## CAP Theorem

- Consistency, availability, partitioning

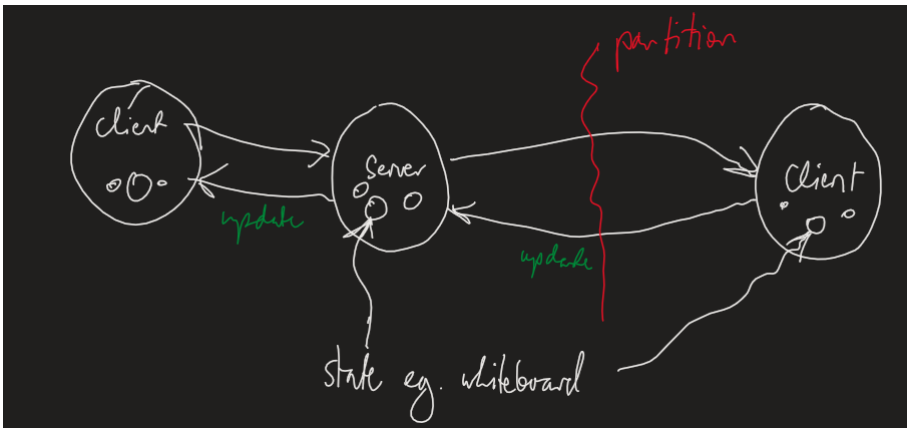


**Figure 1:** CAP Theorem

- **consistency:** every node agrees on current state
- **availability:** you go to get/set state, and the system is available to do that
- e.g. drawing on collaborative whiteboard
  - propagation is required to maintain consistency
  - ideally this would happen instantaneously, but clearly that's not possible
- **partitioning:** network problem creating disjoint regions of distributed system
  - with distributed systems there's always a chance of partitioning occurring
- **CAP Theorem:** when partitioning occurs, you need to choose between consistency and availability

## PACELC Theorem

- PAC: partitioning? then availability vs consistency
- ELC: else latency vs consistency

Wikipedia PACELC:

In case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C) (as per the CAP theorem), but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).

## Fallacies of Distributed Computing

From Wikipedia

The fallacies of distributed computing are a set of assertions made by L Peter Deutsch and others at Sun Microsystems describing false assumptions that programmers new to distributed applications invariably make.

The fallacies are:

1. The network is reliable;
2. Latency is zero;
3. Bandwidth is infinite;
4. The network is secure;
5. Topology doesn't change;
6. There is one administrator;
7. Transport cost is zero;
8. The network is homogeneous.

## Eventual Consistency

- Notes on *Eventual Consistency Today: Limitations, Extensions, and Beyond (Bailis and Ghodsi), 2013*
- **eventual consistency:** if no additional updates are made to a given data item, all reads will eventually return the same value
  - "eventually, all accesses return the last updated value"
  - very weak, but extremely useful
  - systems are strongly consistent most of the time, and are often faster than strongly consistent counterparts
- there are properties eventual consistency can never provide: there is a cost to remaining highly available and providing low latency
- even without partitions, a system that chooses availability over consistency benefits from low latency
- partition tolerance is not something you can trade-off: partitions happen, and when they do, you must choose between availability and consistency
- most of the time, availability wins over consistency

  - e.g. social network and timeline updates: you degrade user experience by preventing user from posting if you choose consistency

- **anti-entropy:** information exchange between replicas about what writes they have seen to ensure convergence towards consistency

  - **concurrent writes:** if these happen, the replicas deterministically choose a winning value
  - needs to be asynchronous to prevent replicas hanging when partitions occur
  - you don't want to return immediately on write, or you risk data durability. Better to trade off between durability and availability by returning after W replicas have acknowledged the write

- **safety:** nothing bad happens, every value that is read was, at some point, written to the database
- **liveness:** something good eventually happens, e.g. all requests eventually receive a response
- eventual consistency makes no safety guarantees, but purely liveness

**How eventual is eventual consistency?**

- **time:** how long will it take for writes to become visible to readers?
- **versions:** how many versions old will a given read be?
- **measurement:** how consistent is my store under given workload presently
- **prediction:** how consistent will the store be under a given configuration/workload
- **probabilistically bounded staleness:** expectation of recency for reads of data items

  - measure how far an eventually consistent store deviates from that of a strongly consistent store
  - e.g. "100ms after a write completes, 99.9% of reads will return the most recent version"
  - degree of inconsistency is determined by the rate of anti-entropy
  - can calculate expected consistency from: anti-entropy protocol, anti-entropy rate, network delay, local processing delay

**How to program under eventual consistency?**

- don't need to write corner cases for downed replicas/partitions: anti-entropy just stalls
- don't need to write complex code for coordination
- external compensation outside the system: proceed as though the value presented is the latest. When you turn out to be wrong, you need to compensate for any incorrect actions taken. Retroactive safety to restore

- guarantees
  - **trade-off:** need to weight benefit of weak consistency against cost per anomaly * rate of anomalies
  - cost of anomalies = cost of compensation
  - if the rate of anomalies is extremely low, you may forgo compensation entirely
- **data structures that avoid inconsistency:** consistency as logical monotonicity (CALM)
  - **monotonic program:** computes an ever-growing set of facts, and never retract facts that they emit
  - such programs can always be safely run on an eventually consistent store
  - CALM tells you where stronger coordination mechanisms are likely required
  - e.g. storing time series of stock market data vs storing latest value
  - overwrites, set deletion, counter resets, negation: not logically monotonic
  - **CALM captures ACID 2.0 (associativity, commutativity, idempotence, distributed)**
    - idempotence: you can call a function any number of times and get the same result
  - Commutative replicated data types e.g. increment-only counter
    - increment is commutative: it doesn't matter in which order two increments are applied
    - replicas understand semantics of increment instead of general purpose read/write which is not commutative
    - key property: separation of data store and application consistency concerns
  - CALM + CRDT: toolkit for consistency without concurrency control

**Can stronger guarantees be provided without losing benefits?**

- high availability can be retained while providing stronger guarantees, e.g. causality and ACID properties
- **causal consistency:** each processes writes are seen in order writes follow reads, useful in ensuring comment threads are seen in correct order without dangling replies