Concurrency Distributed

COS 316 Precept 7

Outline

- Lamport Clock
- Totally Ordered Multicast
- Consistency Models

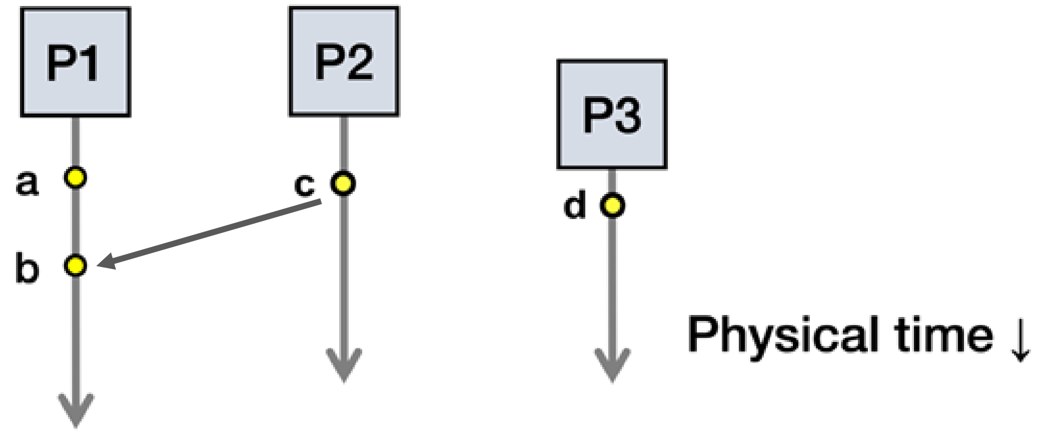
Outline

- Lamport Clock
- Totally Ordered Multicast
- Consistency Models

Staying Synchronized

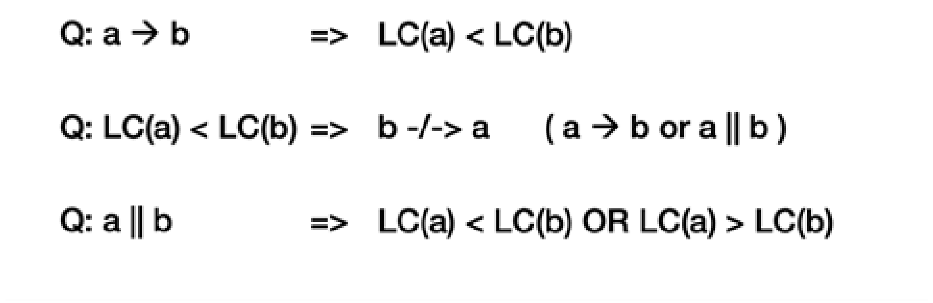
- On a single machine, its wall clock is sufficient
- Matches intuition
- In a distributed setting, each machine's clock may differ from the other machines
    - There is no longer an objective time
- Need some other mechanism to order events

Causality and Happens-Before



Lamport Clocks

- Provide a way to totally order events (when used with some tie-breaking mechanism)
- LC(A) < LC(B) => B -/-> A

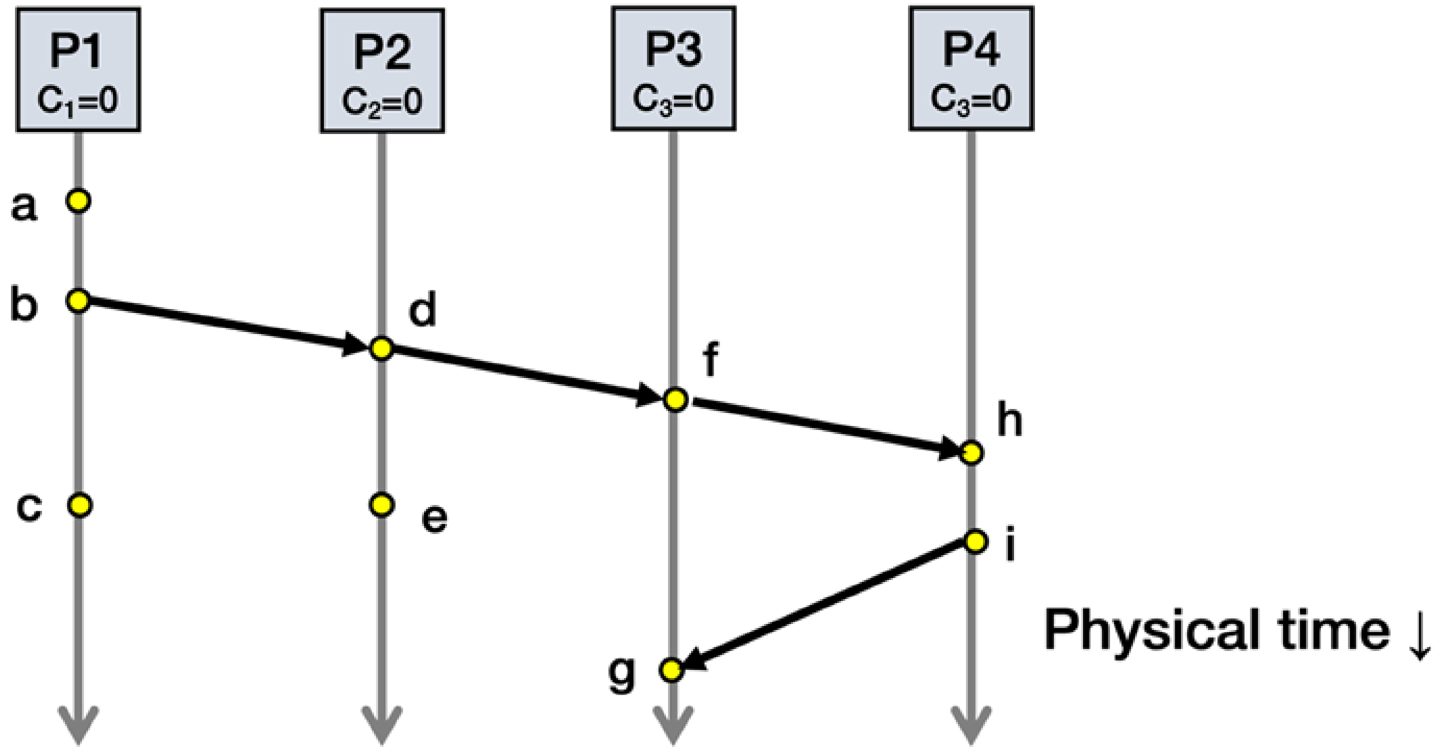


Q: a → b	=>	LC(a) < LC(b)	
Q: LC(a) < LC(b)	=>	b -/-> a	(a → b or a || b)
Q: a || b	=>	LC(a) < LC(b) OR LC(a) > LC(b)	

Lamport Clock Algorithm

1. Before executing an event b, Ci = Ci + 1:
2. Set event time C(b) ← Ci
3. Send the local clock in the message m
4. On process Pj receiving a message m:
5. Set Cj and receive event time C(c) ← 1 + max{ Cj, C(m) }

Lamport Clock Example



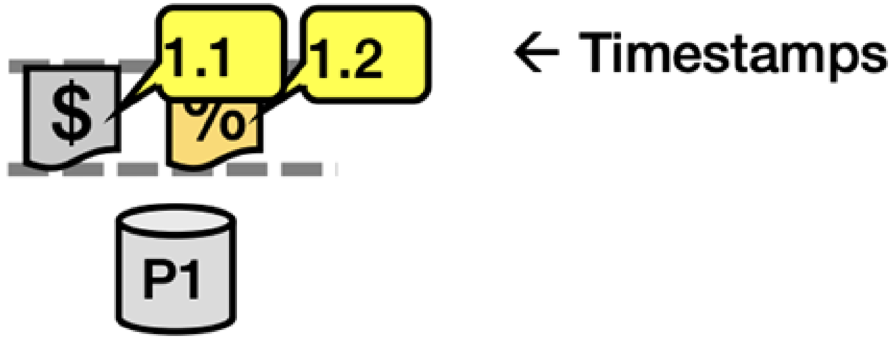
Outline

- Lamport Clock
- Totally Ordered Multicast
- Consistency Models

Totally-Ordered Multicast

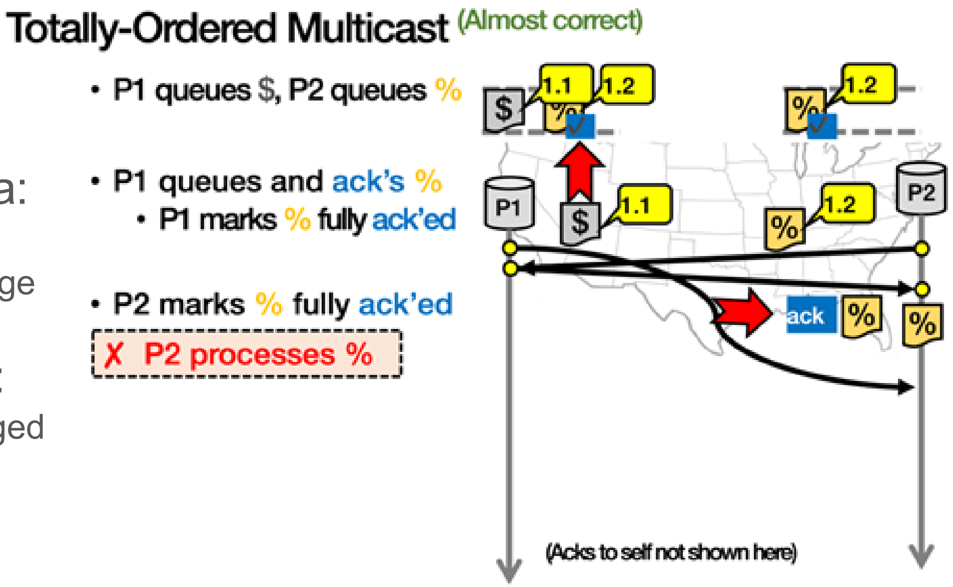
- Goal: All sites apply updates in (same) Lamport clock order
- Client sends update to one replica site j
    - Replica assigns it Lamport timestamp Cj . j
- Key idea: Place events into a sorted local queue
    - Sorted by increasing Lamport timestamps

Example: P1's local queue:



Totally-Ordered Multicast (Incorrect)

1. On receiving an update from client, broadcast to others (including yourself)
2. On receiving an update from replica:
   a. Add it to your local queue
   b. Broadcast an acknowledgement message to every replica (including yourself)
3. On receiving an acknowledgement:
   a. Mark corresponding update acknowledged in your queue
4. Remove and process updates everyone has ack'ed from head of queue



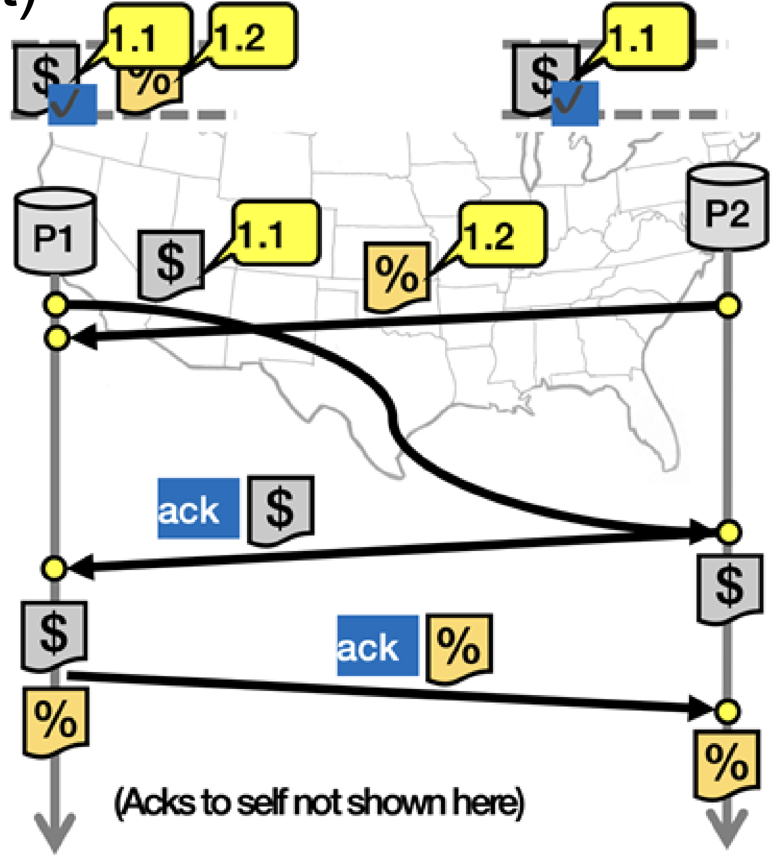
Totally-Ordered Multicast (Almost correct)

- P1 queues \$, P2 queues %
- P1 queues and ack's %
  - P1 marks % fully ack'ed
- P2 marks % fully ack'ed

✗ P2 processes %

Totally-Ordered Multicast (Correct)

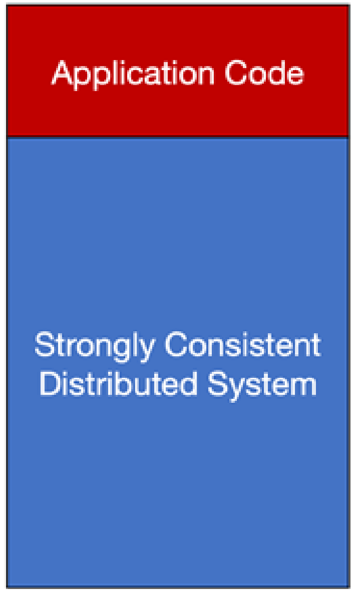
1. On receiving an update from client, broadcast to others (including yourself)
2. On receiving an update from replica:
   a. Add it to your local queue
   b. Broadcast an acknowledgement message to every replica (including yourself) only from the head of the queue
3. On receiving an acknowledgement:
   a. Mark corresponding update acknowledged in your queue
4. Remove and process updates everyone has ack'ed from head of queue

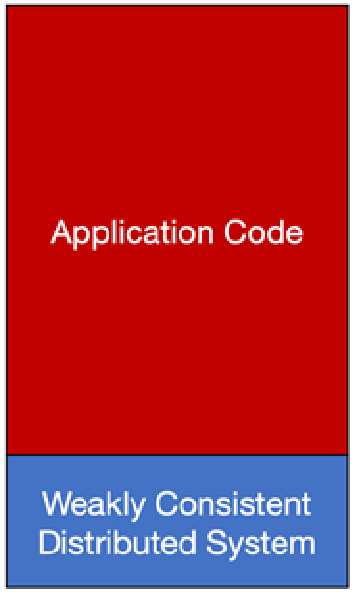


Outline

- Lamport Clock
- Totally Ordered Multicast
- Consistency Models

Stronger vs Weaker Consistency





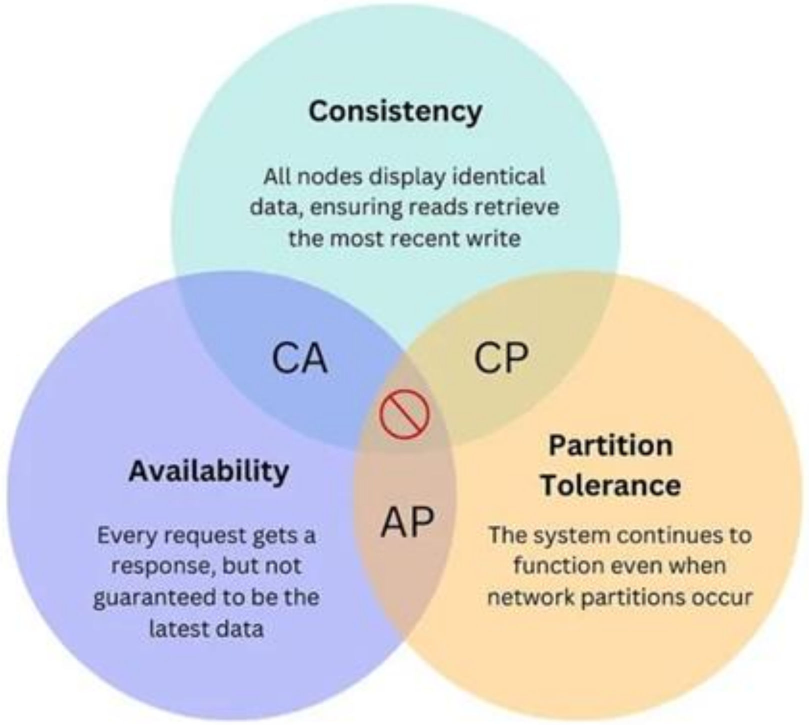
Strongly Consistent:

- 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

There is no free lunch – CAP theorem


Consistency
All nodes display identical data, ensuring reads retrieve the most recent write
CA
CP
AP
Availability
Every request gets a response, but not guaranteed to be the latest data
Partition Tolerance
The system continues to function even when network partitions occur

Linearizability: “Appears to be a single machine”

Order preserves the real-time ordering between operations

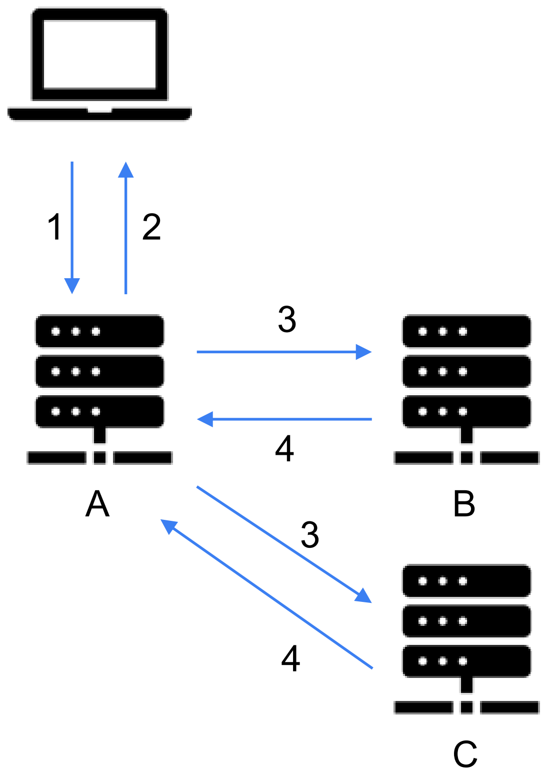
- If operation A completes before operation B begins, then A is ordered before B in real-time
- If neither A nor B completes before the other begins, then there is no real-time order
  - (But there must be some total order)

Linearizability is a form of strong consistency.

Example

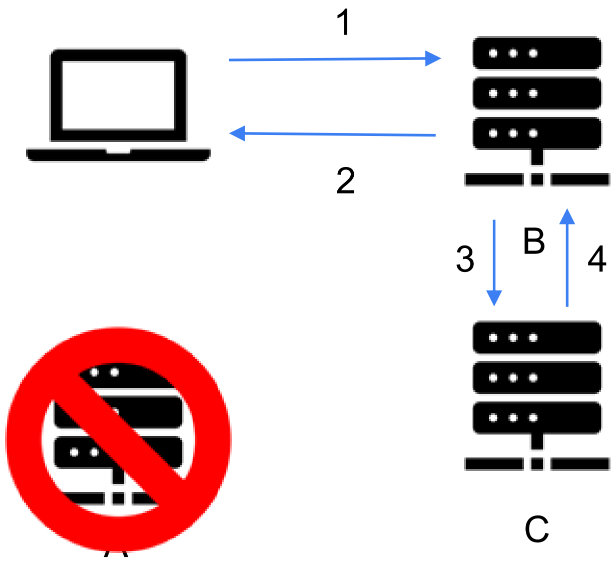
- ETCD: distributed key-value store. Implemented using RAFT.

A broken protocol



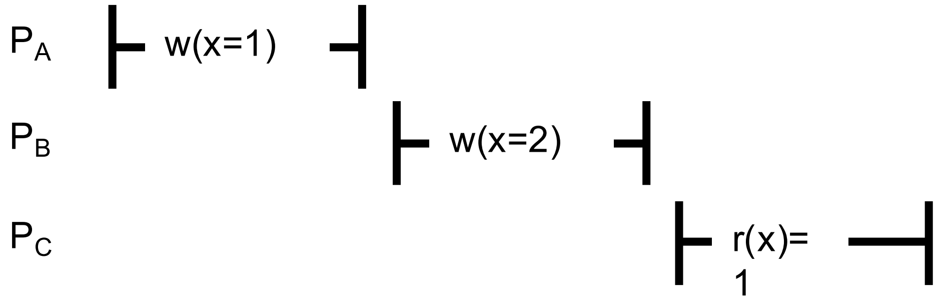
1. Client sends operation to replica A
2. A executes operation and returns result to client
3. A sends operation to B and C
4. B and C execute operation and send acknowledgement to A

A broken protocol

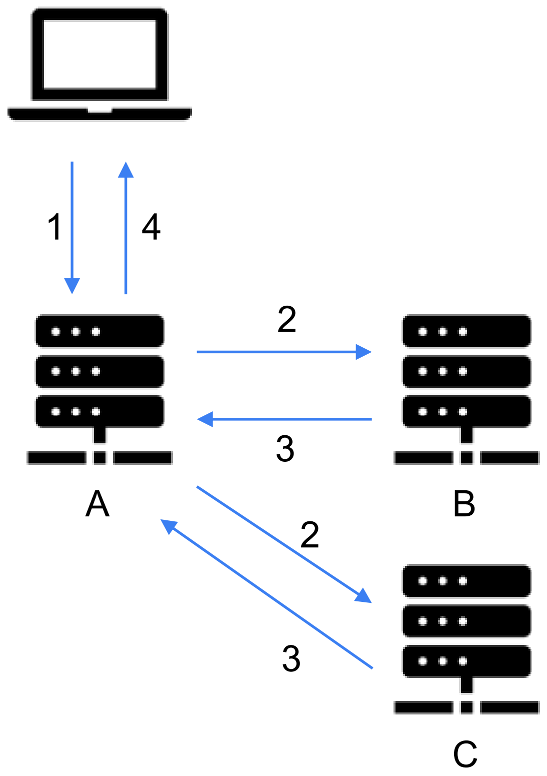


1. Client sends operation to replica B
2. B executes operation and returns result to client
3. B sends operation to C
4. C executes operation and send acknowledgement to B

Non-Linearizable History

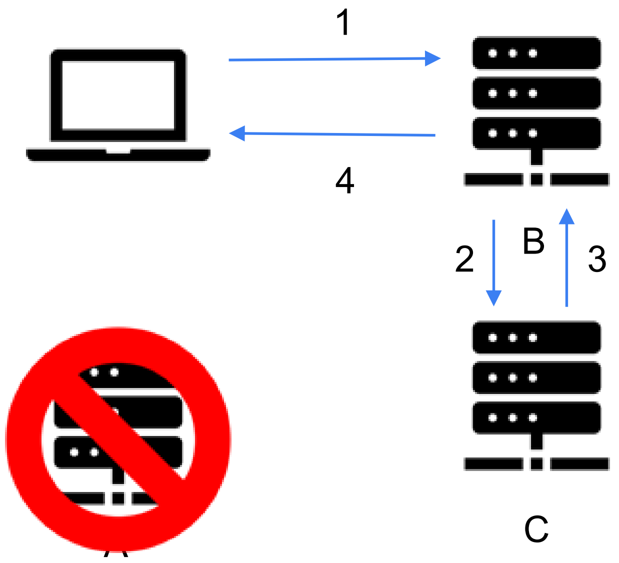


Fixed Protocol



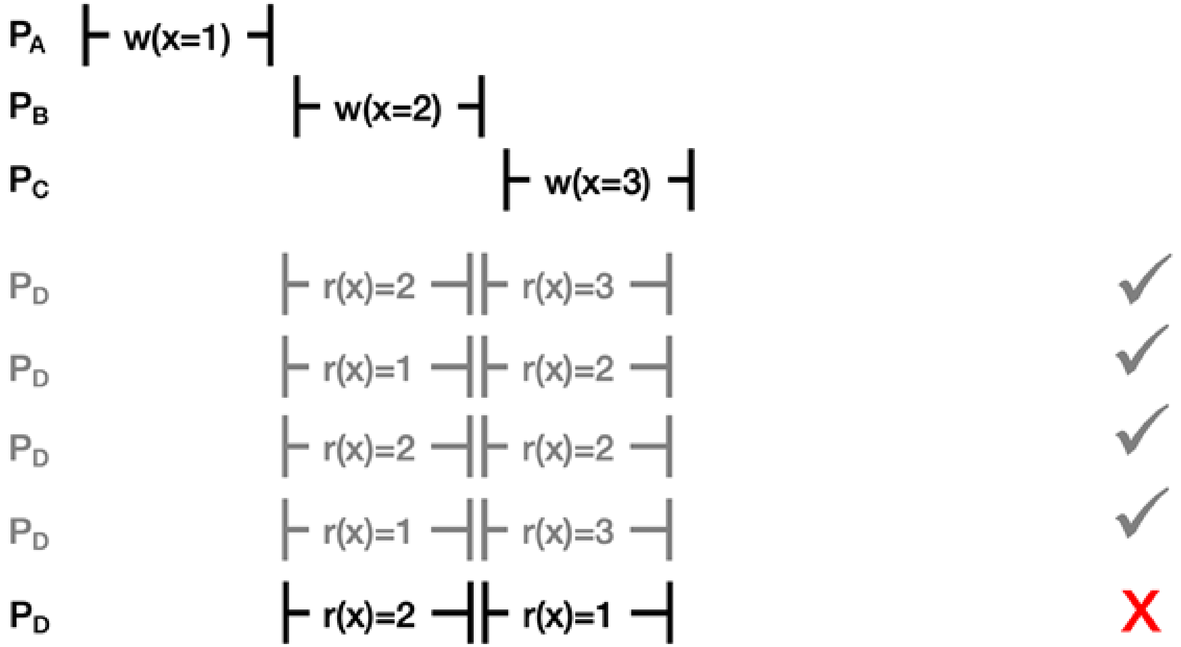
1. Client sends operation to replica A
2. A sends operation to B and C
3. B and C execute operation and send acknowledgement to A
4. A executes operation and returns result to client

Fixed Protocol

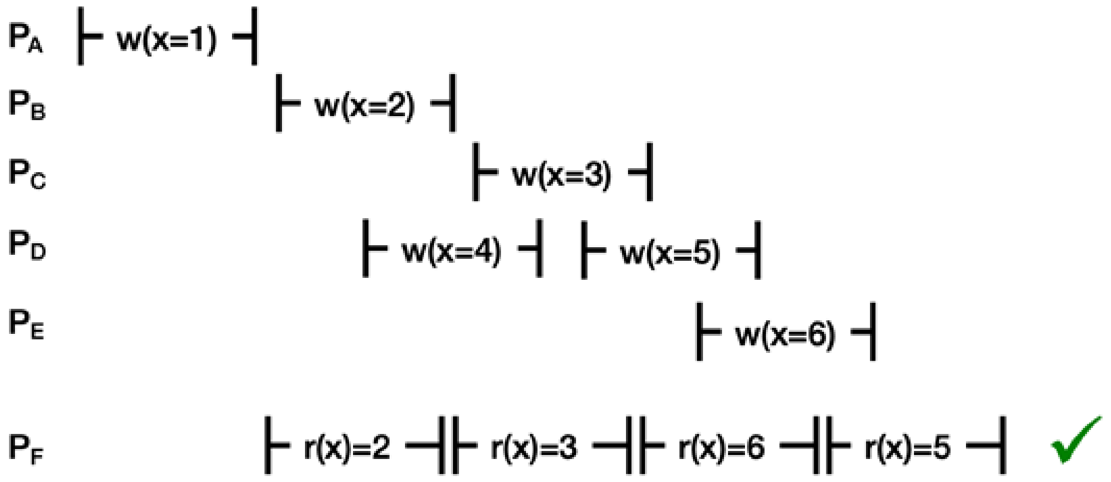


1. Client sends operation to replica B
2. B sends operation to C
3. C executes operation and sends acknowledgement to B
4. B executes operation and returns result to client

Example:


P_A w(x=1)
P_B w(x=2)
P_C w(x=3)
P_D r(x)=2 r(x)=3 ✓
P_D r(x)=1 r(x)=2 ✓
P_D r(x)=2 r(x)=2 ✓
P_D r(x)=1 r(x)=3 ✓
P_D r(x)=2 r(x)=1 X

Example:



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

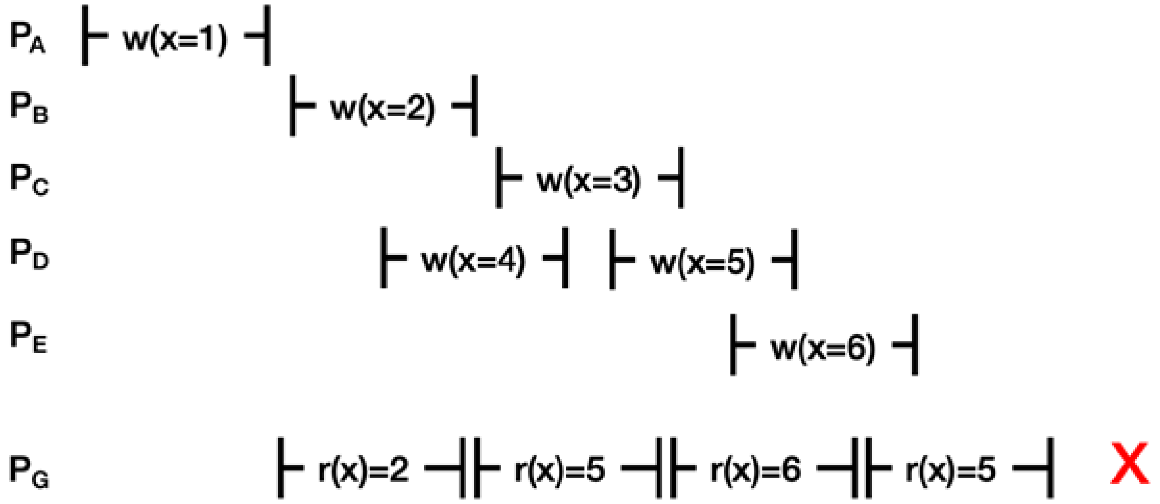
OR

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

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)

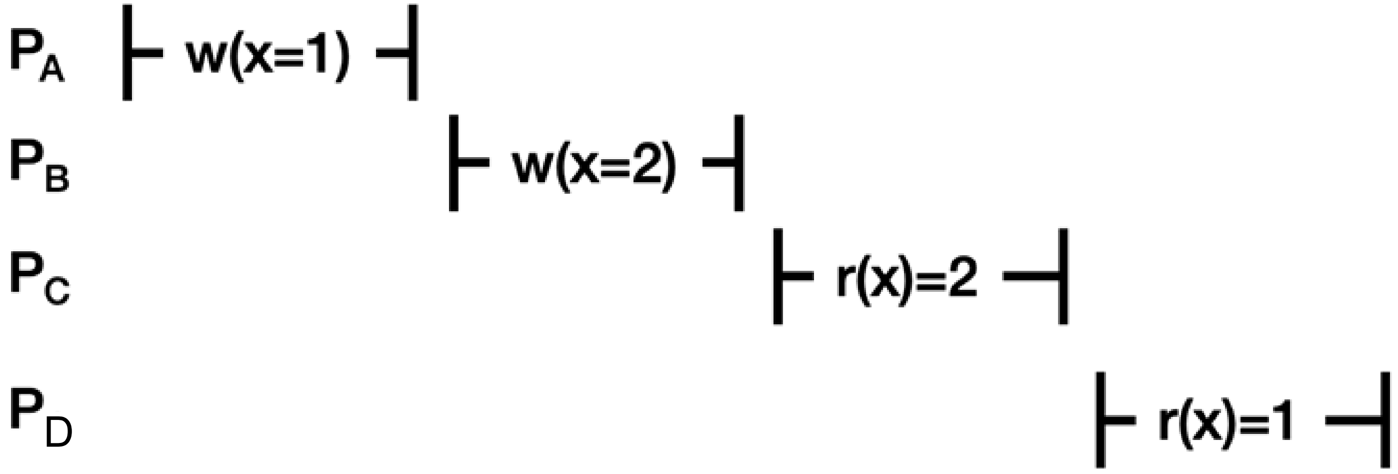
Node B: write another post (Event 3)

Causality: (Event 1 -> Event 2)

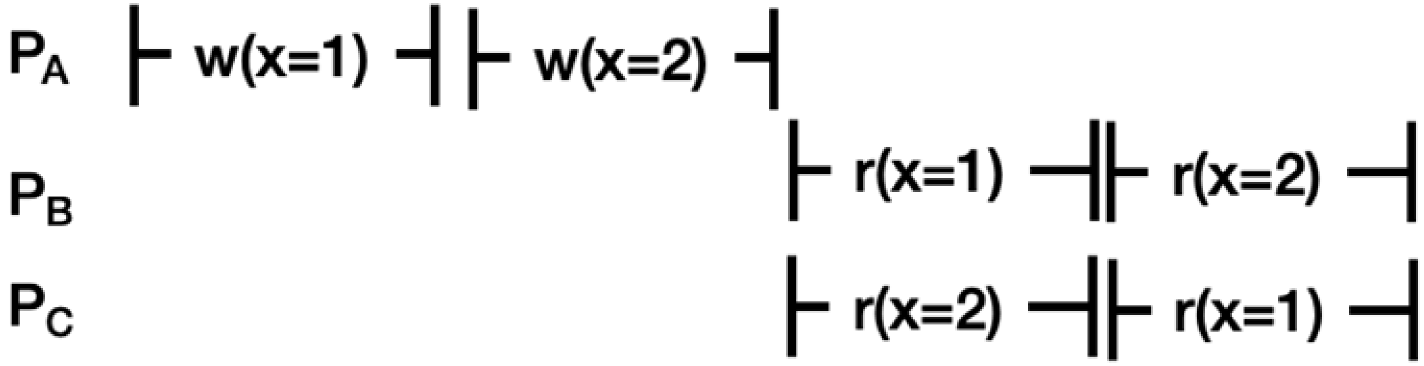
Other nodes may see different order of events, which can be

- 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

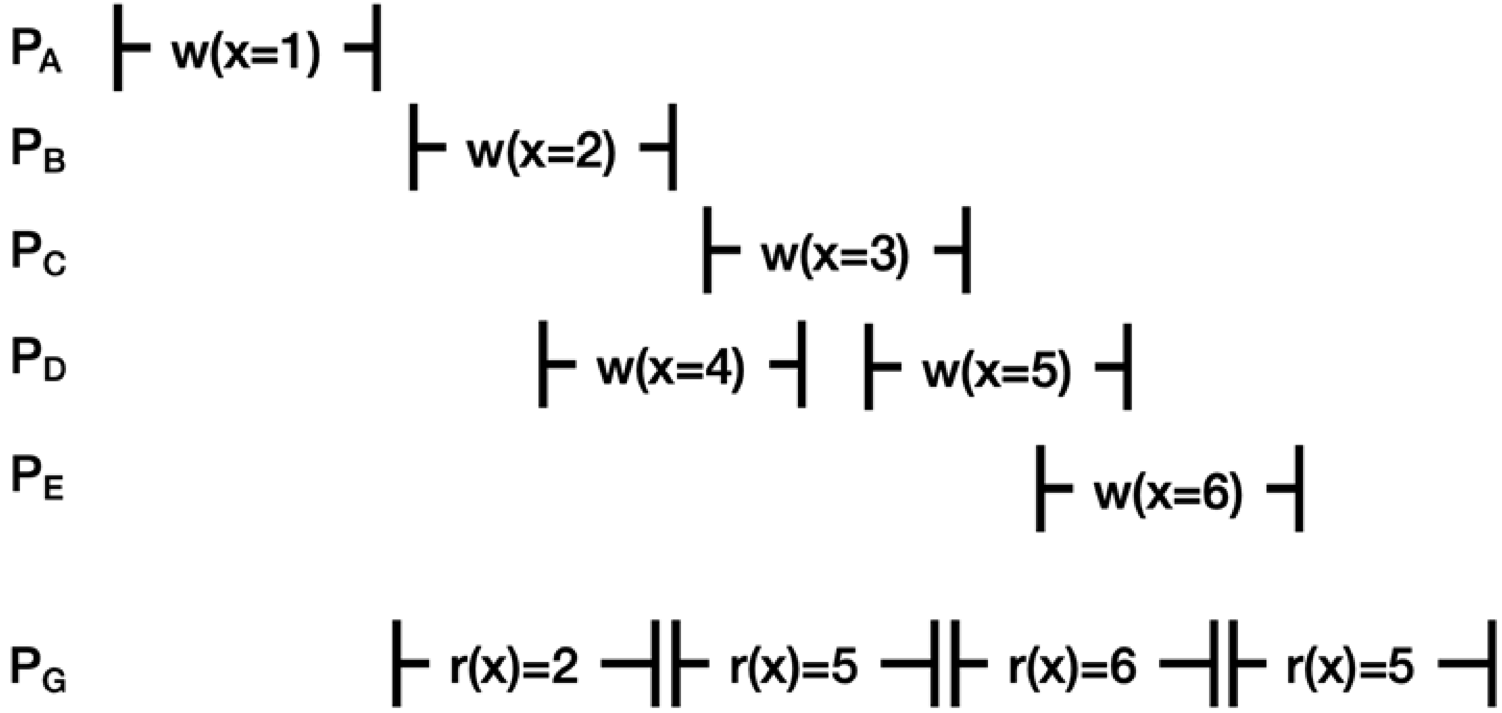
Example:



Example:



Example:



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)