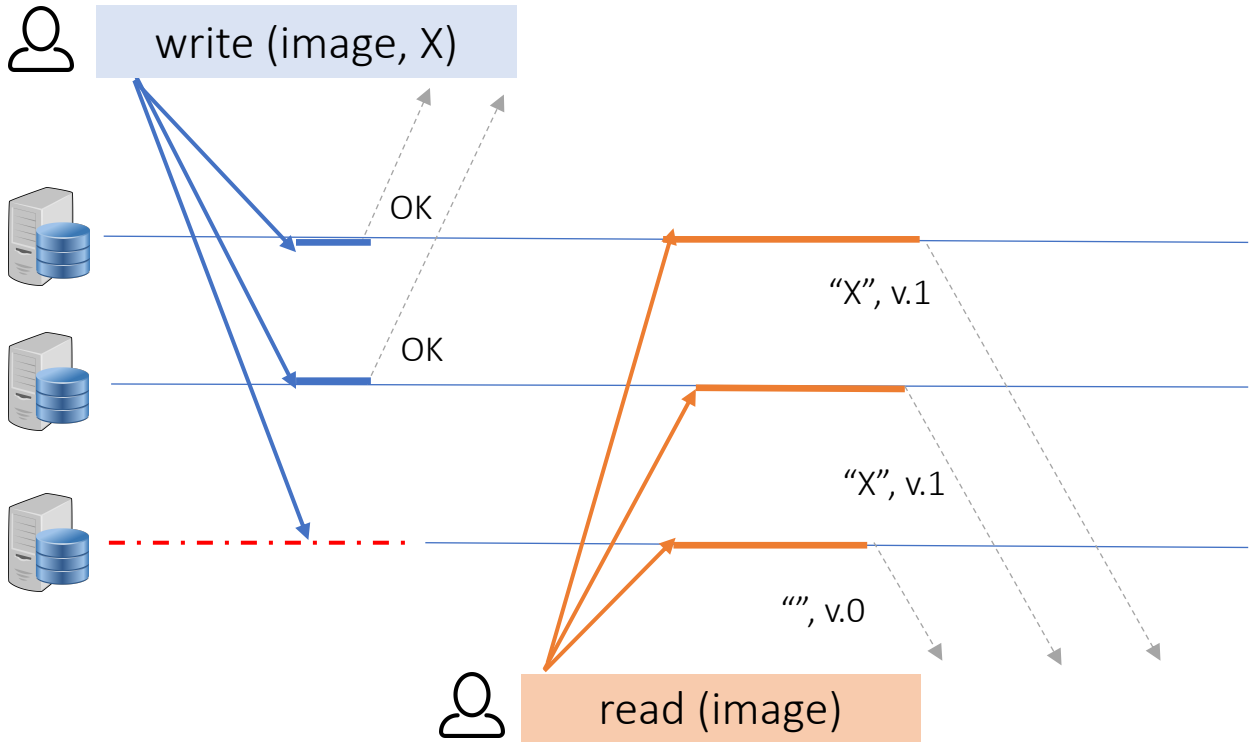
Ordering problems



Leaderless Replication – Quorum based read/writes

- Writes are successful if written to W replicas
- Reads are successful if written to R replicas

Example scenario with N=3, W=2, R=2:



W + R > N

We expect to read up-to-date value

W < N

We can process writes if a node is unavailable

R < N

We can process reads if a node is unavailable

N = 3, W = 2, R = 2:

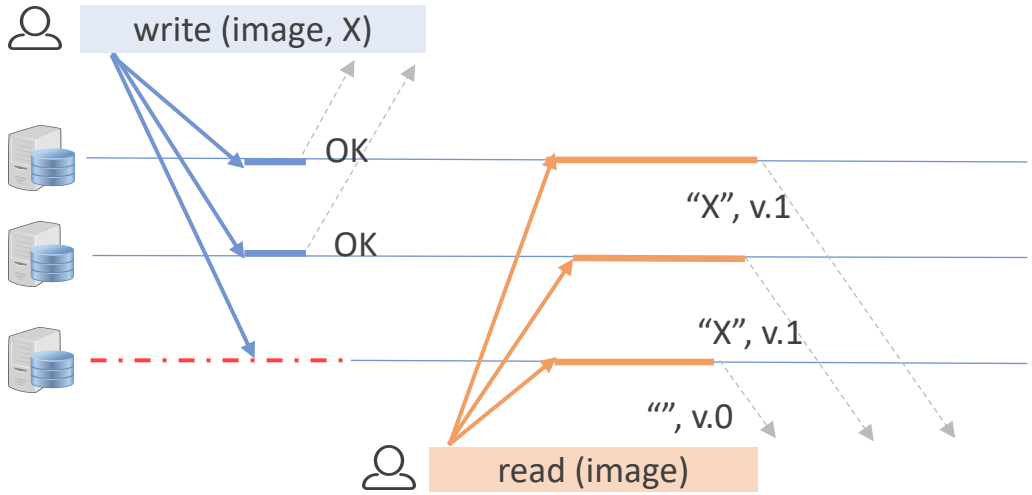
We can tolerate one faulty node

N = 5, W = 3, R = 3

We can tolerate two faulty nodes

Leaderless Replication – Quorum based read/writes

- Even with W + R > N, there are some limitations:
  - In case of concurrent writes and reads, undetermined which value will be read
  - In case of both successful and unsuccessful writes, rollbacks can cause undetermined reads
  - Possible to have conflicting concurrent writes
  - W writes may end up in different nodes than R reads (edge case)



What guarantees for writing/reading values is provided by a system?

Outline & Objectives

Why distribute?

- Scalability
- Performance
- Availability
- Fault-tolerance

Evolution of architectures

From centralized to decentralized

- Monolithic server
- Service oriented
- Microservices

(Cloud-based services)

How to replicate data?

- Leader-based
  - Synchronous
  - Asynchronous
- Multi-leader
- Leaderless

Trade-offs

Consistency model (aka semantics)

- A contract between programmer and system: The system specifies the possible results of operations
  - What can be possible results of a read operation?
  - What are possible write operations? Are concurrent updates allowed?
  - How is the last value of an object is determined?

- E.g. Consistency notions from concurrent programming
  - What are the possible values to be printed by the following multithreaded program?

```
int a = b = 0
```

Thread-1

```
a = 1
print(b)
```

Thread-2

```
b = 1
print(a)
```

Sequential consistent memory: `01, 10, 11`

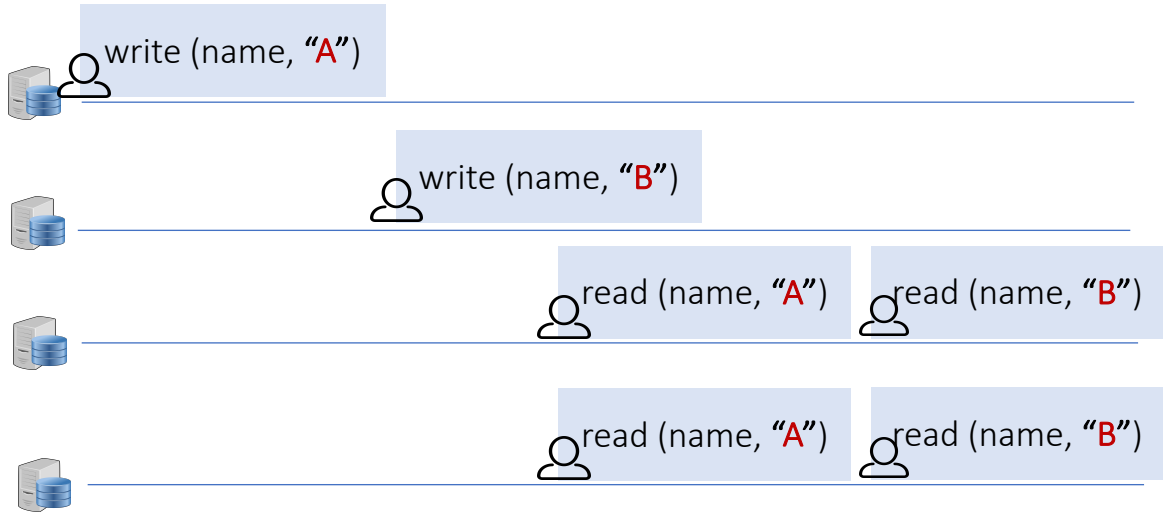
Weakly consistent memory: `01, 10, 11, 00`

Consistency models

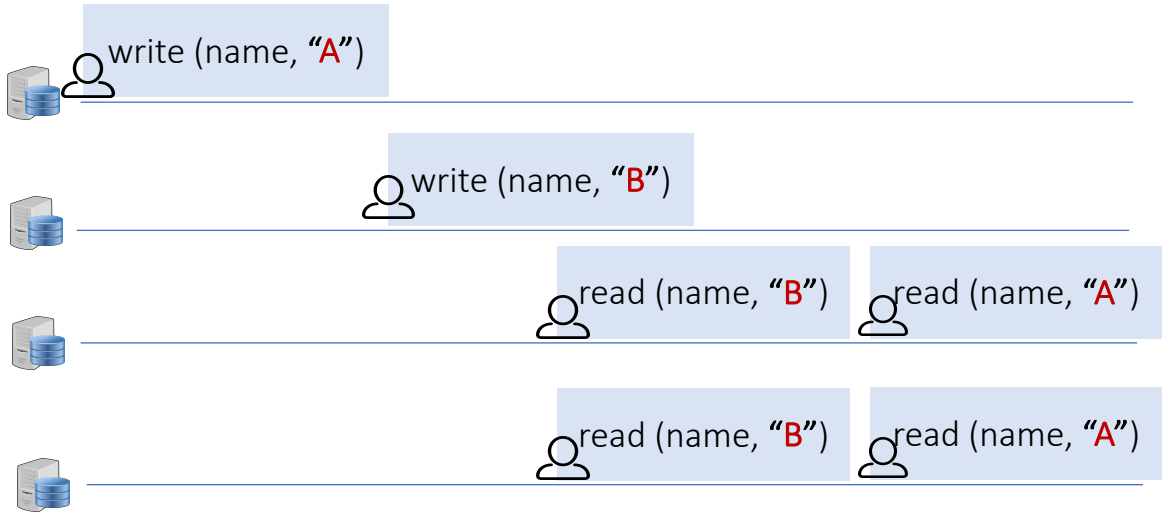
Linearizability

- There exist a total ordering of operations - the same total order at each replica
- The total order preserves **real-time ordering**

- Linearizable



- Not linearizable

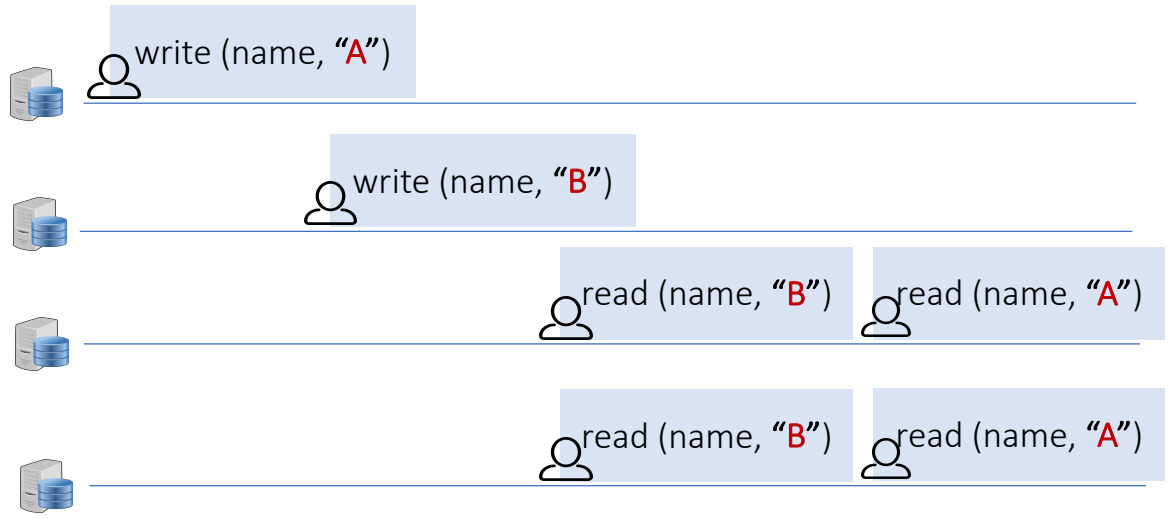


The images only show the client interactions with replicas

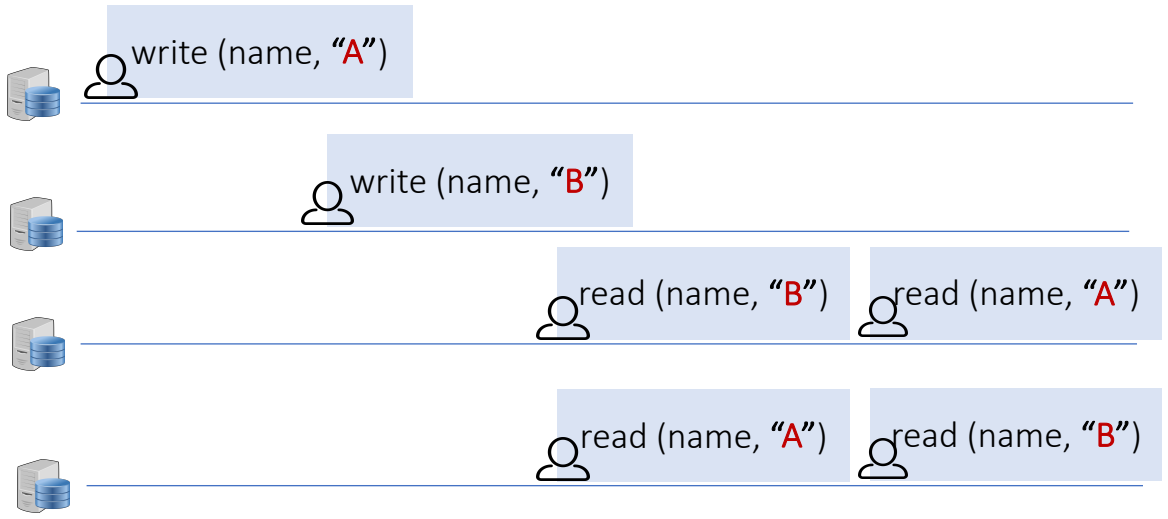
Sequential consistency

- There exist a total ordering of operations - the same total order at each replica

- Sequentially consistent



- Not sequentially consistent

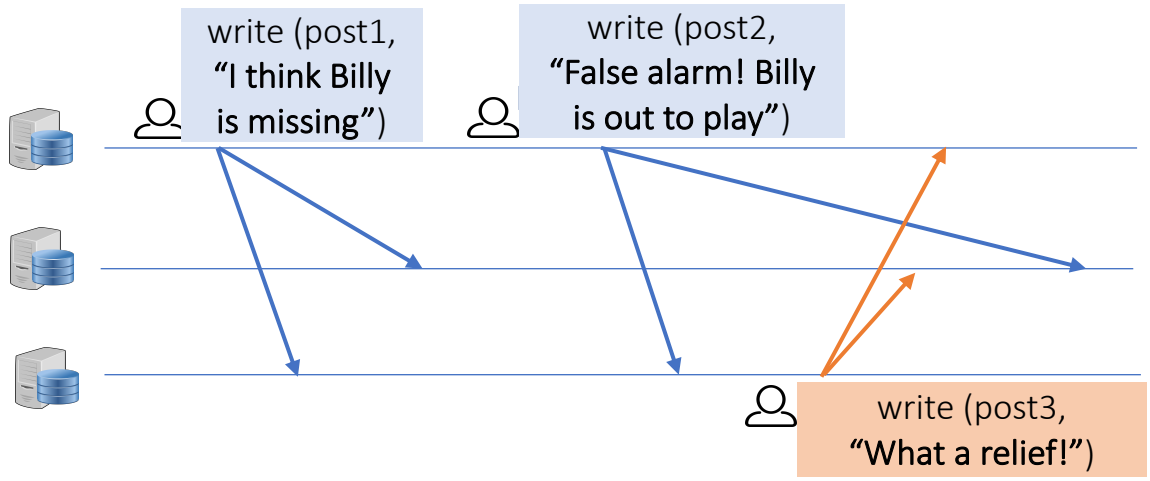


The images only show the client interactions with replicas

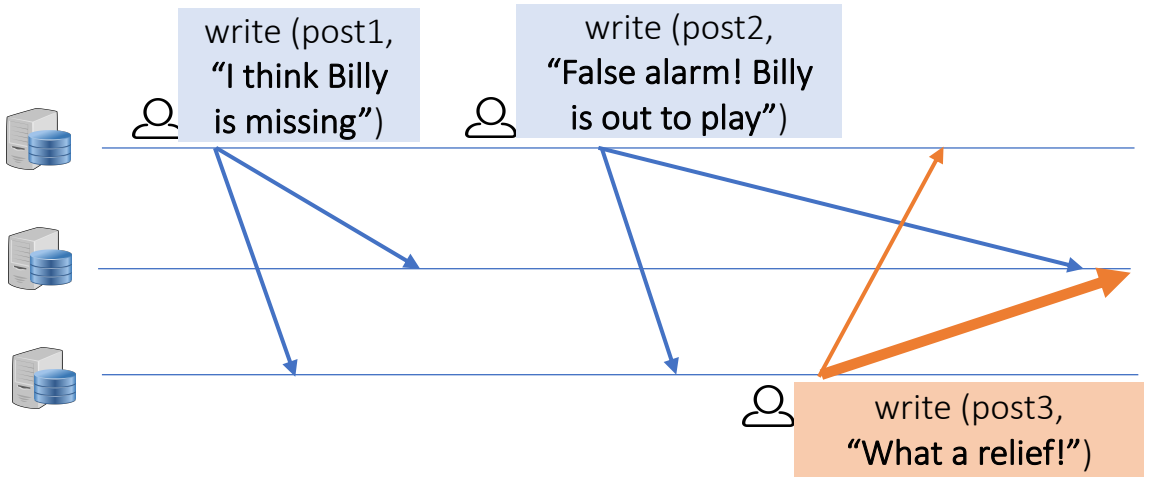
Causal consistency

- **Causally related operations are delivered to other replicas in the correct order**
  (Operations are partially-ordered)

Disallowed by causal consistency:

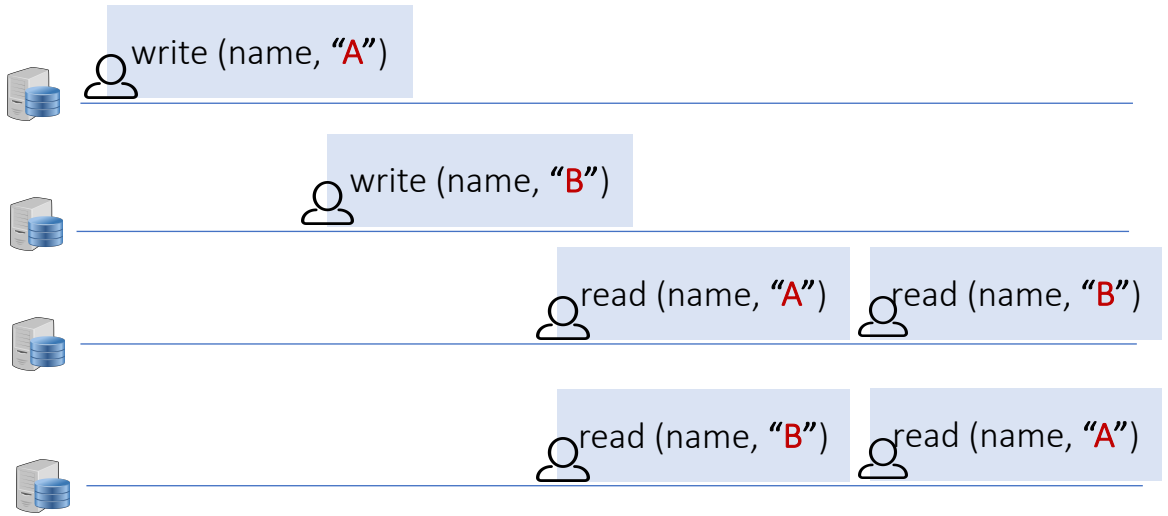


Scenario using causal consistency:

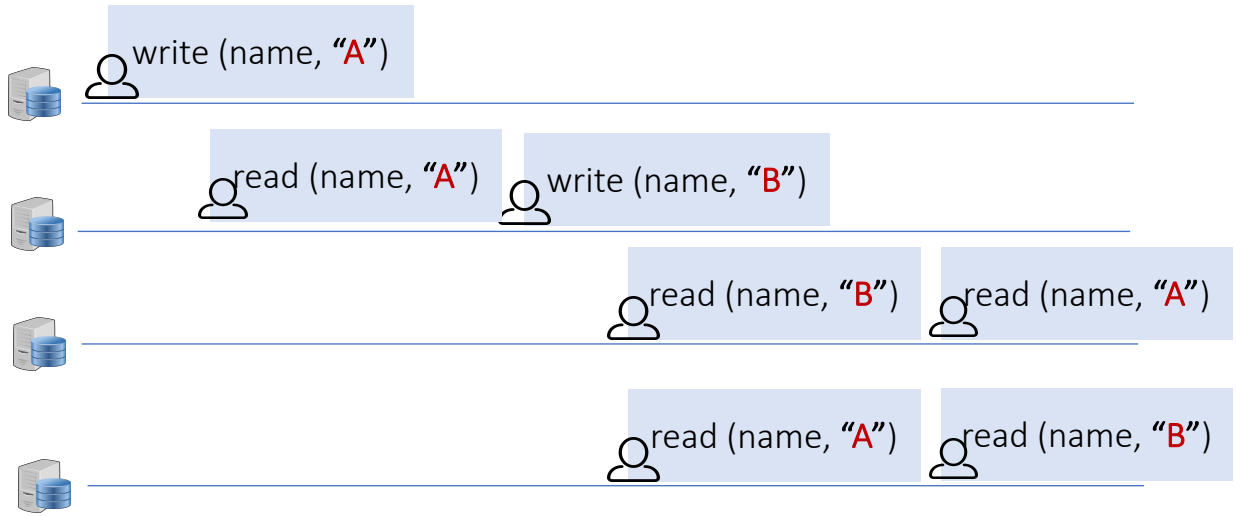


Are the executions causally consistent?

- Causally consistent



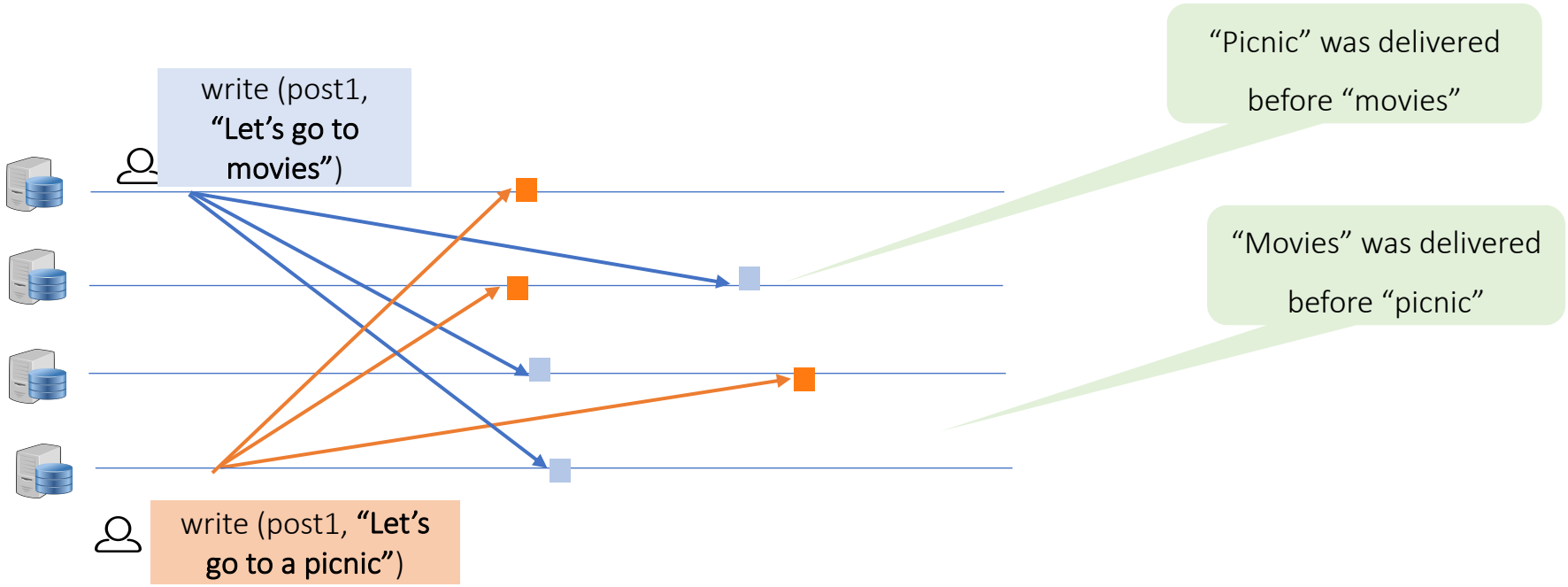
- Not causally consistent



The images only show the client interactions with replicas

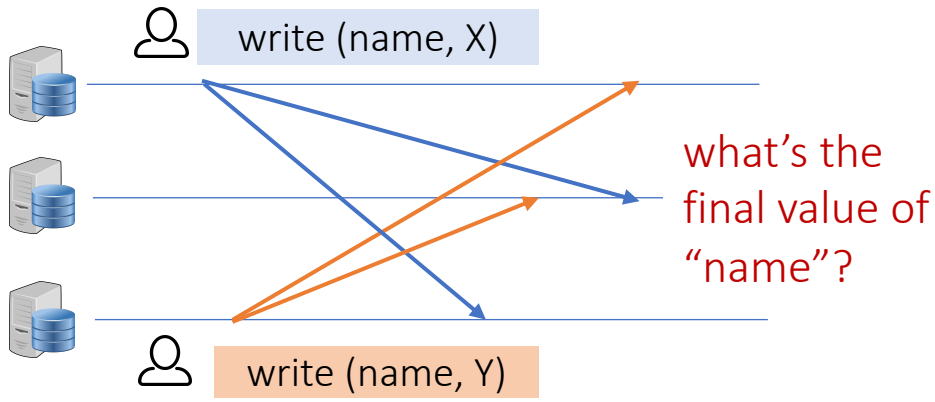
Causal consistency

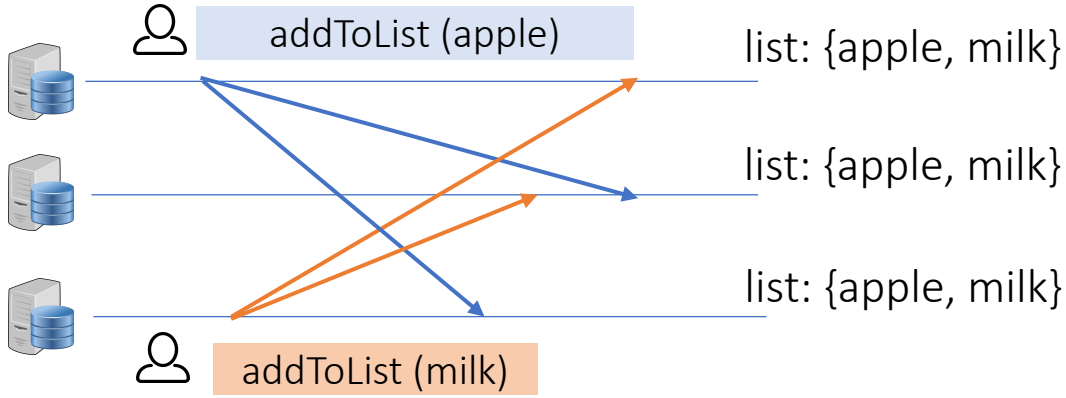
- Causally related operations are delivered to other replicas in the correct order
- **Concurrent writes may be seen in a different order on different replicas**



Causal consistency

- (Potentially) causally related transactions are delivered to other replicas in the correct order
- Concurrent writes may be seen in a different order on different replicas
- **Allows concurrent conflicting writes**



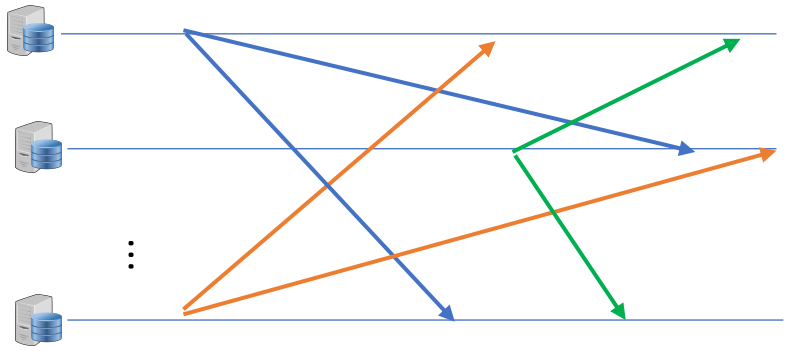


Some systems use "conflict-free" data types
e.g., Conflict-free replicated data types (CRDTs), or cloud types

- **Causal+ consistency: Replicas eventually converge**

Eventual consistency

- All updates are eventually delivered to all replicas
- All replicas reach a consistent state if no more user updates arrive



Examples:

- Search engines
  - Search results are not always consistent with the current state of the web
- Cloud file systems
  - File contents may be out-of-sync with their latest versions
- Social media applications
  - Number of likes for a video

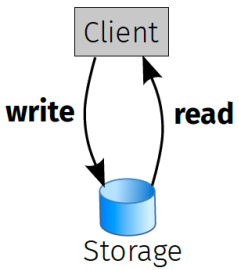
Implementing linearizable systems

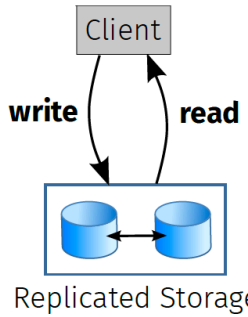
- Illusion of a single copy of the data and all operations on it are atomic
  - Single-leader replication (potentially linearizable)
  - Multi-leader replication (not linearizable)
  - Leaderless replication (not linearizable)
  - Some consensus algorithms provide linearizable executions

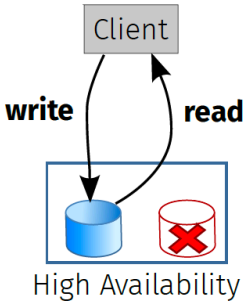
- It is costly to implement linearizability
  - High read and write latencies

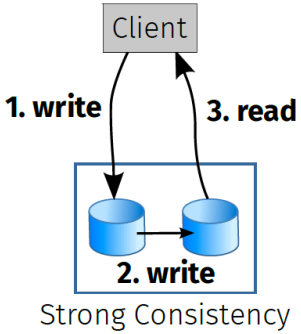
Read more: Chapter 5 of “Designing Data-Intensive Applications”, Martin Kleppmann, 2017

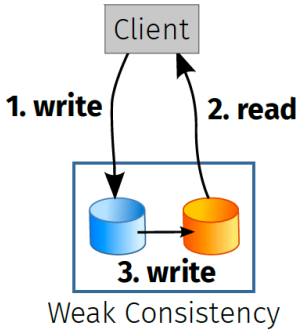
Recap: Replication, availability, consistency, partition tolerance











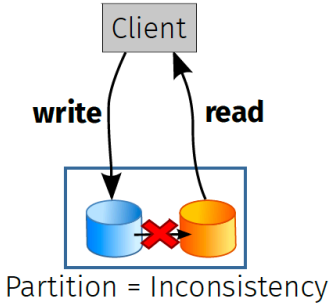
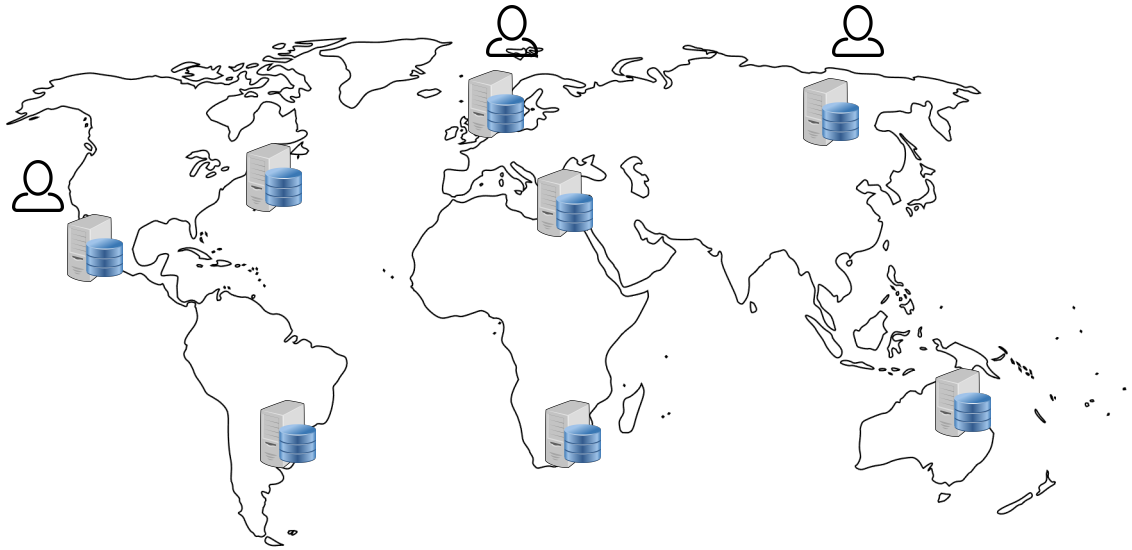


Image from: “Software Architecture”, Cesare Pautasso, 2020

CAP Theorem

Impossible to get all three of:

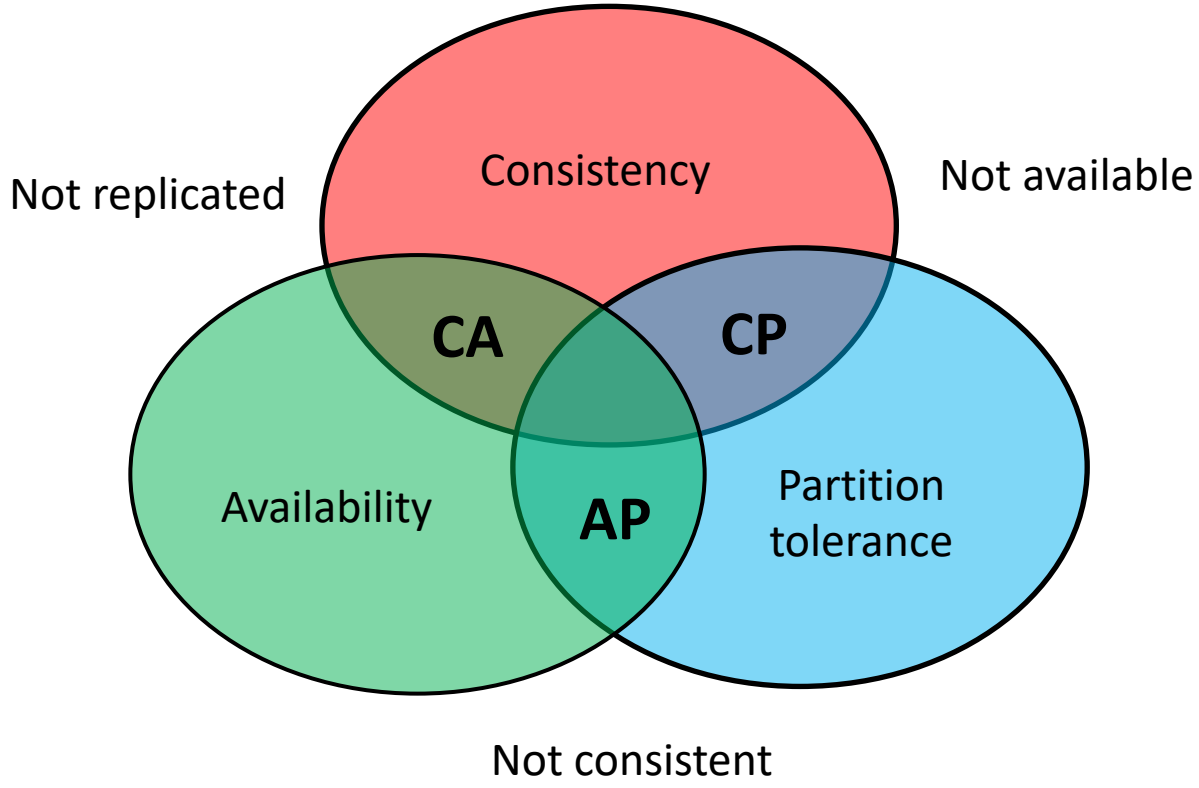
- (Strong) Consistency – All nodes in the network have the same (most recent) value.
- Availability – Every request to a non-failing replica receives a response
- Partition tolerance – The network continues to operate in the existence of component or network faults



CAP Theorem

Impossible to get all three of:

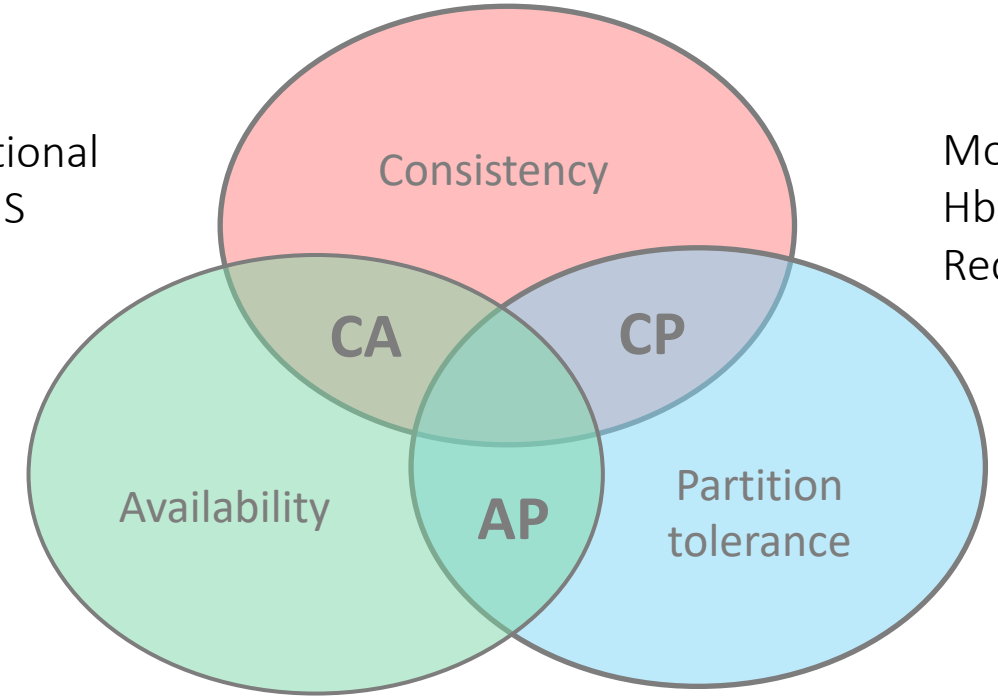
- (Strong) Consistency – All nodes in the network have the same (most recent) value.
- Availability – Every request to a non-failing replica receives a response
- Partition tolerance – The network continues to operate in the existence of component or network faults



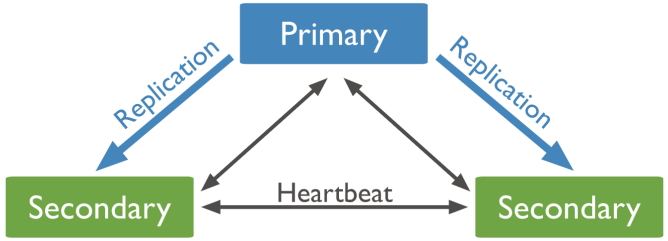
The trade-off is not
Availability vs Consistency
but
Availability vs Strong Consistency

Systems' CAP choices

Relational
DBMS

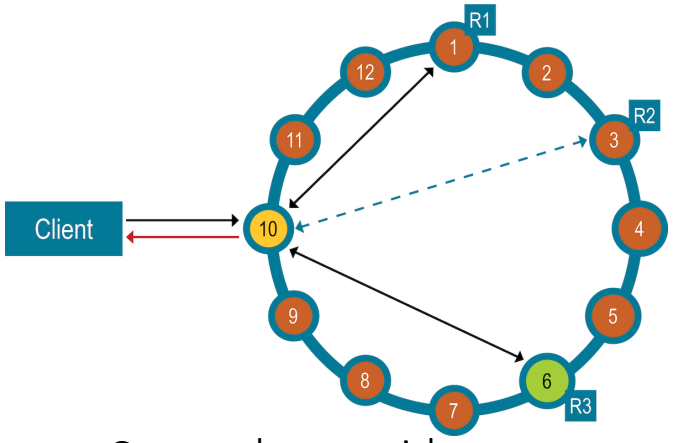


MongoDB
Hbase
Redis



e.g. MongoDB
leader based replication

CouchDB
Cassandra
DynamoDB
Riak



Cassandra provides
tunable quorum reads/writes

"Well, we'll be using some existing systems/services when building our applications. Do we need to know internal design choices of these systems?"

"Yes! Trade-offs and limitations of the underlying services you use reflect on your application."


What are the distributed components in the system?


What information do the components exchange with each other?


How do the system components communicate with each other (e.g., synchronously or asynchronously)?


What kind of faults does the system tolerate? How does it handle failing components?


What are the trade-offs of the distributed design (consistency, availability, partition-tolerance)?

Outline & Objectives

Why distribute?

- Scalability
- Performance
- Availability
- Fault-tolerance

Evolution of architectures

From centralized to decentralized

- Monolithic server
- Service oriented
- Microservices

(Cloud-based services)

How to replicate data?

- Leader-based
  - Synchronous
  - Asynchronous
- Multi-leader
- Leaderless

Trade-offs

CAP Impossibility:

- (Strong) Consistency
- Availability
- Partitioning