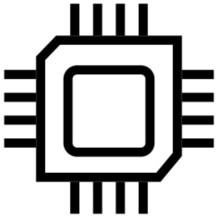
Distributed Systems 101

Intuition of distributed systems.

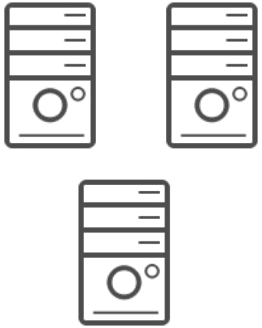
Maksim Ekimovskii

What?



Single processor system

VS



What?

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

- *Leslie Lamport*

Goals

- Better performance
  - Lower latency
- Better scalability
  - More storage
  - More computational capacity
- Better availability -> uptime / (uptime + downtime)
  - Be fault tolerant
  - Be resilient

Availability % → How much downtime is allowed per year?
90% ("one nine") → More than a month
99% ("two nines") → Less than 4 days
99.9% ("three nines") → Less than 9 hours
99.99% ("four nines") → Less than an hour
99.999% ("five nines") → ~ 5 minutes
99.9999% ("six nines") → ~ 31 seconds

Paradox

- Why do we use distributed systems? - Stand higher load with less failures
- What’s the problem with that? **Failures.**

8 fallacies

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

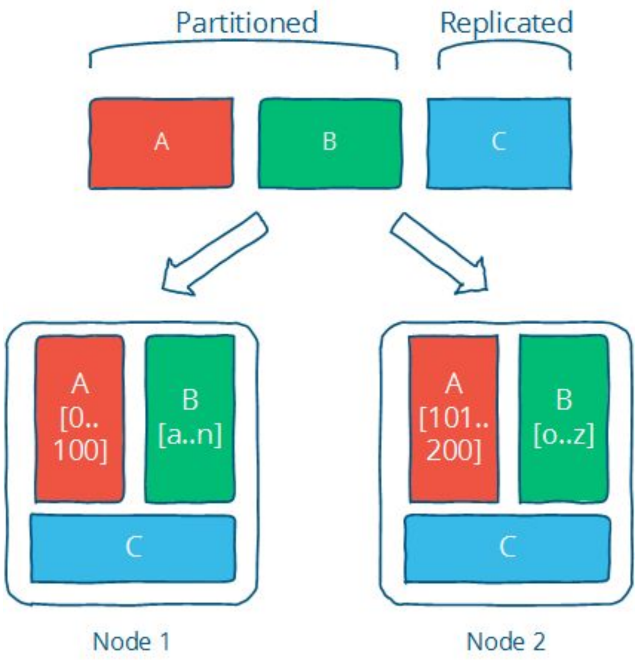
Problems

- Partitioning
- High availability
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection (consensus)

Problems

- Partitioning → DHT, Consistent Hashing
- High availability → Replication / Redundancy
- Handling temporary failures → Replicas synchronization
- Recovering from permanent failures → Replicas synchronization
- Membership and failure detection (consensus) → Gossip-based membership protocols

Partitioning and Replication



http://book.mixu.net/distsys/intro.html

CAP theorem

“Out of C, A and P you can’t choose CA”

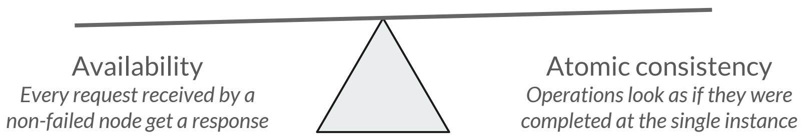
https://codahale.com/you-cant-sacrifice-partition-tolerance/

CAP theorem aka “Pick ~~any~~ two”

- Consistency
  - Atomic consistency among multiple nodes for a single object on a single operation. Users don’t see any stale data. (e.g Two-Phase commit protocol)
- Availability
  - Any non-failed node sends non-error response to every request it gets.
- Partition tolerance
  - Nodes communicate over network asynchronously and messages may be delayed or dropped (e.g implies faults tolerance and resiliency patterns)

Trade-offs

- Availability vs Consistency
- Scalability vs Transactionality



ACID vs CAP

- Atomicity Consistency Isolation Durability - single node system, typical SQL RDBMS
- CAP - distributed

ACID vs CAP

- Atomicity Consistency Isolation Durability - single node system, typical SQL RDBMS
- CAP - distributed

ACID: Proven strong consistency and transactionality (poor availability)

AP: Large volumes of data and high availability

CP: Distributed locks, strong consistency among several nodes

ACID vs CAP

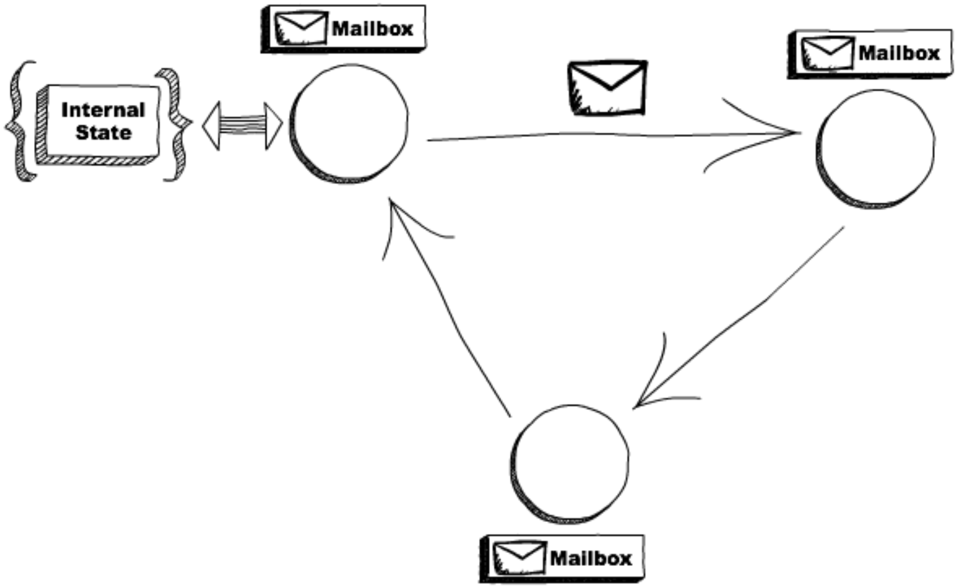
- Atomicity Consistency Isolation Durability - single node system, typical SQL RDBMS
- CAP - distributed

ACID: PostgreSQL, MySQL, Oracle, Microsoft SQL Server, DB2

AP: Cassandra, Riak, DynamoDB, ElasticSearch

CP: Zookeeper, Google BigTable, MongoDB (well, not really, but they claim that), HBase

Actor model



Actors vs microservices

Same same, but different

Actors vs microservices

Actors

- Math model of concurrent computations
- Helps (simplifies) to develop concurrent apps

Microservices

- Implementation of SOA
- Helps to build distributed systems

Abstraction models

Set of assumptions about the system's behaviour and environment

- Nodes capabilities and failure scenarios
- Communication channel between nodes and link failures
- Time and order

Abstraction models

- System model
- Failure model
- Consistency model

Abstraction models

- System model
- Failure model
- Consistency model

- Synchronous
  - Multiple processes coordinated using locks.
  - Message transmitting timeout exists.
  - Each process has an accurate clock.
- Asynchronous
  - No locks, you can’t rely on time.

Real-world systems are not synchronous

Partially synchronous at best

Abstraction models

- Crash failure - process died, can’t perform any computations

- System model
- Failure model
- Consistency model

Abstraction models

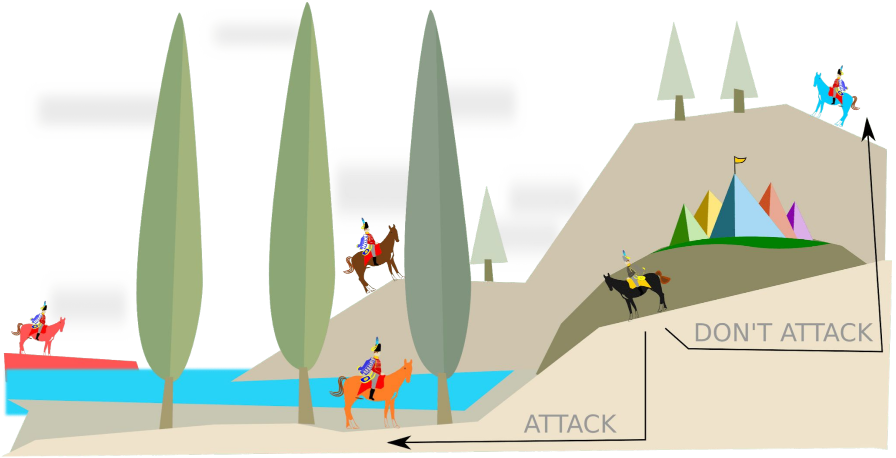
- System model
- Failure model
- Consistency model

- Crash failure - process died, can’t perform any computations
- Omission failure - a message is never delivered
- Link failure - a network failure

Abstraction models

- System model
- Failure model
- Consistency model

- Crash failure - process died, can’t perform any computations
- Omission failure - a message is never delivered
- Link failure - a network failure
- Byzantine failure (includes any combination of above)



Abstraction models

- System model
- Failure model
- Consistency model

- Strong consistency models
  - Guarantees that the order and visibility of updates is equivalent to a non-replicated systems → a lot of coordination
- Weak consistency models
  - Don’t guarantee the above.
- Eventual consistency

Consensus

When multiple processes agreed on the same value.

Consensus

We need consensus for

- Agreement and data integrity (*e.g leader election*)
- Prevent divergence between nodes (*e.g replication*)

Consensus. FLP impossibility result

Assumptions

- Nodes only failing by crashing
- Network is reliable
- No bounds for time (the system is asynchronous)

Consensus. FLP impossibility result

Assumptions

- Nodes only failing by crashing
- Network is reliable
- No bounds for time (the system is asynchronous)

No solution without letting the system stay undecided forever

Is this the end?

Failure detectors

- Is a separate process
- Detects process failures, replies with “OK”, “NOT OK” to incoming requests
- May lie

Failure detectors

- Completeness = each failure is detected
- Accuracy = no false positives

Failure detectors

- Completeness = each failure is detected
- Accuracy = no false positives
- Speed = how fast the failure was discovered
- Scale = equal load on each member

Failure detectors

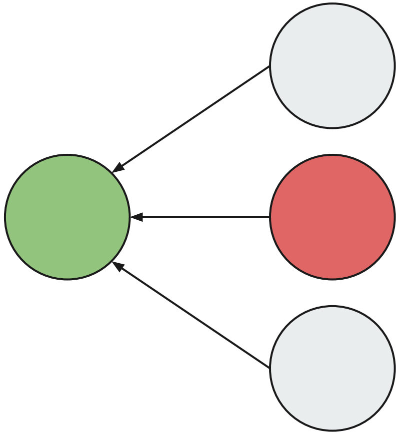
- Reliable failure detectors
  - Replies with “working” or “failed”
  - **Problems:** failure models limitations

Failure detectors

- Reliable failure detectors
  - Replies with “working” or “failed”
  - **Problems:** failure models limitations
- Unreliable failure detectors
  - Replies with “suspected” or “unsuspected”
  - **Problems:** we’d like to have reliable and accurate detectors, but it’s not realistic

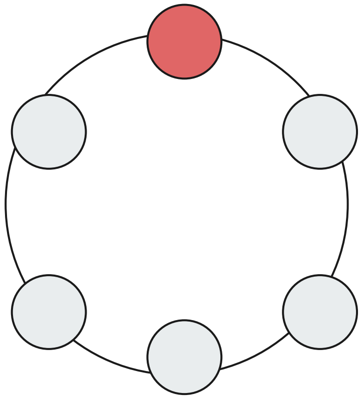
Failure detectors

- Centralized heartbeating



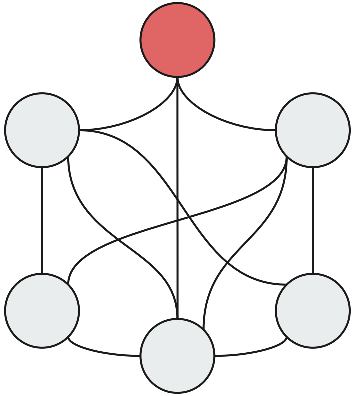
Failure detectors

- Centralized heartbeating
- Ring heartbeating



Failure detectors

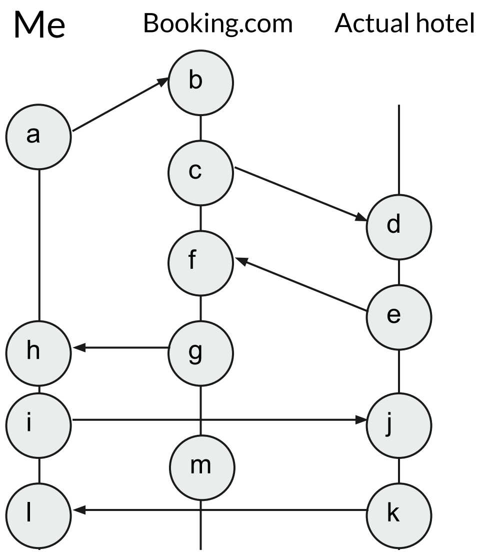
- Centralized heartbeating
- Ring heartbeating
- All-to-all heartbeating



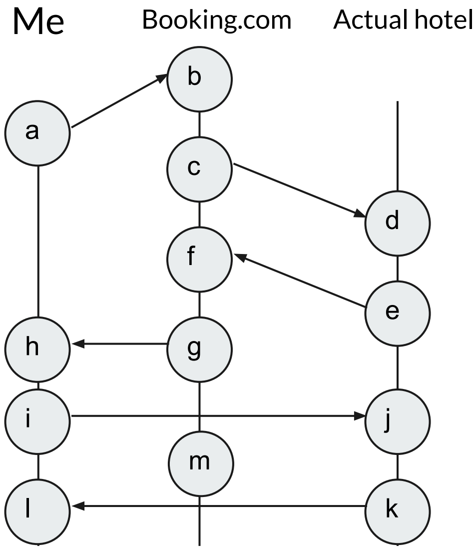
Consensus. Algorithms

- Paxos (Chubby, Cloud Spanner)
- Raft (consul, etcd, kubernetes)
- ZAB (ZooKeeper atomic broadcast, Cassandra, Hadoop/HDFS, Kafka)
- Proof-of-{NAME}
  - Work
  - Stake
  - Authority

Events ordering



Events ordering



- Connected events
  - a → h → i → l
  - b → c → f → g
  - d → e → j →k
  - a → b, e → f, i →k
- Transitive events
  - a → e
  - e → l
- Concurrent events
  - h ll m, i ll m, j ll m

Events ordering

- Total ordering
- Partial ordering

Events ordering

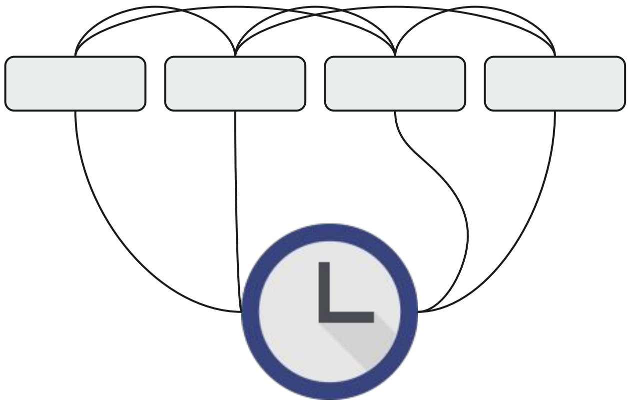
- Total ordering
- Partial ordering
- Total ordering driven by *partial ordering* using *logical clocks*

Time

- Global clock model
- Local clock model
- Logical clock model

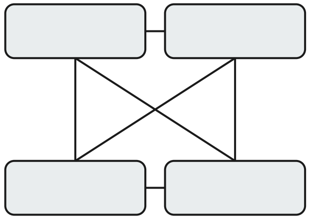
Time

- Global clock model
- Local clock model
- Logical clock model



Time

- Global clock model
- Local clock model
- Logical clock model



NTP

Time

- Global clock model
- Local clock model
- Logical clock model

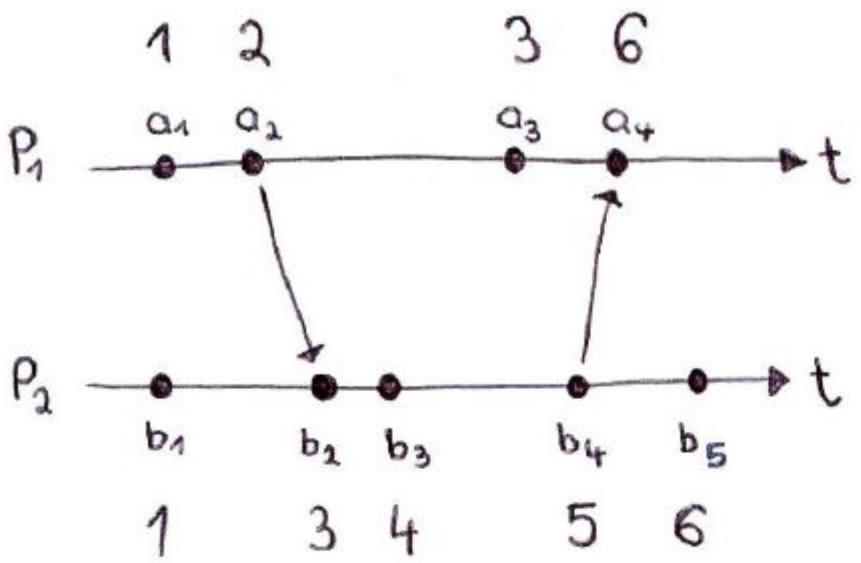
Lamport timestamps and logical clock

Sender:

```
time = 0
...
time = time + 1;
time_stamp = time;
send(message, time_stamp);
```

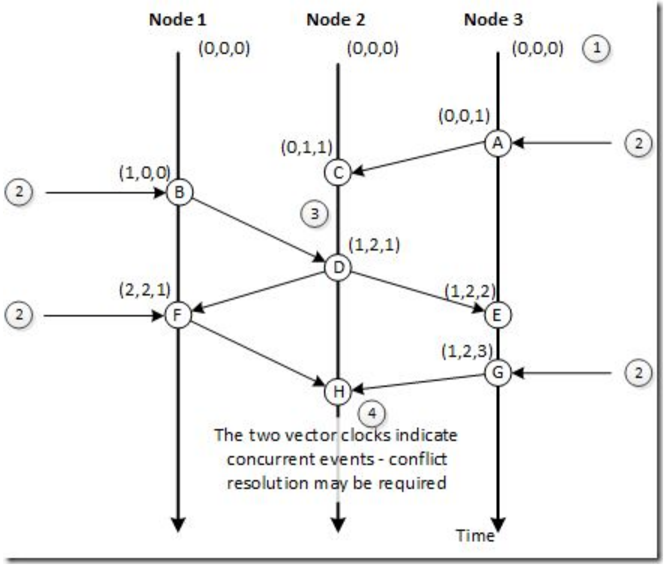
Receiver:

```
time = 0
...
(message, time_stamp) = receive();
time = max(time_stamp, time) + 1;
```



Lamport timestamps and logical clock

VClock = [
  Actor 1 → VClocksCounter: int
  Actor 2 → VClocksCounter: int
  ....
  Actor X → ...
]



Conflict-free Replicated Data Types

Conflict-free Replicated Data Types

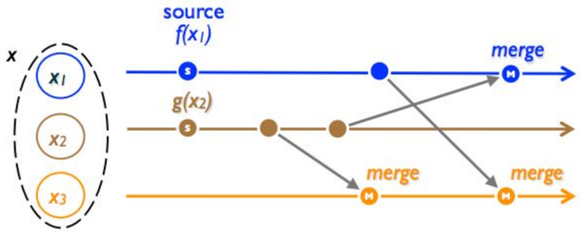
Eventual consistency without coordination

Conflict-free Replicated Data Types

- Commutative (CmRDT)
  - Operation-based: replicas propagate state by transmitting only update operations
  - Protocol must guarantee no duplicates and delivery only once
  - Operations are commutative *(a + b = b + a)*, associative *(a + (b + c) = (a + b) + c)*, but **not** idempotent *(a + a = a)*

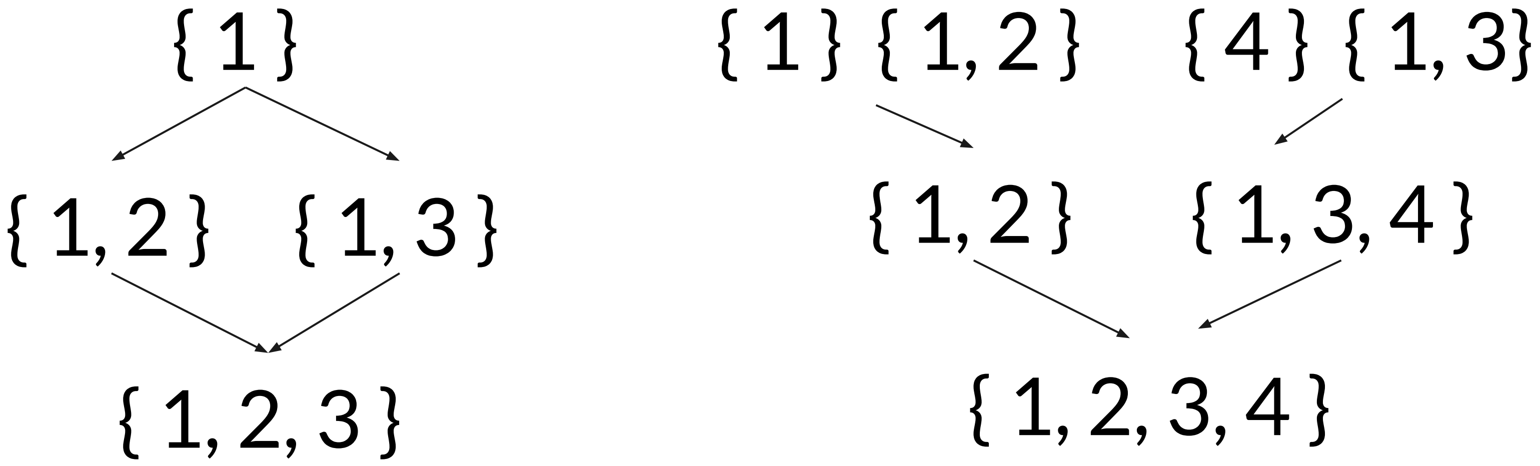
Conflict-free Replicated Data Types

- Commutative (CmRDT)
  - Operation-based: replicas propagate state by transmitting only update operations
  - Protocol must guarantee no duplicates and delivery only once
  - Operations are commutative *(a + b = b + a)*, associative *(a + (b + c) = (a + b) + c)*, but **not** idempotent *(a + a = a)*
- Convergent (CvRDT)
  - State-based CRDT
  - Every replica sends full state to others
  - Implements commutative *(a + b = b + a)*, associative *(a + (b + c) = (a + b) + c)* and idempotent *(a + a = a)* **merge** function



https://medium.com/@istanbul_techie/a-look-at-conflict-free-replicated-data-types-crdt-221a5f629e7e

Conflict-free Replicated Data Types



It's ok to see same writes many times

Conflict-free Replicated Data Types

- Grow-only counter
- Positive-negative counter
- Grow-only set
- Two-Phase set
- Last-Write-Wins-Element-Set
- Observed-Removed Set
- Sequences

https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

Conflict-free Replicated Data Types

Use cases

- Replication (Redis, Riak)
- Multi-master writes (Azure Cosmos DB)
- Real-time document editing (Google docs uses OT algorithms)

To sum up

- Can be more performant
- Can be more scalable
- Can be more resilient

To sum up

- Deep rabbit hole
- Hard to deal with
- But no other options