https://cw.fel.cvut.cz/b251/courses/b4m36ds2/start

Lecture 3

Basic Principles: CAP theorem, Consistency

Yuliia Prokop
prokoyul@fel.cvut.cz

6. 10. 2025

Based on the presentation of Martin Svoboda (martin.svoboda@matfyz.cuni.cz)

Czech Technical University in Prague, Faculty of Electrical Engineering

Lecture Outline

Different aspects of **data distribution**

- **Scaling**
  - Vertical vs. horizontal
- **Distribution** models
  - **Sharding**
  - **Replication**: master-slave vs. peer-to-peer architectures
- **CAP** properties
  - **Consistency**, **availability** and partition tolerance
  - ACID vs. **BASE guarantees**
- **Consistency**
  - Read and write quorums

Limitations of Traditional RDBMS at Scale

Why SQL databases struggle with massive scale

Scalability Limitations:

- Primarily vertical; horizontal scale is possible but complex
- Expensive hardware
- Single points of failure
- Limited by single machine resources

ACID Constraints:

- Global consistency overhead
- Distributed transactions costly
- Cross-datacenter challenges
- Performance vs consistency

Aspect	Traditional RDBMS	Modern Requirements
Data Volume	Gigabytes to Terabytes	Petabytes to Exabytes
Request Rate	Thousands/second	Millions/second
Global Users	Regional	Worldwide 24/7
Schema Changes	Planned downtime	Zero-downtime deployments

CAP Theorem

CAP Theorem

Assumptions

- Distributed system with **sharding and replication**
- Read and write **operations on a single aggregate** only

CAP properties

- Properties of a distributed system
- Consistency, Availability, and Partition tolerance **CAP theorem**

> In the presence of a network **partition**, a distributed system can choose either **consistency** or **availability**, but not both.

But, what these properties actually mean?

CAP Properties

Property	Formal Definition	Practical Meaning
Consistency	Linearizability: Operations appear to execute atomically	All reads return the most recent write
Availability	Every request receives a response (success or failure)	The system always responds, never times out
Partition Tolerance	System continues despite message loss between nodes	Works even when network splits occur

- Hardware failures are inevitable
- Network congestion causes effective partitions
- Slow networks trigger timeouts
- Geographic distribution increases partition probability

CAP Properties

- Every read and write on a given item/key behaves **as if it were executed atomically**.
- **Formally:** there is a single, global order of operations such that each operation appears to take effect **instantaneously at some point between its invocation and its completion** – as if all operations were executed sequentially on a single standalone node.
- **Practical consequence:** after a successful write, any subsequent read (on the same item) will return the updated value.
- Because any replica can serve read requests, writes must be replicated to a **sufficient set of replicas (e.g., a quorum)** before being acknowledged to maintain this strong consistency.
- Other, weaker consistency models also exist and will be discussed later.

CAP Properties

Availability

- **If a node is working, it must respond to user requests**
  - *A bit more formally…*
    Every read or write request successfully received by a non-failing node in the system must result in a response (success or failure), not be silently dropped.
    I.e., their execution must not be rejected

Partition tolerance

- **The system continues to operate even when two or more sets of nodes get isolated**
  - *A bit more formally…*
    The network is allowed to lose arbitrarily many messages sent from one node to another
- I.e. a connection failure must not shut the whole system down

CAP Theorem Proof

- **Proof by contradiction**
  - **Assume all three properties can be satisfied simultaneously**
  - Consider a network partition scenario
- **Partition scenario setup**
  - Network splits into two disjoint sets of nodes: G_1 and G_2
  - No communication possible between G_1 and G_2
- **Write operation on G_1**
  - Client writes to G_1, must be consistent across all replicas
  - G_2 cannot receive this update due to partition
- **Read operation on G_2**
  - If system is available, G_2 must respond to read requests
  - If system is consistent: G_2 must return the updated value

⚡ **Contradiction: G_2 cannot have updated value (violates C) but must respond (requires A)**

$C \land A \land P$ is impossible in distributed systems

Network Partition Scenarios

- **Complete partition**
  - Network splits into isolated groups
  - No communication between groups
- **Partial partition**
  - Some nodes can communicate; others cannot
  - Asymmetric partitions possible
- **Common causes of partitions**
  - Router/switch failures
  - Network congestion (appears as a partition)
  - Data center connectivity loss
  - Slow networks triggering timeouts
- **Some illustrative incidents include:**
  - AWS us-east-1 partition (2017)
  - Google Cloud networking outage (2019)
  - GitHub's network split (2018)

Consistency Spectrum

- **Strong consistency models**
  - Linearizability (strongest for a single operation/key)
  - Transactional models (Serializability / Snapshot Isolation)
  - Sequential consistency
  - Causal consistency
- **Weak consistency models**
  - Session consistency
  - Monotonic read/write consistency
  - Eventual consistency (weakest)
- **Consistency vs. Performance trade-off**
  - Stronger consistency → Higher latency
  - Weaker consistency → Better performance
- **Application requirements determine choice**
  - Banking: Strong consistency required
  - Social media: Eventual consistency acceptable
  - Collaborative editing: Causal consistency needed

Availability Measurement

- **Availability metrics**
  - Uptime percentage: 99.9%, 99.99%, 99.999% per year
  - Downtime per year: 8.76 hours, 52.56 minutes, 5.26 minutes
- **Factors affecting availability**
  - Mean Time Between Failures (MTBF)
  - Mean Time To Repair (MTTR)
  - Availability = MTBF / (MTBF + MTTR)
- **High availability techniques**
  - Redundancy and failover
  - Load balancing
  - Circuit breakers
- **CAP availability definition**
  - Every request receives a response
  - Different from uptime availability
  - About request handling, not system uptime

CAP Theorem Consequences

If **at most two properties** can be guaranteed…

- **CA** = **consistency + availability**
  - Traditional ACID **properties** are easy to achieve
  - Examples: **RDBMS**
    Any single-node system, but even clusters (at least in theory)
    - However, should the network partition happen, all the nodes must be forced to stop accepting user requests

CA: Consistency + Availability – only possible if **no network partitions occur**
(e.g., traditional RDBMS under normal conditions)

CAP Theorem Consequences

If **at most two properties** can be guaranteed…

- **CP** = **consistency + partition tolerance**
  - Other examples: distributed locking
- **AP** = **availability + partition tolerance**
  - New concept of **BASE properties**
  - Examples: Apache Cassandra, Apache CouchDB.
  - Other examples: web caching, DNS

In real-world environments, network partitions can and do occur. Distributed systems therefore **should be designed to tolerate partitions (P)** and then choose between C and A during a partition. Systems that sacrifice P effectively stop responding when a partition occurs.

CAP Theorem Consequences

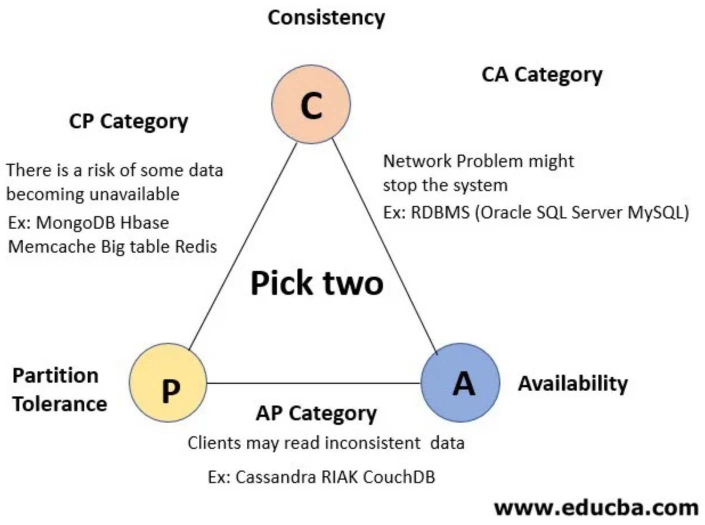
Design for partitions in clusters

- Why?
  - Because <u>it is difficult to detect network failures</u>
- Does this mean that only purely CP and AP systems are possible?
  - No…

The real meaning of the CAP theorem:

- *The real world does not need to be just black and white*
- **Partition tolerance** is a must,
  but we can **trade off consistency versus availability**
  - A relaxed consistency can bring a lot of availability.
  - Such trade-offs are not only possible,
    but often work very well in practice

CAP Theorem Consequences



ACID Properties

Traditional **ACID** properties

- **Atomicity**
  - Partial execution of transactions is not allowed (all or nothing)
- **Consistency**
  - Transactions bring the database from one consistent (valid) state to another
- **Isolation**
  - Transactions executed in parallel do not see uncommitted effects of each other
- **Durability**
  - Effects of committed transactions must remain durable

BASE Properties

New concept of **BASE** properties

- **Basically Available**
  - The system works basically all the time
  - Partial failures can occur, but there are no total system failures
- **Soft State**
  - The system is in flux (unstable), non-deterministic state
  - Changes occur all the time
- **Eventual Consistency**
  - Sooner or later the system will be in some consistent state

BASE is just a vague term, no formal definition was provided

- **Proposed to illustrate design philosophies at the opposite ends of the consistency-availability spectrum**

ACID and BASE

ACID

- Choose consistency over availability
- Pessimistic approach
- Implemented by traditional **relational databases**

BASE

- Choose availability over consistency
- Optimistic approach
- Common in **NoSQL databases**
- **Allows levels of scalability that cannot be acquired with ACID**

Historical move:

strong consistency → **eventual consistency**

Current trend in NoSQL:
eventual only → **tunable/stronger** consistency options

Don’t confuse CAP-C and ACID-C

Aspect	CAP-Consistency (C)	ACID-Consistency (C)
Definition	All nodes return the same (latest) value after a write; operations appear instantaneous (strong/linearizable consistency)	A transaction brings the database from one valid state to another, preserving integrity constraints
Scope	Replication across multiple nodes in a distributed system	Single database state and constraints within a transaction
Goal	Up-to-date and uniform view across replicas	No violation of schema rules or constraints during/after transaction
Typical trade-off	Must choose between C and A during partition	No direct CAP trade-off; ACID databases can still be “CAP-A or CAP-C” depending on setup

Consistency

Consistency

Consistency in general…

- **Consistency is the lack of contradiction** in the database
- However, it has many facets…
  - For example, we only assume atomic operations constantly manipulating just a single aggregate.
    But set operations could also be considered, etc.

Strong consistency is achievable in clusters **with appropriate replication/consensus (e.g., quorum/majority, consensus protocols)**, but **eventual consistency** might often be sufficient.

- One minute obsolete article on a news portal does not matter
- Even when an already unavailable hotel room is booked once again, the situation can still be figured out in the real world
- …

Consistency vs. Latency Trade-offs

- **Strong consistency costs**
  - Synchronous replication to a **quorum/majority** of nodes
  - Latency ≈ latency to the slowest node in the quorum
  - Example: 3 nodes, majority = 2, 100 ms each → ~100 ms latency
- **Weak consistency benefits**
  - Asynchronous replication
  - Latency = latency to a single node
  - Example: 3 nodes, 10ms local → 10ms total latency
- **Real-world measurements**
  - MongoDB: 5ms local read, 50ms strongly consistent read
  - Cassandra: 2ms eventual read, 20ms quorum read
- **Tunable consistency**
  - Applications can choose per-operation
  - Critical operations: strong consistency
  - Non-critical operations: eventual consistency

Consistency

Write consistency (update consistency)

- Problem: **write-write** conflict
  - Two or more write requests on the same aggregate are initiated concurrently
- **Context: multi-leader or leaderless architectures**
- Issue: lost update
- Solution:
  - **Pessimistic** strategies
    - **Preventing conflicts from occurring**
    - Write locks, …
  - **Optimistic** strategies
    - **Conflicts may occur, but are detected and resolved later on**
    - Version stamps, vector clocks, …

Consistency

Read consistency (replication consistency)

- Problem: **read-write** conflict
  - Write and read requests on the same aggregate are initiated concurrently
- Context: **both master-slave and peer-to-peer architectures**
- Issue: inconsistent read
- When not treated, **an inconsistency window** will exist
  - **Propagation of changes to all the replicas takes some time**
  - Until this process is finished, inconsistent reads may happen
  - Even the initiator of the write request may read wrong data!
    - Session consistency / read-your-writes / sticky session

Strong Consistency

How many nodes need to be involved to get strong consistency?

General rule: R + W > N (read and write quorums must intersect)

- **Write quorum**: $W > N/2$

  Idea: a majority write ensures only one write can succeed at a time

  W = number of nodes successfully acknowledged the write

  N = number of nodes involved in replication (replication factor)

- **Read quorum**: choose **R** such that **R + W > N** (e.g., **R > N − W**)

Idea: intersecting quorums ensure reads see the latest committed write

R = number of nodes participating in the read

If the retrieved replicas return different versions, resolve to the **latest committed version** (e.g., via version/timestamp) and then return it.

When a quorum is not attained → the request cannot be handled

Strong Consistency

Examples

Examples for replication factor $N = 3$

- Write quorum $W = 3$ and read quorum $R = 1$
  - All the replicas are always updated
  - $\Rightarrow$ we can read any one of them
- **Write quorum $W = 2$ and read quorum $R = 2$**
  - *Typical configuration, reasonable trade-off*

Consequence

- **Quorums can be configured to balance the read and write workload**
  - The higher the write quorum is required,
    the lower the required read quorum (and vice versa)

Measuring and Testing Consistency

- **Consistency testing challenges**
  - Distributed systems are non-deterministic
  - Race conditions are difficult to reproduce
  - Network delays affect observed behavior
- **Testing approaches**
- **Jepsen testing:** Simulate network partitions, clock skew
- **Linearizability checking:** Elle, Knossos tools
- **Property-based testing:** Generate random operations, check invariants
- **Consistency metrics**
- **Staleness:** Time lag between write and consistent read
- **Divergence:** Degree of inconsistency between replicas
- **Convergence time:** Time to reach consistency after partition heals

Measuring and Testing Consistency

- **Monitoring in production**
  - Track replica lag
  - Measure read-after-write latency
  - Alert on consistency violations
- **Tools and frameworks**
  - Hermitage: Database consistency testing
  - FoundationDB: Deterministic simulation
  - MongoDB: Built-in consistency monitoring

Bank:
Different Tasks = Different Decisions

Prefer CP semantics

- Account Balance
- Money Transfers
- Loan Approvals
- Transaction Processing
- Credit Limits

Prefer AP semantics

- Transaction History
- Product Recommendations
- Market News
- Branch Locator
- Customer Chat

🛍️ Online Store: Customer Journey

E-commerce System

Product Browsing
AP
Discovery over accuracy

Shopping Cart
Mixed
Session consistency

Inventory Check
CP
Prevent overselling

Payment Processing
CP
Financial accuracy

Order Confirm
CP
Customer trust

University: Academic vs Administrative

Academic Functions (CP)

- Student Grades
- Course Registration
- Tuition Payments
- Financial Aid
- Transcripts

Campus Services (AP)

- Library Search
- Campus Events
- Dining Menus
- Student Organizations
- News & Updates

University:
Critical Example – Course Registration

Problem: Popular Course with Limited Seats

'Machine Learning 101' - 30 seats, 200 students at 8 AM → Need fair, accurate registration

Solution: CP (Consistency Required): the system may sacrifice availability to avoid overbooking.

Trade-off: System slower during peak times, but zero overbooking

Universal Patterns Across Industries

Function Type	Bank	E-commerce	University	Pattern
Money/Financial	CP	CP	CP	Usually CP
User Identity	CP	Mixed	CP	Usually CP
Limited Resources	—	CP	CP	Usually CP
Content/Search	AP	AP	AP	Usually AP
History/Logs	AP	AP	AP	Usually AP
Recommendations	AP	AP	AP	Usually AP

Function type predicts CP/AP choice across all industries

How to Decide: CP or AP?

1

Identify Function Type

Financial? → Usually **CP**
Content? → Usually **AP**
Registration? → Usually **CP**

2

Analyze Error Impact

Money lost? → **CP** required
User frustration? → **AP** better
Legal issue? → **CP** required

3

User Expectations

Instant response? → **AP**
Accuracy critical? → **CP**
Both needed? → Hybrid

4

Design Implementation

CP: Transactions, locks
AP: Caches, replicas
Mixed: Different DBs

Lecture Conclusion

There is a wide range of options influencing…

- **Availability** – when nodes may refuse to handle user requests?
- **Consistency** – what level of consistency is required?
- **Latency** – how long does it take to handle user requests?
- **Durability** – is the committed data written reliably?
- **Resilience** – can the data be recovered in case of failures?

⇒ it's good to know these properties and choose the right trade-off