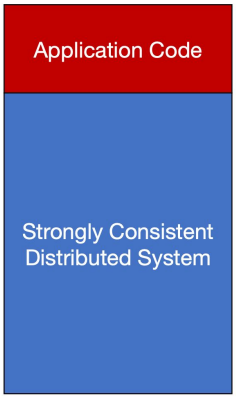
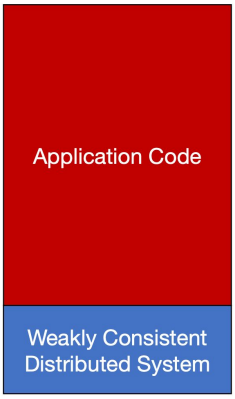
COS 316 Precept #9

Consistency

Stronger vs Weaker Consistency





Strongly Consistent:

- All nodes in the system see the same data at the same time
- Characteristics:
  - Synchronization
  - “Immediate” consistency

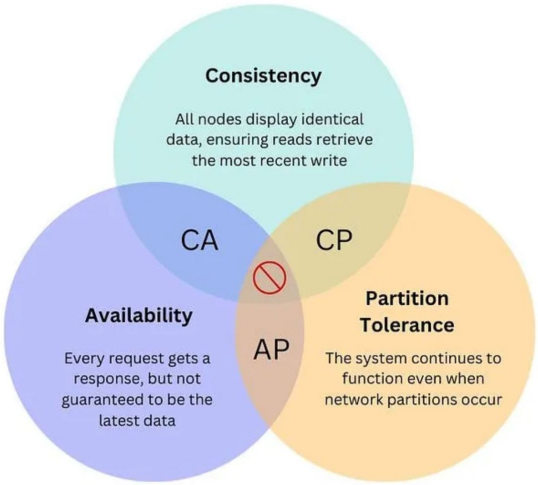
Weakly Consistent:

- May take time for all nodes to converge to the latest state, or even not finally converge
- Characteristics:
  - Asynchronization

- Let’s dive into one of the fundamental concepts in distributed systems: consistency models
- Distributed systems offer varying levels of consistency guarantees
- Some distributed systems are strongly consistent
  - The characteristics for strongly consistent distributed system is every node is synchronized on the state. Once an entity writes, it will be reflected to any other entity.
- Besides strong consistency, there are also other distributed systems which are not shooting for high level of consistency guarantee
  - Usually, those systems would take some time for all the nodes to converge to the latest state or even not converge.
  - One of the characteristics is about asynchronization. Different nodes may see different states/order.

- Usually, there would be tradeoffs between the consistency and the performance.

There is no free lunch – CAP theorem



- This is called an impossibility result, which places constraints on the properties that a distributed system can have
- The CAP theorem defines three properties of a distributed system
    - Strong consistency, which has been discussed extensively in lecture
    - Availability, which essentially means that requests are answered by the system
    - And Partition tolerance, which means that the system continues to function even when parts of the system aren't able to communicate with each other over the network
- It goes on to say that any distributed system can only have 2 of these properties at a time
- The reason why is relatively intuitive:
    - If a system is strongly consistent and always responds to requests, a network partition will prevent replicas from communicating with each other. If the system then wants to maintain strong consistency, it needs to stop responding until replicas can communicate. If instead it wants to maintain high availability, then some of those replicas will be allowed to diverge, breaking consistency.

Linearizability: “Appears to be a single machine”

Order preserves the real-time ordering between operations

- If operation A completes before operation B begins, then A is ordered before B in real-time
- If neither A nor B completes before the other begins, then there is no real-time order
    - (But there must be some total order)

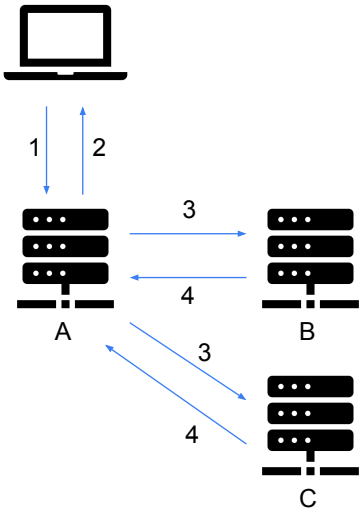
Linearizability is a form of strong consistency.

Example

- ETCD: distributed key-value store. Implemented using RAFT.

- One of the example is ETCD, which is a distributed key-value store. It is implemented using RAFT, which is a distributed consensus algorithm

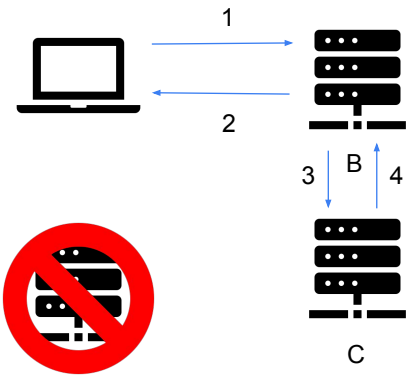
A broken protocol



1. Client sends operation to replica A
2. A executes operation and returns result to client
3. A sends operation to B and C
4. B and C execute operation and send acknowledgement to A

- Let's look at an example of a protocol that's meant to provide fault tolerance, but in the process breaks linearizability
- In this setup, there's a client that communicates with a system by sending operations and receiving the results of those operations
- There is one replica A, that's in charge of executing all operations. One of the other replicas, B and C, takes over if the first replica fails
- When A receives an operation, it replicates it asynchronously

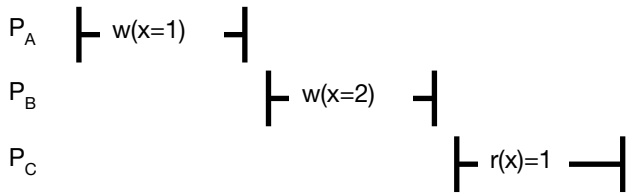
A broken protocol



1. Client sends operation to replica B
2. B executes operation and returns result to client
3. B sends operation to C
4. C executes operation and send acknowledgement to B

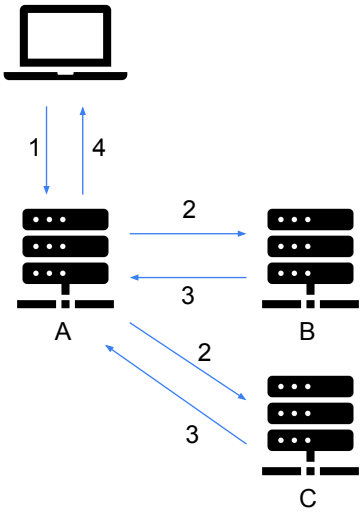
- Now, let's assume that A has failed. The system fails over to B as the lead replica, which is now responsible for communicating with the client
- What if A fails after 2 but before 3?
- This would mean that the client receives the result of an operation that hasn't been replicated yet
- So when the system fails over to replica B, B will have no knowledge of this operation
- This is an example of how linearizability is violated
- Let's look at a non-linearizable history that would be possible because of this

Non-Linearizable History



- This is an example of a non-linearizable execution that this flawed system could produce
- In this case, replica A would fail before the write of 2 has been replicated, causing none of the other nodes to be aware of that operation

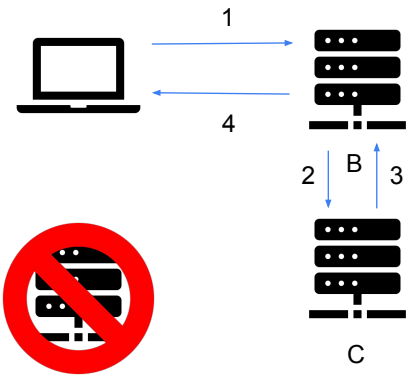
Fixed Protocol



1. Client sends operation to replica A
2. A sends operation to B and C
3. B and C execute operation and send acknowledgement to A
4. A executes operation and returns result to client

- In this example, replication is synchronous, meaning that it happens before the lead replica returns to the client
- Now, there is no way for a client to be observe the completion of an operation without the other replicas having observed it as well

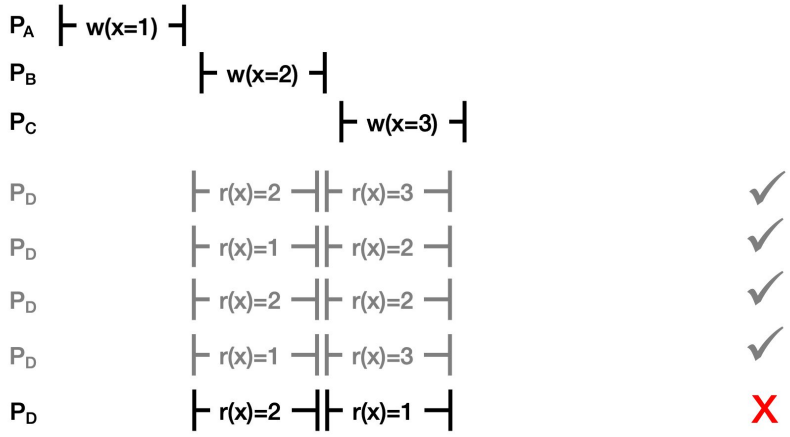
Fixed Protocol



1. Client sends operation to replica B
2. B sends operation to C
3. C executes operation and sends acknowledgement to B
4. B executes operation and returns result to client

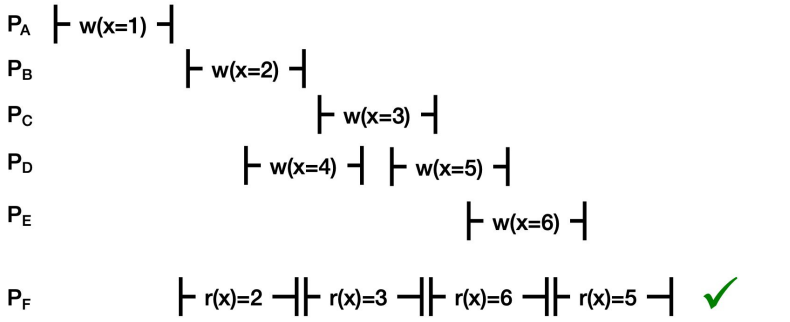
- This is the same example from earlier, with the fixed protocol

Example:



- Let's revisit some examples from lecture

Example:



$w_1, w_2, r_2, w_4, w_3, r_3, w_6, r_6, w_5, r_5$

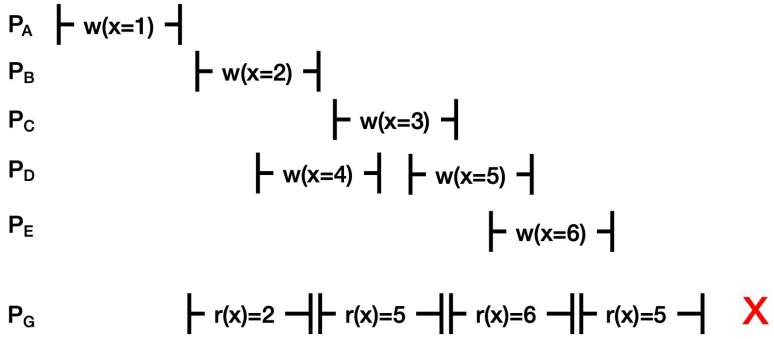
OR

$w_1, w_4, w_2, r_2, w_3, r_3, w_6, r_6, w_5, r_5$

OR

$w_1, w_2, r_2, w_3, r_3, w_4, w_6, r_6, w_5, r_5$

Example:



Causal+ Consistency

1. Writes that are potentially causally related must be seen by everyone in the same order.
2. Concurrent writes may be seen in a different order by different entities.
    a. Concurrent: Writes not causally related

Example:

Node A: write a post (Event 1), then delete that (Event 2)

Node B: write another post (Event 3)

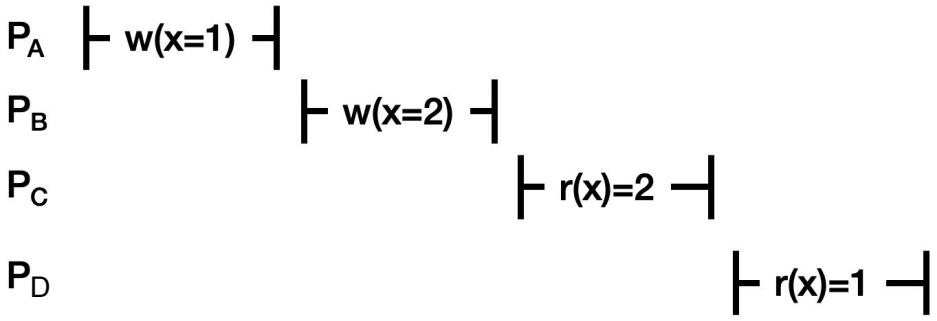
Causality: (Event 1 -> Event 2)

Other nodes may see different order of events, which can be

- Event 1, Event 2, Event 3
- Event 3, Event 1, Event 2
- Event 1, Event 3, Event 2
- **But not** Event 2, Event 3, Event 1

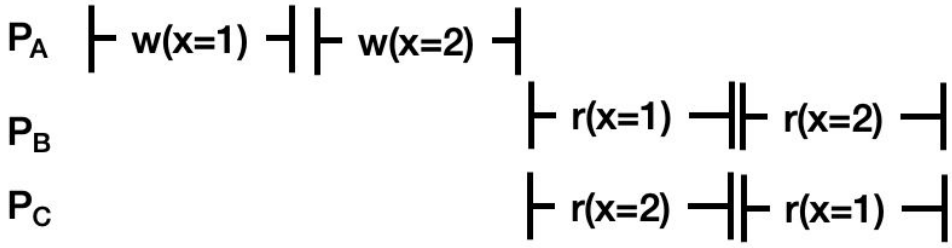
- Causal+ consistency is weaker than Linearizability.
- Causal+ consistency only guarantees the same order for **causally related writes**.
    - For concurrent writes, different entities may have different order. This flexibility allows better performance by reducing coordination overhead.
- For every node, Event 1 should always appear before event 2

Example:



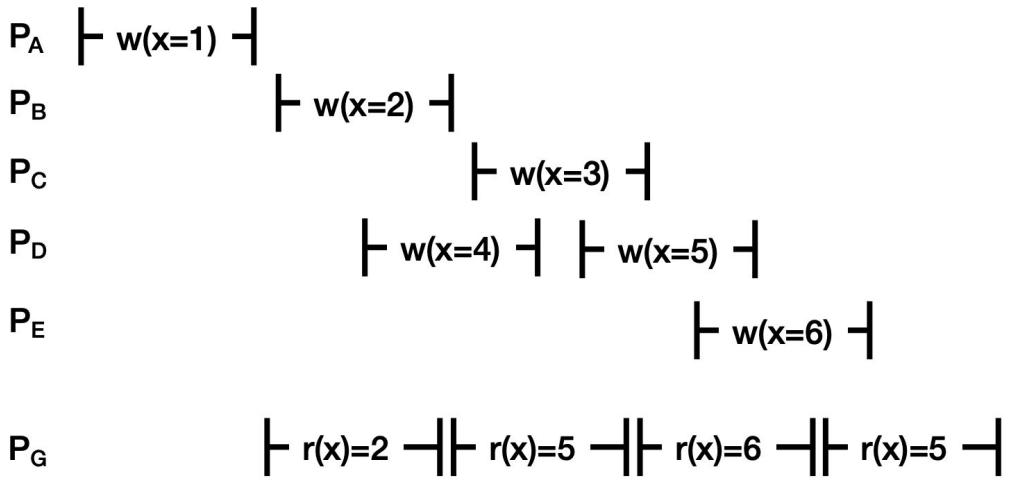
- Is this linearizable? No
- But it is causally consistent; the only orderings a causally consistent system has to uphold in this case is (w1, r1) and (w2, r2), basically making sure that ‘read x happens before write x’. As long as these orders are upheld, a causally consistent system can choose to order/observe any individual operations in any way.

Example:



Processes B and C are seeing different orders of the causally related writes (w1 and w2; causally related since they're on the same process), hence this history isn't causally consistent.

Example:



- This is neither linearizable or causally consistent; there's no way to order the causally related operations w5, r5, r5, w6, and r6 without breaking the logical order (we can't read 5 after writing 6 to x). If the r(x)=6 operation was on another separate process, then suddenly we will have an ordering that respects all necessary happens-before relationships.

Eventual consistency

If update stops, all the nodes finally reach the latest state

Prioritize performance (such as low latency, improved scalability)

Example:

- NoSQL database
- CDN (Content deliverable networks)

- Eventual consistency is weaker than causal+ consistency
- It only guarantees that all the nodes would finally reach the latest state. There is no guarantee about the order or timing of convergence.
- Usually, distributed systems leveraging eventual consistency is shooting for lower latency and better scalability
    - NoSQL refers to “Not only SQL”. It is a database to store data in non-relational format. It can handle large volumes of unstructured or semi-structured data. It is known for being performant and highly scalable.
    - CDN has to respond to users’ request fast (low latency) by sacrificing the guarantee to always provide the latest content.