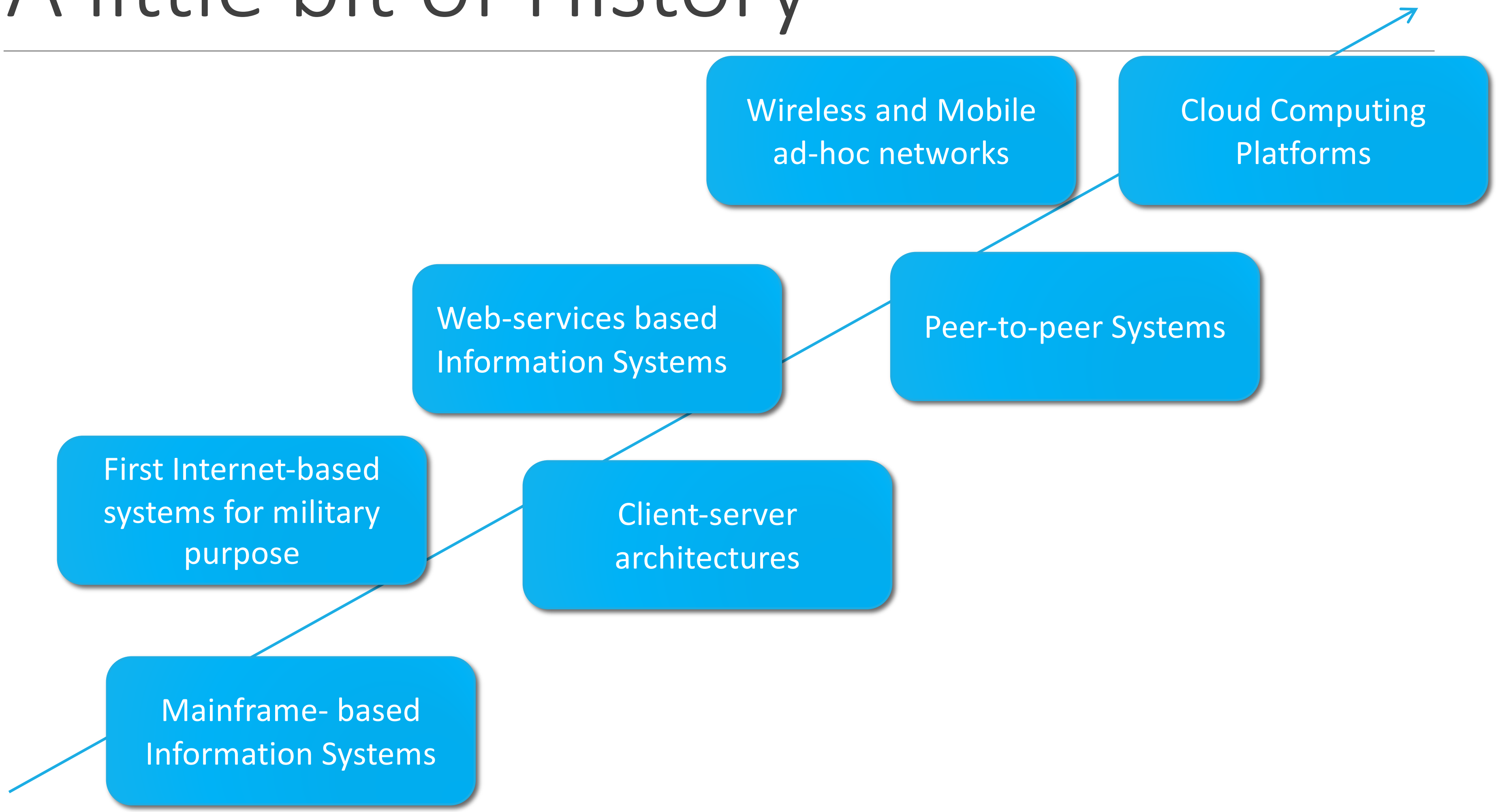
Distributed Systems
Master of Science in Engineering in Computer Science

AA 2020/2021

LECTURE 16: CAP THEOREM

A little bit of History



Relational Databases History

Relational Databases – mainstay of business

Web-based applications caused spikes

- Especially true for public-facing e-Commerce sites

Developers begin to front RDBMS with memcache or integrate other caching mechanisms within the application

Scaling Up

Issues with scaling up when the dataset is just too big

RDBMS were not designed to be distributed

Began to look at multi-node database solutions

Known as ‘scaling out’ or ‘horizontal scaling’

Different approaches include:

- Master-slave
- Sharding

Scaling RDBMS – Master/Slave

Master-Slave

- All writes are written to the master. All reads performed against the replicated slave databases
- Critical reads may be incorrect as writes may not have been propagated down
- Large data sets can pose problems as master needs to duplicate data to slaves

Scaling RDBMS - Sharding

Partition or sharding

- Scales well for both reads and writes
- Not transparent, application needs to be partition-aware
- Can no longer have relationships/joins across partitions
- Loss of referential integrity across shards

Other ways to scale RDBMS

Multi-Master replication

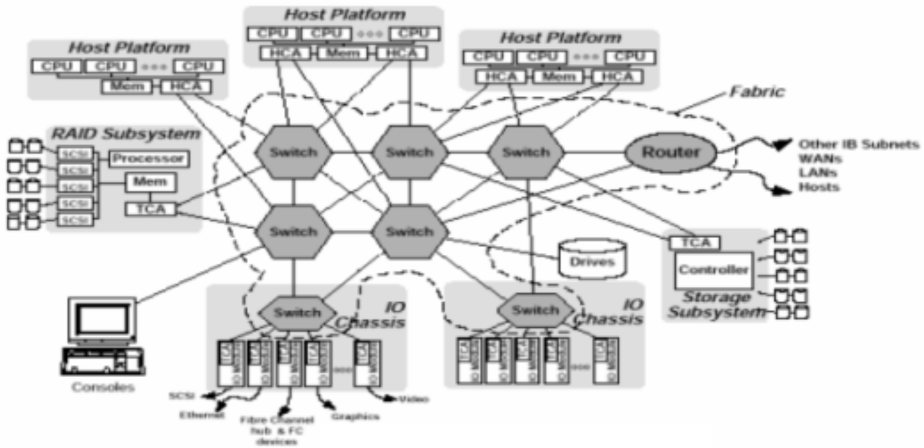
INSERT only, not UPDATES/DELETES

No JOINs, thereby reducing query time

- This involves de-normalizing data

In-memory databases

Today…


Host Platform
CPU
CPU
CPU
Mem
HCA
Host Platform
CPU
CPU
CPU
HCA
Mem
HCA
Host Platform
CPU
CPU
CPU
HCA
Mem
HCA
Fabric
RAID Subsystem
SCSI
Processor
Mem
TCA
Switch
Switch
Switch
Router
Other IB Subnets
WANs
LANs
Hosts
Switch
Switch
Drives
TCA
Controller
Storage Subsystem
Switch
IO Chassis
Switch
IO Chassis
Consoles
SCSI
Ethernet
Fibre Channel hub & FC devices
Graphics
Video

Replication

PRELIMINARY NOTIONS

Replication as a way to scale up

N clients c_1, c_2,...,c_n access a set X of objects (tables, tuples, etc.)

Each object $x \in X$ has its own internal state

Clients access the object x trough the invocation of operations

The set of operations that allow clients to interact with the object define its semantics

Replication Model

Each object **x** is developed by a set {x^1, x^2… x^m } of physical copies called "replicas"

Each replica is located in a different physical location

Requirements

Transparency:

- Clients must have the illusion to interact with a single object
- Object interfaces do not change

Consistency:

- Operations must produce results as if they are executed on a single object

Consistency: Linearizability

Sequential System:

- At each time unit only one process interacts with the object
- Operations are specified by pre conditions and post conditions

Concurrent System:

- Multiple clients concur to access the object
- Management of concurrent updates

LINEARIZABILITY

Specify how a concurrent object must behave according with its sequential specification

Linearizability: Idea

Clients should have the illusion of interacting with a unique physical object even in case of concurrency

Each operation must produce the same effect it would produce if executed in isolation

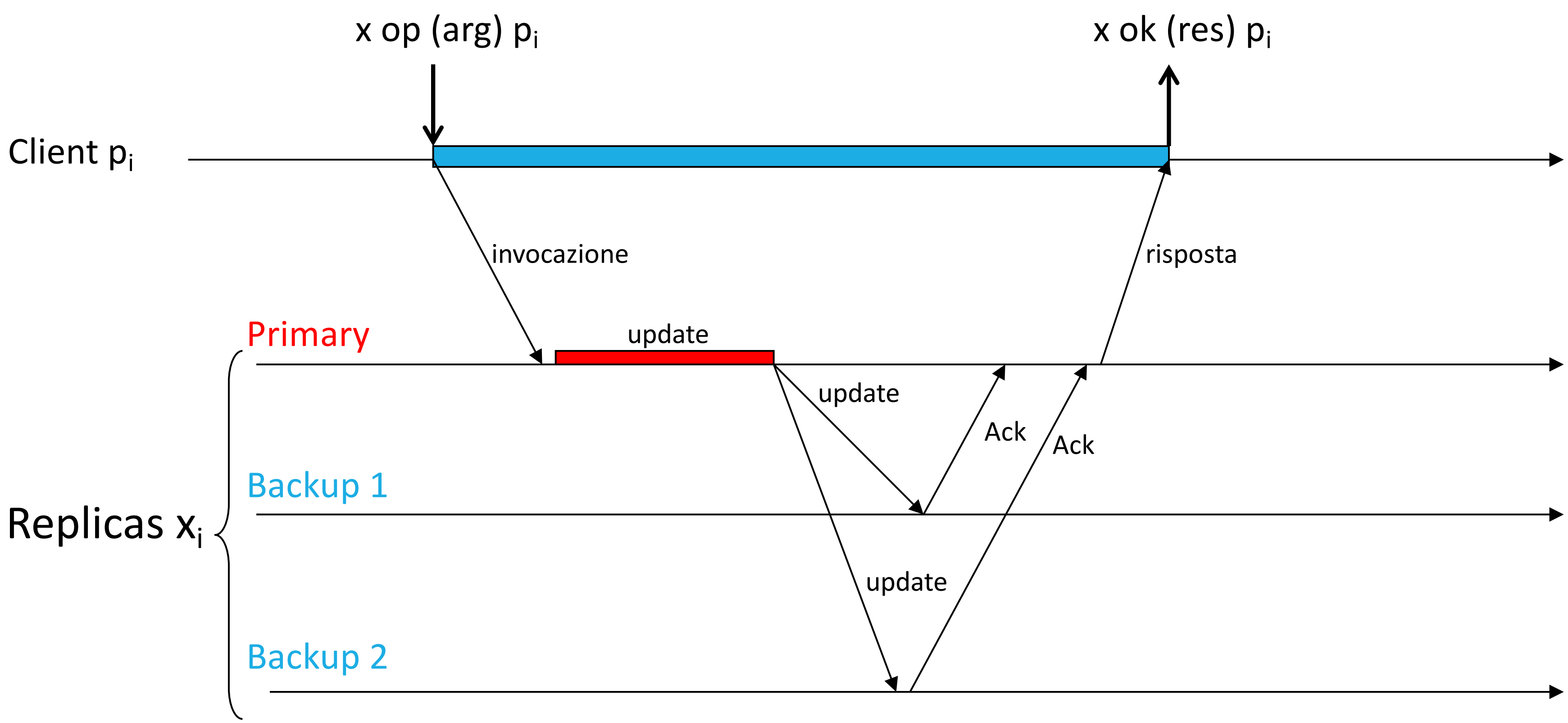
The order between sequential operations must be preserved

Basic Replication Techniques

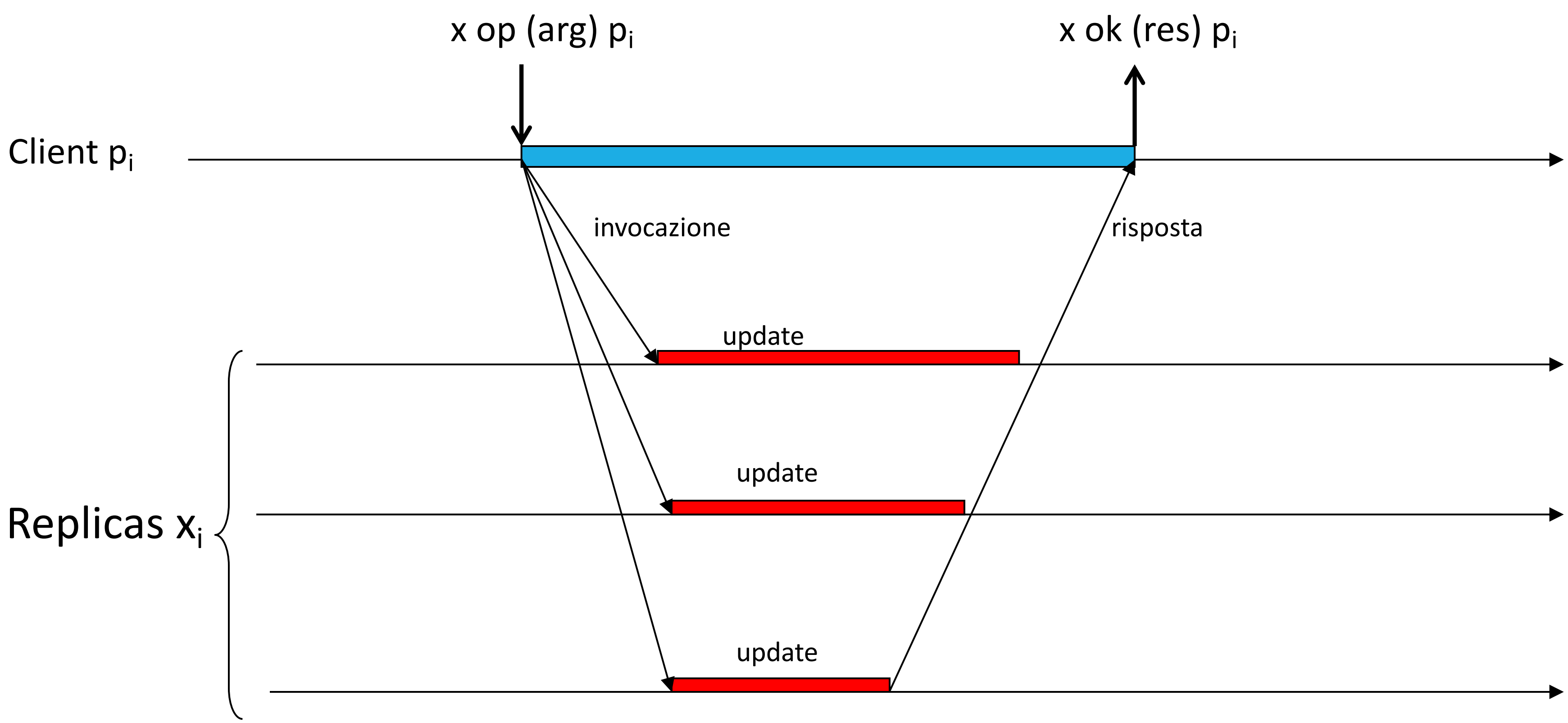
Two basic approaches to replication:

- Primary Backup (passive replication)
- Active Replication

Passive Replication

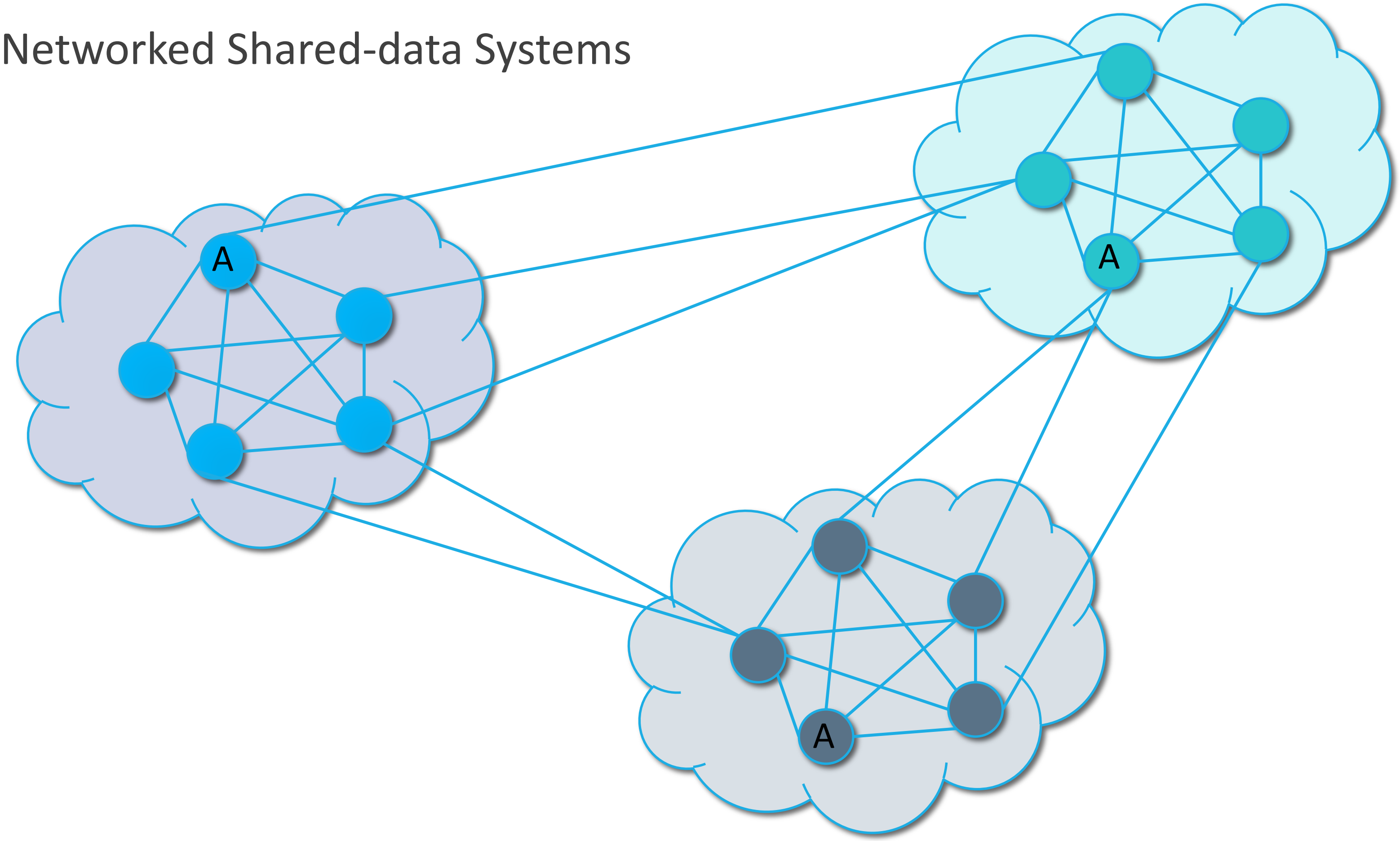


Active Replication



Context

Networked Shared-data Systems



Fundamental Properties

Consistency
- (informally) “every request receives the right response”
- E.g. If I get my shopping list on Amazon I expect it contains all the previously selected items

Availability
- (informally) “each request eventually receives a response”
- E.g. eventually I access my shopping list

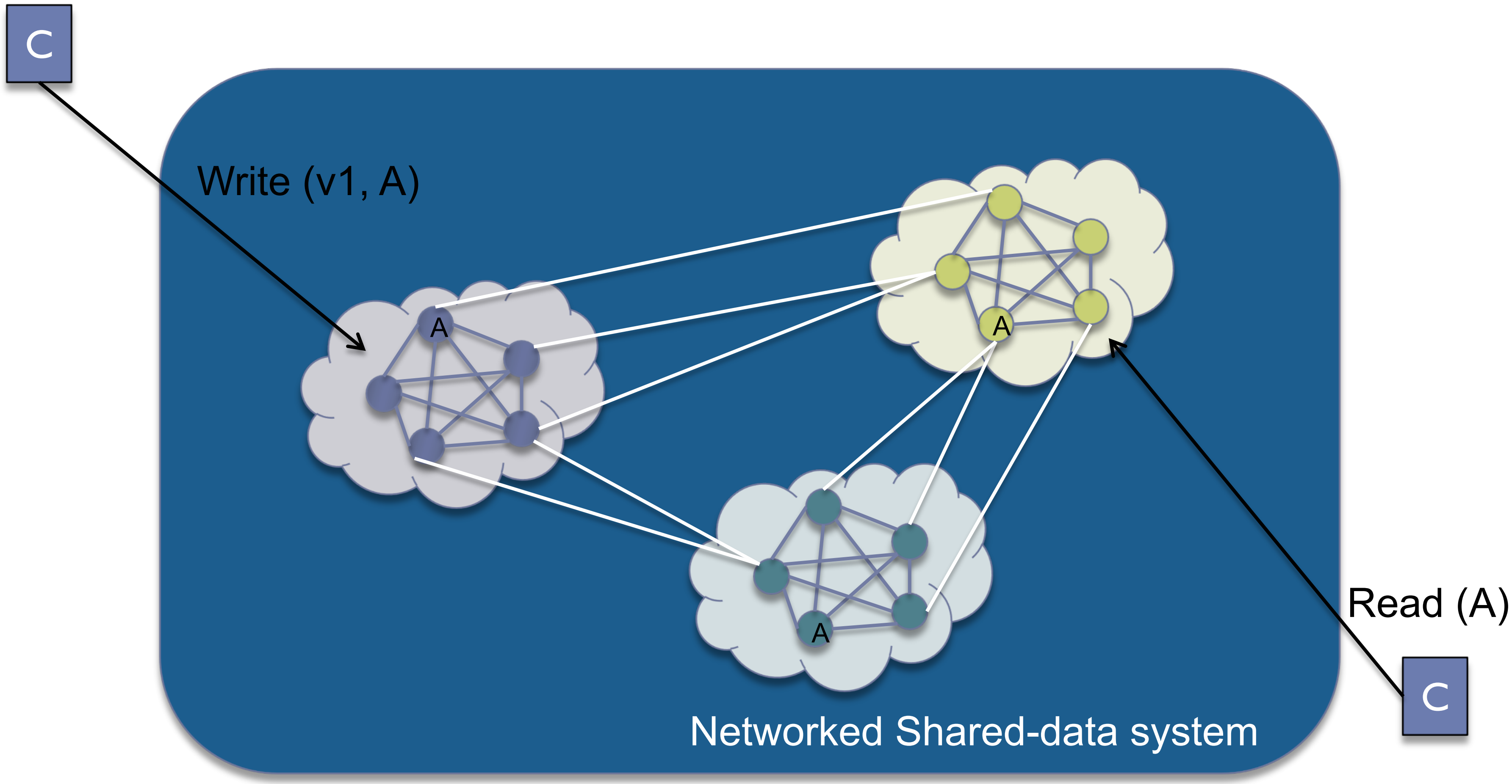
tolerance to network **P**artitions
- (informally) “servers can be partitioned in to multiple groups that cannot communicate with one other”

CAP Theorem

- 2000: Eric Brewer, PODC conference keynote
- 2002: Seth Gilbert and Nancy Lynch, ACM SIGACT News 33(2)

"Of three properties of shared-data systems (Consistency, Availability and tolerance to network Partitions) only two can be achieved at any given moment in time."

Proof Intuition

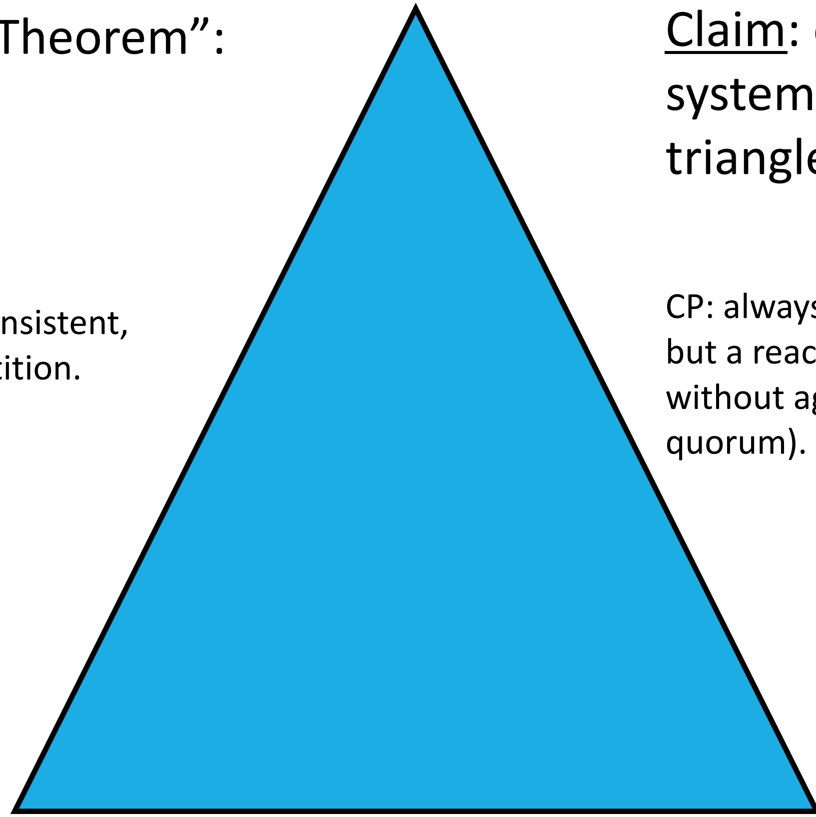


consistency

C

Fox&Brewer “CAP Theorem”:
C-A-P: choose two.

<u>Claim</u>: every distributed system is on one side of the triangle.



CA: available, and consistent, unless there is a partition.

CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).

A

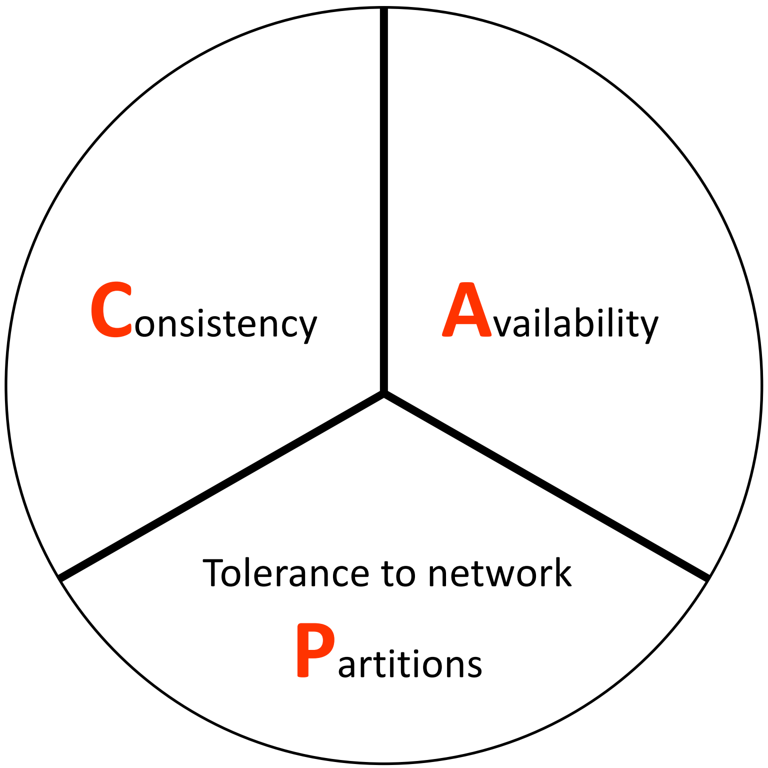
Availability

AP: a reachable replica provides service even in a partition, but may be inconsistent.

P

Partition-resilience

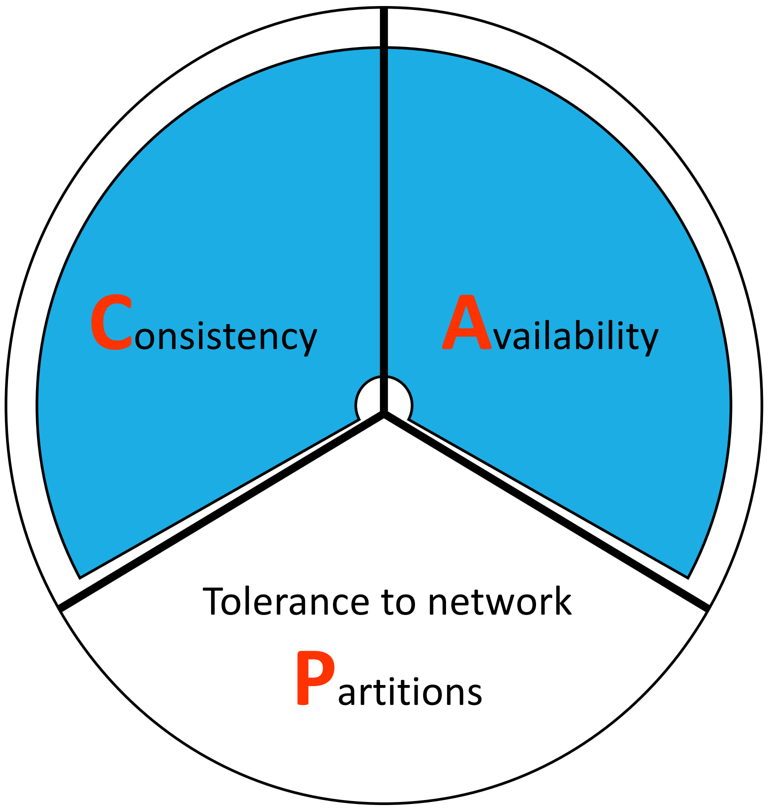
The CAP Theorem



Theorem: You can have **at most two** of these invariants for any shared-data system

Corollary: consistency boundary must choose A or P

Forfeit Partitions



Examples

Single-site databases

Cluster databases

LDAP

Fiefdoms

Traits

2-phase commit

cache validation protocols

The “inside”

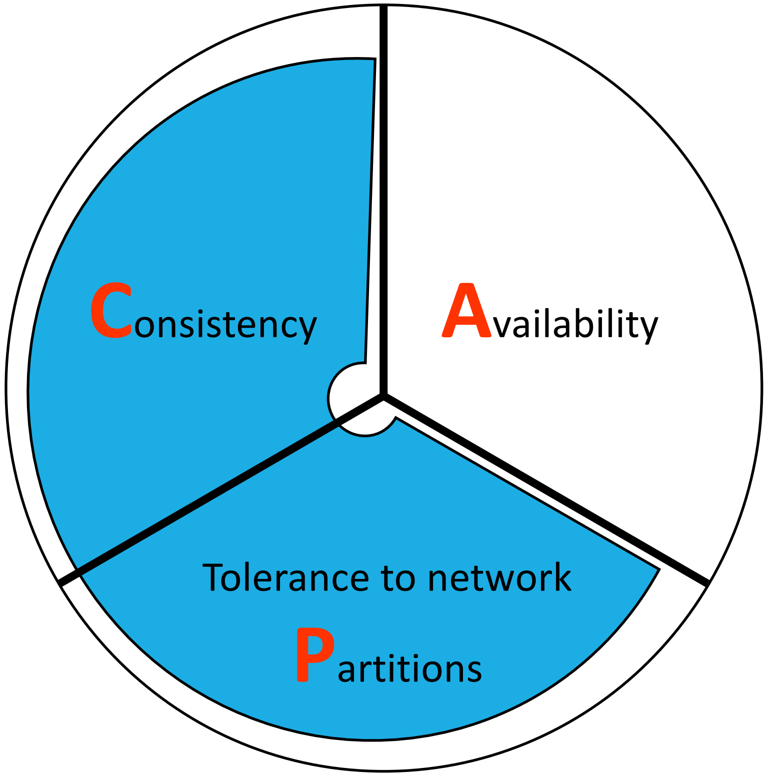
Observations

CAP states that in case of failures you can have at most two of these three properties for any shared-data system

To scale out, you have to distribute resources.

- P in not really an option but rather a need
- The real selection is among consistency or availability
- In almost all cases, you would choose availability over consistency

Forfeit Availability



Examples

Distributed databases

Distributed locking

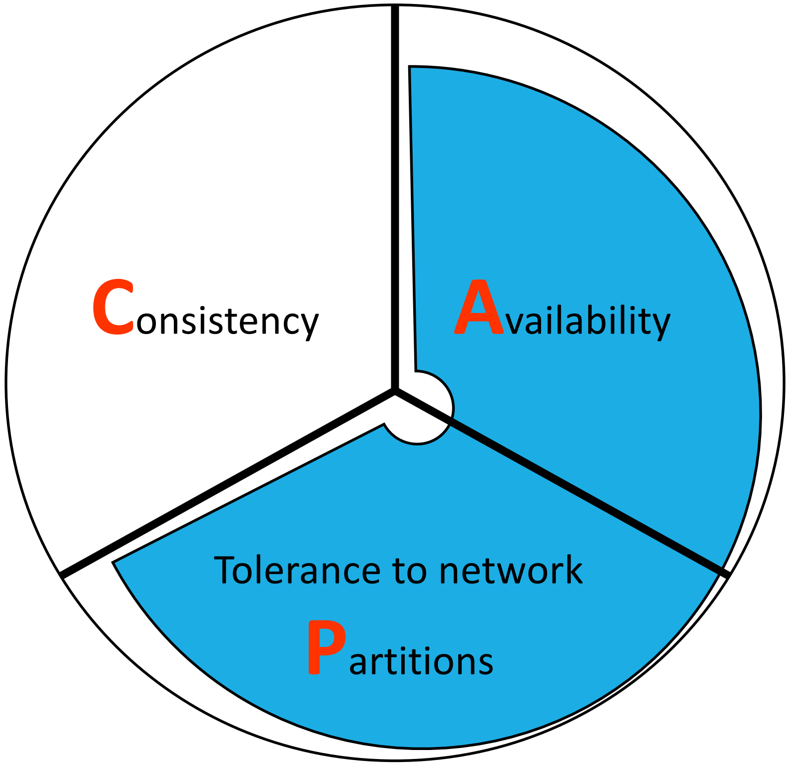
Majority protocols

Traits

Pessimistic locking

Make minority partitions unavailable

Forfeit Consistency



Examples

Coda

Web caching

DNS

Emissaries

Traits

expirations/leases

conflict resolution

Optimistic

The “outside”

Consistency Boundary Summary

We can have consistency & availability within a cluster.

- No partitions within boundary!

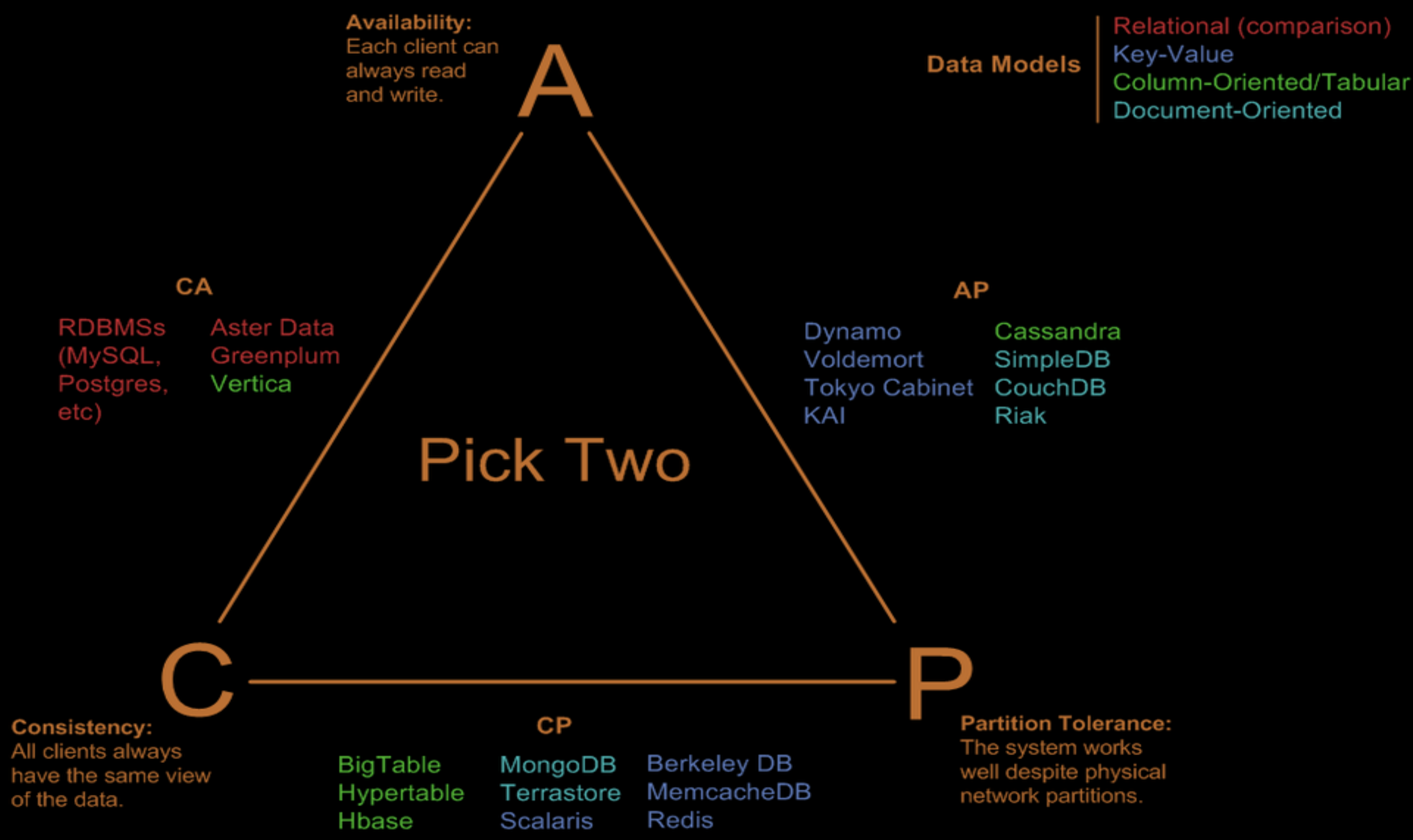
OS/Networking better at A than C

Databases better at C than A

Wide-area databases can't have both

Disconnected clients can't have both

Visual Guide to NoSQL Systems



CAP, ACID and BASE

BASE stands for Basically Available Soft State Eventually Consistent system.

Basically Available: the system available most of the time and there could exists a subsystems temporarily unavailable

Soft State: data are “volatile” in the sense that their persistence is in the hand of the user that must take care of refresh them

Eventually Consistent: the system eventually converge to a consistent state

CAP, ACID and BASE

Relation among ACID and CAP is more complex

Atomicity: every operation is executed in “all-or-nothing” fashion

Consistency: every transaction preserves the consistency constraints on data

Integrity: transaction does not interfere. Every transaction is executed as it is the only one in the system

Durability: after a commit, the updates made are permanent regardless possible failures

CAP, ACID and BASE

CAP

C here looks to single-copy consistency

A here look to the service/data availability

ACID

C here looks to constraints on data and data model

A looks to atomicity of operation and it is always ensured

I is deeply related to CAP. I can be ensured in at most one partition

D is independent from CAP

Warning!

What CAP says:

- When you have a partition in the network you cannot have both C and A

Wh

- '

During Normal Periods (i.e. period with no partitions) both C and A can be achieved

2 out of 3 is misleading

Partitions are rare events

- there are little reasons to forfeit by design C or A

Systems evolve along time

- Depending on the specific partition, service or data, the decision about the property to be sacrificed can change

C, A and P are measured according to continuum

- Several level of Consistency (e.g. ACID vs BASE)
- Several level of Availability
- Several degree of partition severity

2 of 3 is misleading

In principle every system should be designed to ensure both C and A in normal situation

When a partition occurs the decision among C and A can be taken

When the partition is resolved the system takes corrective action coming back to work in normal situation

Consistency/Latency Trade Off

CAP does not force designers to give up A or C but why there exists a lot of systems trading C?

LATENCY !

CAP does not explicitly talk about latency…

… however latency is crucial to get the essence of CAP

Consistency/Latency Trade Off

High Availability

- High Availability is a strong requirement of modern shared-data systems

Replication

- To achieve High Availability, data and services must be replicated

Consistency

- Replication impose consistency maintenance

Latency

- Every form of consistency requires communication and a stronger consistency requires higher latency

PACELC

Abadi proposes to revise CAP as follows:

"*PACELC (pronounced pass-elk): if there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?*"

Consistency Spectrum

Consistency Criteria

Different perspective between Data base community and Distributed System community

- Data-centric *vs* client-centric

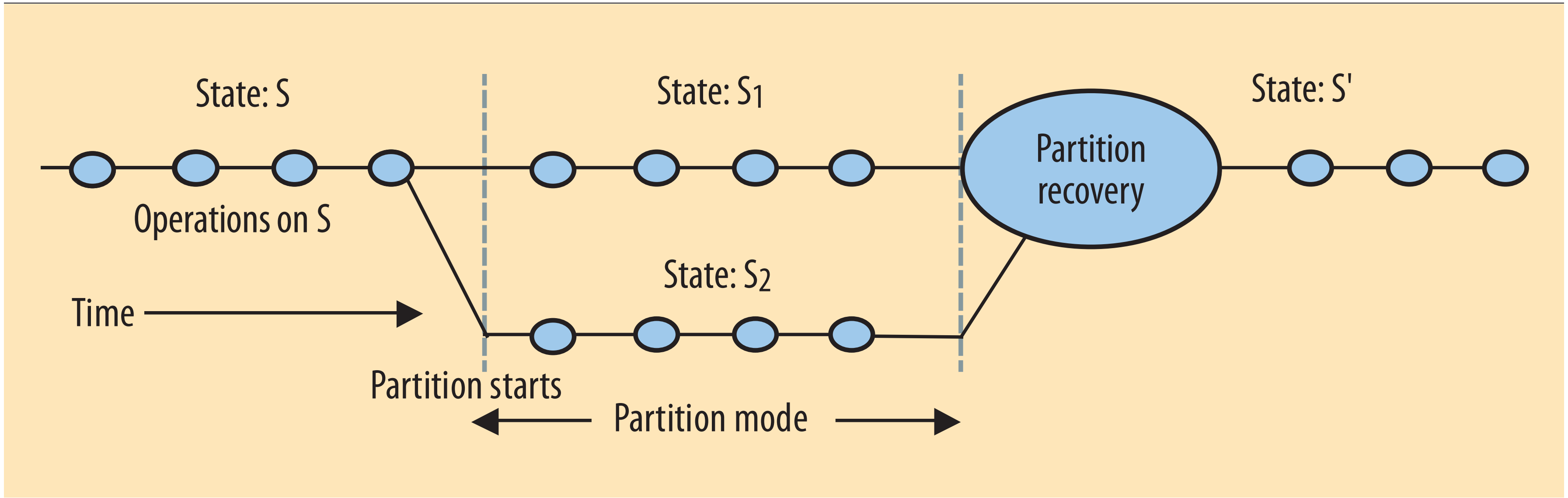
ACID ➔ Strong Consistency

Several Degrees of Weak Consistency

- Eventual Consistency
- Read-your-writes
- Monotonic read
- Monotonic write

Partition Management

Partitions Management



Partition Detection

Activating Partition Mode

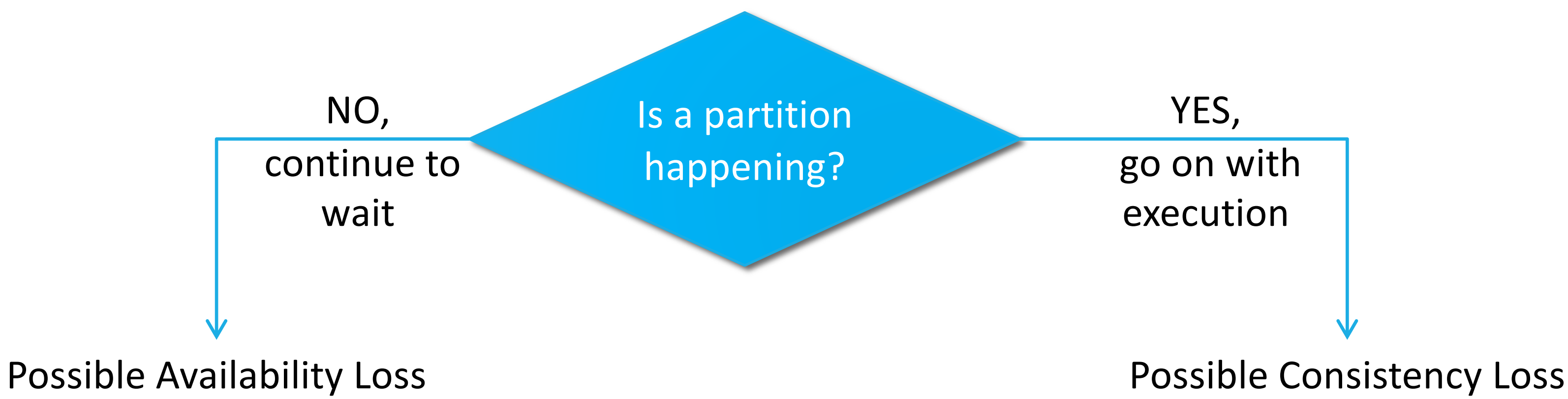
Partition Recovery

Partition Detection

CAP does not explicitly talk about latencies

However…

- To keep the system live time-outs must be set
- When a time-out expires the system must take a decision



Partition Detection

Partition Detection is not global

- An interacting part may detect the partition, the other not.
- Different processes may be in different states (partition mode vs normal mode)

When entering Partition Mode the system may

- Decide to block risk operations to avoid consistency violations
- Go on limiting a subset of operations

Which Operations Should Proceed?

Live operation selection is an hard task

- Knowledge of the severity of invariant violation
- Examples
  - every key in a DB must be unique
    - Managing violation of unique keys is simple
    - Merging element with the same key or keys update
  - every passenger of an airplane must have assigned a seat
    - Managing seat reservations violation is harder
    - Compensation done with human intervention
- Log every operation for a possible future re-processing

Partition Recovery

When a partition is repaired, partitions' logs may be used to recover consistency

Strategy 1: roll-back and execute again operations in the proper order (using version vectors)

Strategy 2: disable a subset of operations (Commutative Replicated Data Type - CRDT)

Basic Techniques: Version Vector

In the version vector we have an entry for any node updating the state

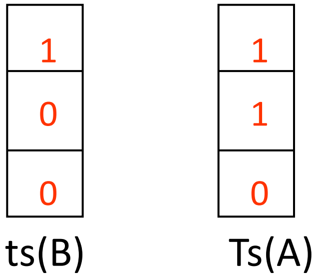
Each node has an identifier

Each operation is stored in the log with attached a pair <nodeId, timeStamp>

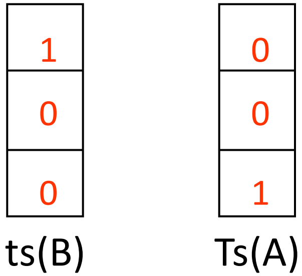
Given two version vector A and B, A is newer than B if

- For any node in both A and B, ta(B) ≤ ts(A) and
- There exists at least one entry where ta(B) < ts(A)

Version Vectors: example



ts(A) < ts(B) then A $\rightarrow$ B



ts (A) $\neq$ ts (B) then A || B

POTENTIALLY INCONSISTENT!

Basic Techniques: Version Vector

Using version vectors it is always possible to determine if two operations are causally related or they are concurrent (and then dangerous)

Using vector versions stored on both the partitions it is possible to re-order operations and raising conflicts that may be resolved by hand

Recent works proved that this consistency is the best that can be obtained in systems focussed on latency

Basic Techniques: CRDT

Commutative Replicated Data Type (CRDT) are data structures that provably converges after a partition (e.g. set).

Characteristics:

- All the operations during a partition are commutative (e.g. add(a) and add(b) are commutative) or
- Values are represented on a lattice and all operations during a partitions are monotonically increasing wrt the lattice (giving an order among them)
  - Approach taken by Amazon with the shopping cart.
- Allows designers to choose A still ensuring the convergence after a partition recovery

Basic Techniques: Mistake Compensation

Selecting A and forfaiting C, mistakes may be taken

- Invariants violation

To fix mistakes the system can

- Apply deterministic rule (e.g. “last write win”)
- Operations merge
- Human escalation

General Idea:

- Define specific operation managing the error
  - E.g. re-found credit card

References

1. Brewer "CAP twelve years later: How the "rules" have changed" http://ieeexplore.ieee.org/document/6133253/ (see NOTE above)
2. Abadi "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story" http://ieeexplore.ieee.org/document/6127847/ (see NOTE above)

NOTE: Use the Sapienza proxy to access this paper. Instruction on how to do it can be found here https://web.uniroma1.it/sbs/easybixy/easybixy