Weak Consistency Models

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

Overview

- Limitations of strong consistency models

- CAP rule and base methodology

- Partition-tolerant consistency models

Strong consistency models

- Recall, a data consistency model describes the expected behavior from a storage system when clients access data

  - When clients issue get()/put(), what values can get() read?

- We have discussed two strong consistency models

  - Linearizability: all processes execute operations in some total order, while preserving real-time ordering

    - All clients see same order of writes
    - All clients read latest data

  - Sequential consistency: all processes execute operations in some total order

    - All clients see same order of writes
    - Clients may read stale data

- Problem: total order limits performance and availability

Issues with strong consistency

- Operations need to communicate with majority of replicas

    - Linearizability: every read and write request

    - Sequential consistency: every write request

- Problem 1: low performance, high latency

    - With 5 replicas, every operation must communicate with 3 replicas

    - With leader-based algorithms, leader becomes bottleneck

- Problem 2: low availability

    - A stale or partitioned replica cannot process operations

    - If a majority of replicas are down, system is unavailable

- Takeaway: need weaker consistency models for higher performance and availability

CAP Rule and Base Methodology

Amazon, Google Experiments (2006)

- Amazon found that every 100ms in added page load time cost them 1% in sales

  - Today, would lose over 5 billion!

A sprint to render your web page!

- Google took 0.4s to generate a web page with 10 results, 0.9s to generate a page with 30 results

  - However, 0.5s delay caused a 20% drop in traffic!

- Conclusion: performance at scale determines revenue

  - And revenue shapes technology

  - An arms race began to improve cloud performance, availability

In the Cloud, Not Every Subsystem Needs the Strongest Guarantees

- Brewer argued that strong consistency delays response

  - For example, conflicting database updates
    can be forced into an agreed order,
    but this takes time and involves
    node-node communication,
    and with a network partition,
    the system provides no availability

- But cloud services make money only when they always provide fast response, so they must relax consistency

Examples of relaxing consistency

- Cache data in the application tier and serve it even when back-end database servers are unavailable, though cached data may be potentially stale

- Store a copy of data periodically (e.g., nightly) and use this read-only (potentially stale) data for analysis

- Allow delayed updates by enqueuing update tasks for later processing to amortize processing costs

- Guess the effects of updates and fix conflicts later, e.g., buy an item, if it was sold out, get refund

CAP

- Brewer captured tradeoff between speed of response and consistency by postulating a rule that connects consistency, availability and partition tolerance (CAP)

- Consistency: Updates performed in some system-selected order at all replicas. Queries return most up-to-date values. Users see a single system.

- Availability: System responds to every user request, even when some nodes are down.

- Partition Tolerant: System can tolerate network failures between subsystems. E.g., machines are partitioned into separate subnets because switch between subnets fails.

Cap Rule

- You cannot achieve all three together:
  - Consistency
  - Availability
  - Partition-Tolerance

- Popular interpretation: choose 2-out-of-3
  - CA: Assumes partitions don't occur, not realistic
  - CP: poor availability, users unhappy
  - AP: hard to program, possibly confusing to users

- None of these options are appealing!

Consistency **CA** Availability

CP   AP

Partition Tolerance
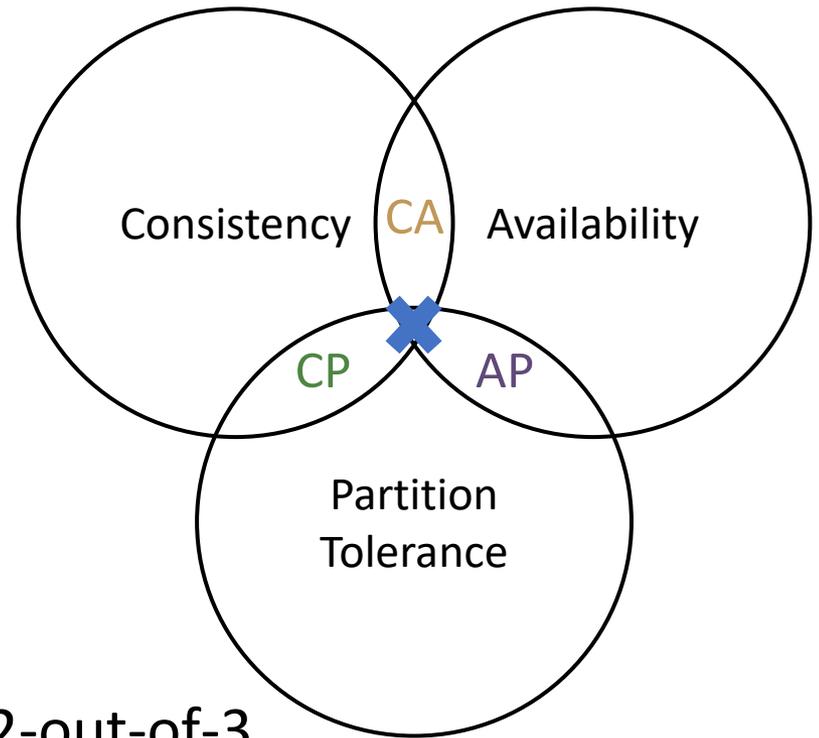
CAP rule in practice

- Partitions do occur, so systems must tolerate partitions

  - You cannot "not choose" partition tolerance …

  - But you can design systems to make them rare

- When there are no partitions,
  provide both consistency and availability

- When there is a partition, systems need to choose
  between consistency, availability

  - E.g., design systems that are best suited for application's
    consistency and availability needs

- When partition is fixed, restore consistency & availability

  - E.g., reconcile inconsistent replicas

BASE methodology

- BASE: set of rules for implementing CAP-based solutions

- Invented at eBay, adopted by Amazon, others

  - Basic Availability: provide continuous availability, despite failures or temporary inconsistency

  - Soft State: use state that can be regenerated (e.g., cached data) for efficiency

  - Eventual Consistency: assuming no further updates to an item, all users will eventually see the same value of the item

- Soft state and eventual consistency help recovery from failures, network partitions, data inconsistency, etc.
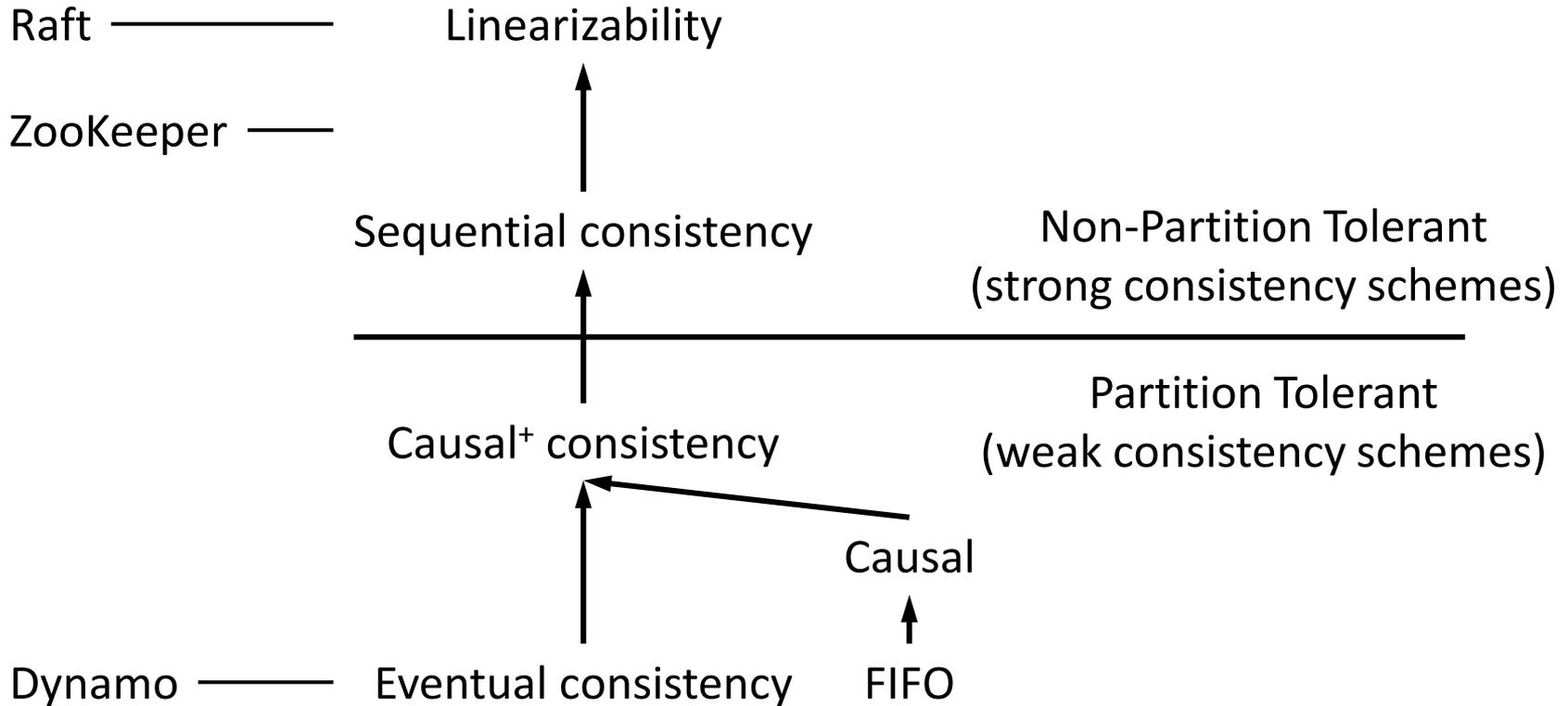
BASE example

- For example, if product photos rarely change, cache them, do not check for staleness with each cache access, let them expire after a few days or weeks

  - Avoids all cache refresh traffic

  - If a photo does change, you do see a stale photo, but this is rare

- BASE = "CAP in practice"
       = "Use CAP. You can clean up later."

- BASE encourages developers to think about when they need or do not need consistency

Partition-Tolerant Consistency Models

Consistency hierarchy

= stronger than

Raft ———————— Linearizability

ZooKeeper ———

Sequential consistency

Non-Partition Tolerant
(strong consistency schemes)

Partition Tolerant
(weak consistency schemes)

Causal⁺ consistency

Causal

Dynamo ——— Eventual consistency   FIFO

15

Eventual consistency

- All nodes execute operations (e.g., get, put) in any order

  - Allows partition tolerance

- Assuming no new updates to a data item, all accesses to that item will eventually return the last updated value

  $\Rightarrow$ Replicas must synchronize to converge to same state eventually

- Used in optimistically replicated systems

  - Weakest "reasonable" form of consistency for replicated data

  - Provides high availability and partition tolerance

  - But eventual convergence is not suitable for all apps

  - Later, we will look at Dynamo, an example of such a system

16

Understanding eventual consistency

- Reads, writes are performed at a replica without synchronizing with other replicas, so replicas may diverge

- Conceptually, updates are totally ordered "immediately", but replicas establish/learn the total order eventually when they synchronize updates with each other later

- Why is this model partition tolerant?

Understanding eventual consistency

- Applications need to handle out-of-order writes

    - Say total order of two updates is Wx1, Wx2

    - Initially, a replica receives Wx2 (out-of-order)

    - Application tentatively reads Rx2

    - During synchronization, when replica receives Wx1,
      Wx2 may need to be rolled back,
      invalidating application's Rx2 read

        - E.g., Wx1 is sale of last item of x, Wx2 is no longer possible

    - Application needs to be able to handle such tentative reads
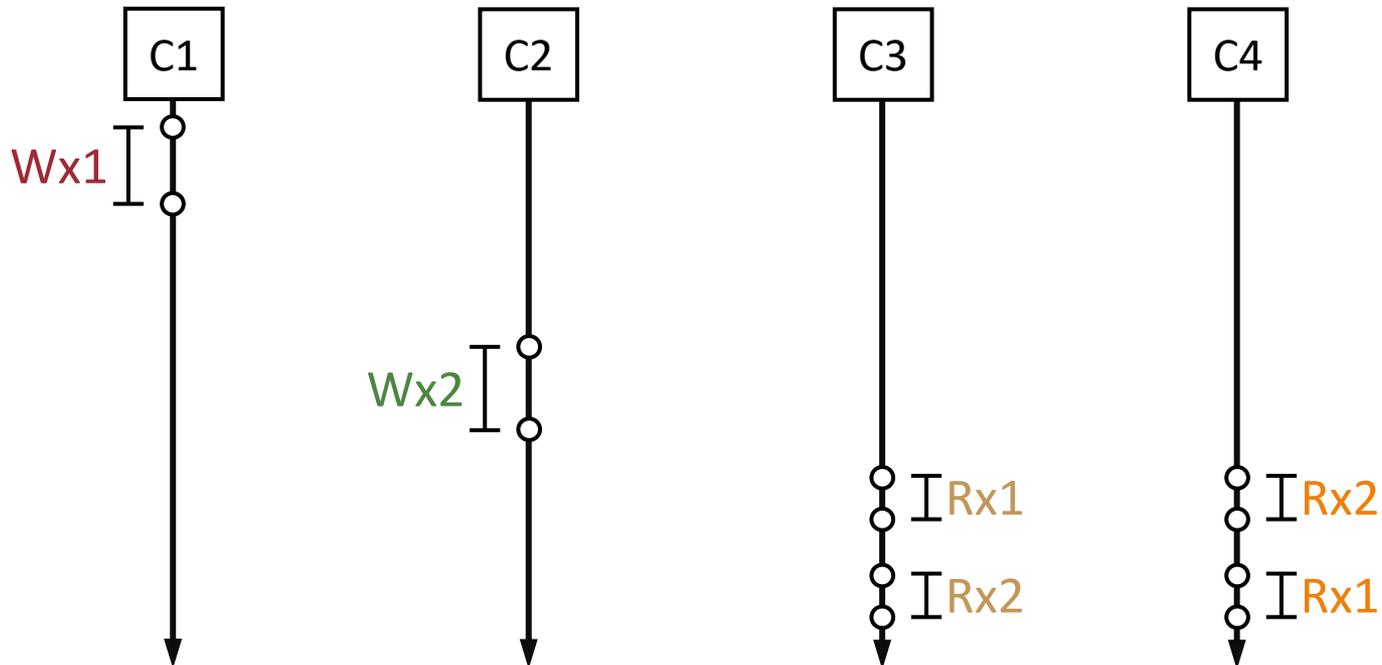
Causal+ consistency

- Causal consistency: all processes execute operations in an order that satisfies causality (happens-before)

  - Say Client 1 writes WxA, Client 2 reads RxA and then writes WyB

    - Then, WxA -> WyB (happens-before)

  - If a process reads RyB, then it cannot read RxA', if WxA' -> WxA

    - Otherwise, there would be cyclic dependency

- Implications

  ```
          RxA
  WxA ──────────→ WyB
   ↑      RyB      │
   │      RxA'     │
   └───────────────┘
  ```
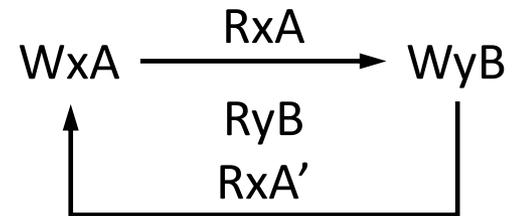
  - High availability: allows partition tolerance since causally related operations cannot occur across partitions

  - No guarantee of convergence: replicas can apply two non-causally-related events in different orders

- Causal⁺: data is eventually consistent also

Understanding causal consistency

- With causal consistency, there should be no cyclic dependencies among operations

- Are these operations causally consistent?

  - Yes: no dependency between Wx1 and Wx2, C3 and C4 can observe them in either order

Understanding causal consistency

- With causal consistency, there should be no cyclic dependencies among operations

- Are these operations causally consistent?

  - No: Wx1 happens before Wx2,
    C4 should not observe Rx2 and then Rx1
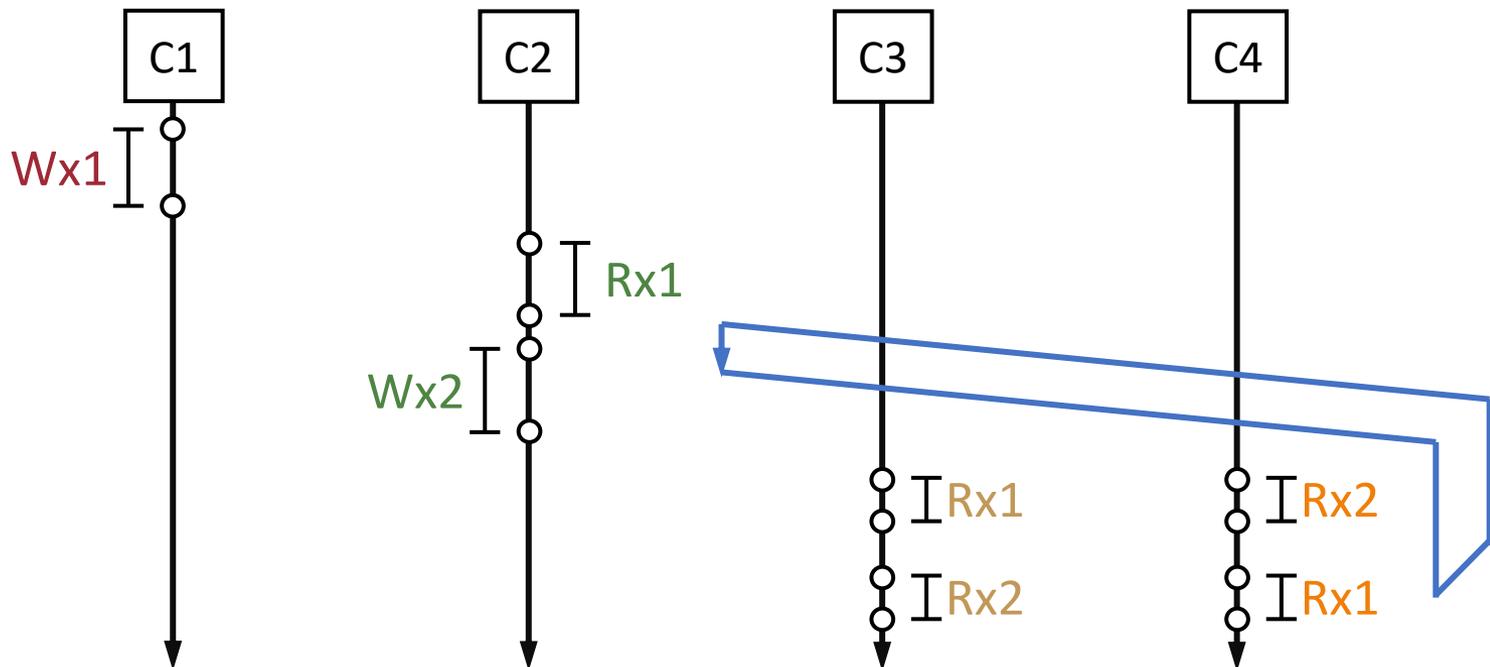
Session (client-centric) guarantees

- We have focused on consistency guarantees provided by a system to all users

- Session guarantees describe consistency guarantees provided by a system to a single client

  - No guarantees are made about accesses from different clients

- Motivation from mobile computing

  - Consider a mobile client that is connected to a replica

  - Client travels to another continent, connects to another replica

  - What guarantees can the client expect when it accesses its data?

Session (client-centric) guarantees

- Causal consistency ensures four session guarantees

  - Read-your-writes: a process's read can only be served by replicas that have applied all previous writes of the process

  - Monotonic writes: a process's write is only applied on replicas that have applied all previous writes of the process

  - Monotonic reads: a process's read can only be served by replicas that have applied all previous writes observed (read) by the process (i.e., reads don't go backwards)

  - Writes-follow-reads: a process's write is only applied on replicas that have applied all previous writes observed (read) by the process (i.e., causal writes)

- These guarantees become important when client sessions switch replicas (otherwise, trivially satisfied)

FIFO consistency

- FIFO consistency ensures the first three session guarantees: read-your-writes, monotonic writes and monotonic reads

- Writes across processes that are causally related may be performed in any order

Comparing consistency models

- Consider a message board, e.g., Piazza

    - A user posts a new message

    - Users reply to the post

Linearizability	Users sees the post and all replies in same real-time order
Sequential	Users sees the post and a prefix of all replies in same order, replies may not be real-time order
Causal	Users sees the post before replies, but may see replies in different order
FIFO	Users read messages from each user in order but not across users, so may see replies before post

Conclusions

- Strong consistency models such as linearizability and sequential consistency ensure total order of operations

  - Pros: Make it easier to write applications

  - Cons: Limit performance and availability under partitions

- CAP rule and base methodology

  - In the cloud, fast response and partition tolerance is critical

  - Base: continuous availability, cache data, eventual consistency

- Causal and eventual consistency models

  - Pros: Improve performance and availability (partition tolerance)

  - Cons: Make it harder to write applications, e.g., users can see temporarily inconsistent data, replicas need reconciliation