Storage with Strong Consistency

ENGR689 (Sprint)

TEXAS A&M UNIVERSITY

Weak vs Strong Consistency

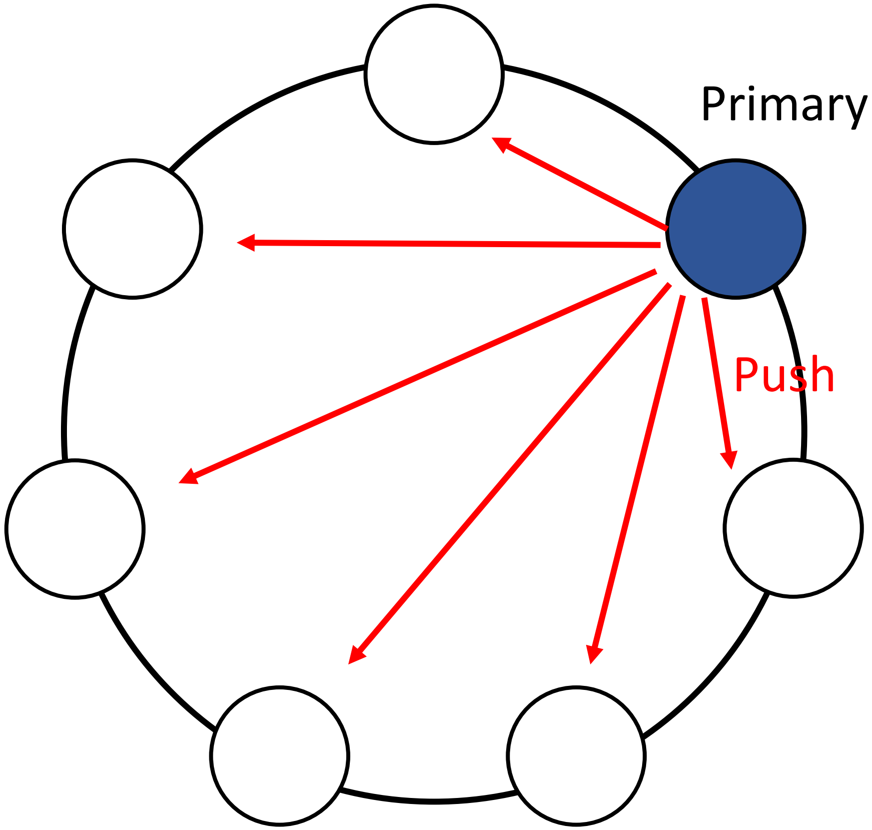
- In distributed systems, consistency is often the target of weakening
- Examples of weak consistency:
  - NoSQL servers – Dynamo
  - Datanodes in HDFS
- But sometimes, **strong consistency** is needed

Use Cases of Strong Consistency

- Configuration management
- Message Queues
- Group membership
- Synchronization:
  - Mutexes and read/write locks
  - Barriers: process joins

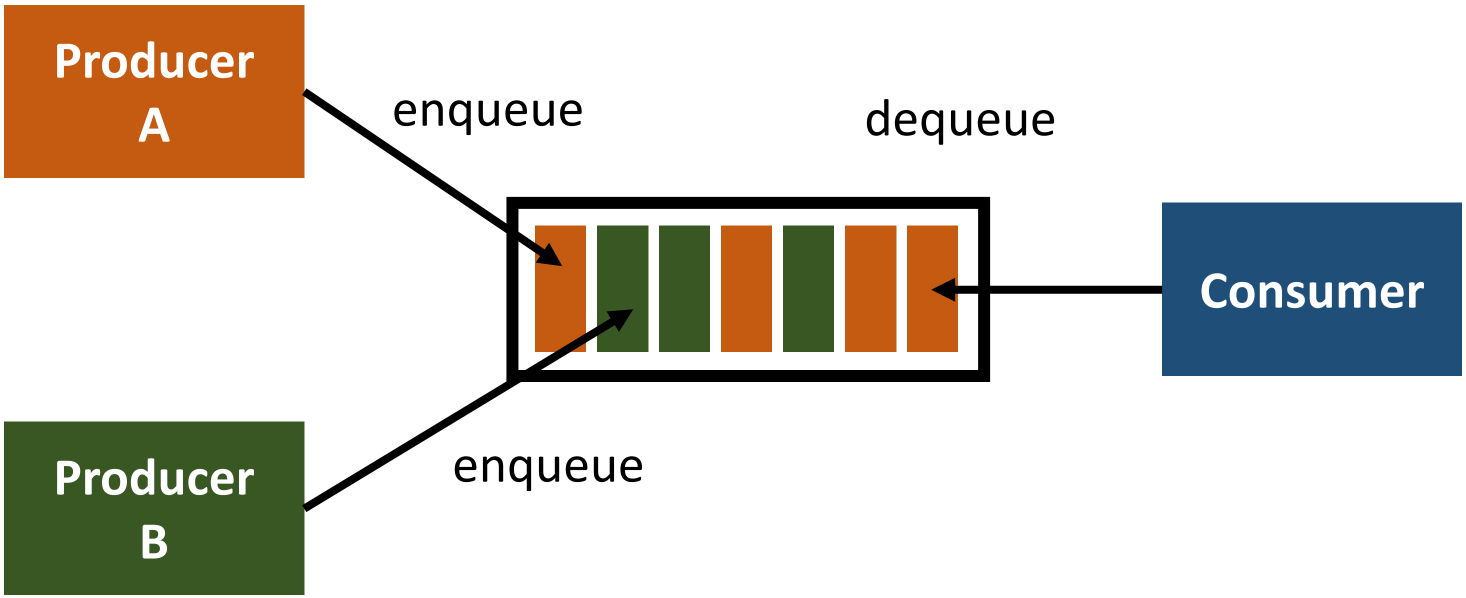
(Will talk about them one by one)

Use Case: Configuration & Group Membership



[Configuration]
Primary: …
Secondaries: …
Port IDs: …
Created by: …

Use Case: Message Queues



Use Case: Synchronization

- Mutexes

```
lock();

last_x = read(x);
write(y, last_x);
write(x, last_x + 1);

unlock();
```

- Read/write locks

```
rd_lock();

if (queue.size > 0) {
  wr_lock();
  x = queue.dequeue();
  wr_unlock();
}

rd_unlock();
```

Shared among readers

Exclusive for one writer

How to Define Strong Consistency?

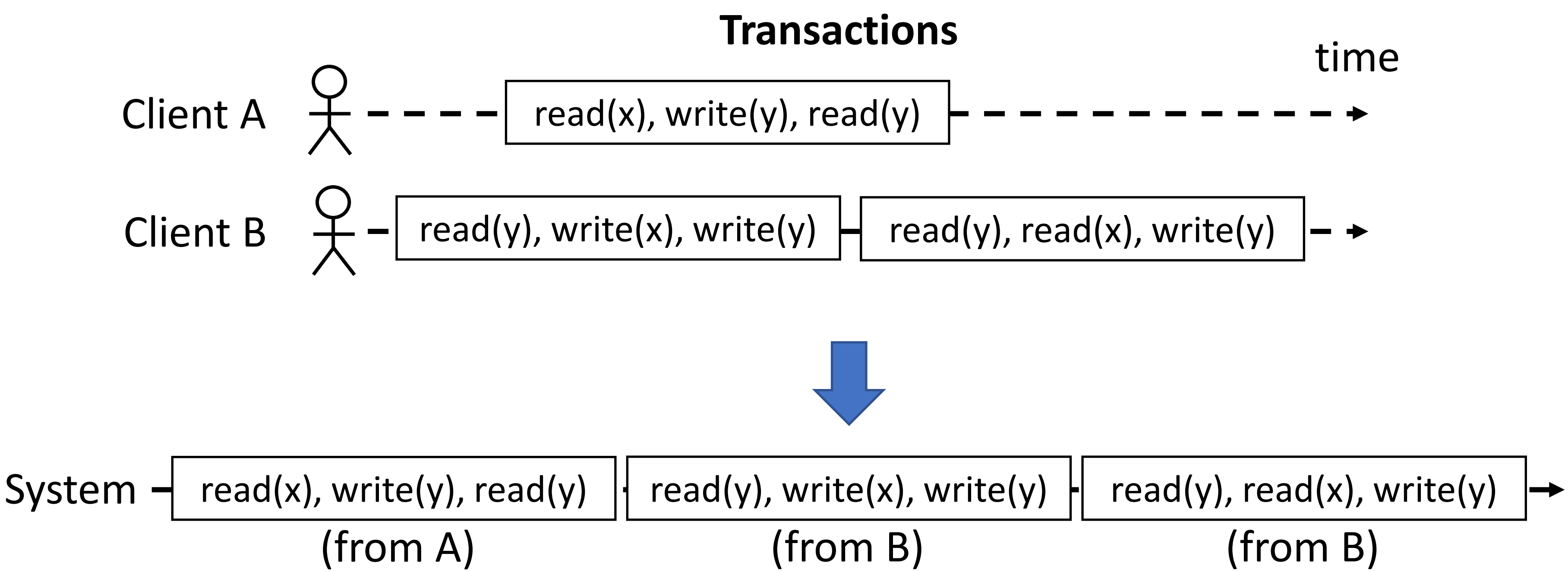
- **Linearizability**
  - Also called **atomic consistency** or **immediate consistency**
  - As soon as a write operation finishes, the whole system should see the latest data.

Serializability vs Linearizability

- For distributed systems, these two words are used in very specific contexts
- Serializability: for databases
  - Equivalent to Serializable **Isolation (I)** in ACID
  - No ordering for concurrent transactions
- Linearizability: for strongly-consistent reads/writes
  - In respect to operations, not transactions
  - Considering the global ordering of reads/writes

Serializability vs Linearizability

- <u>**Serializability**</u>

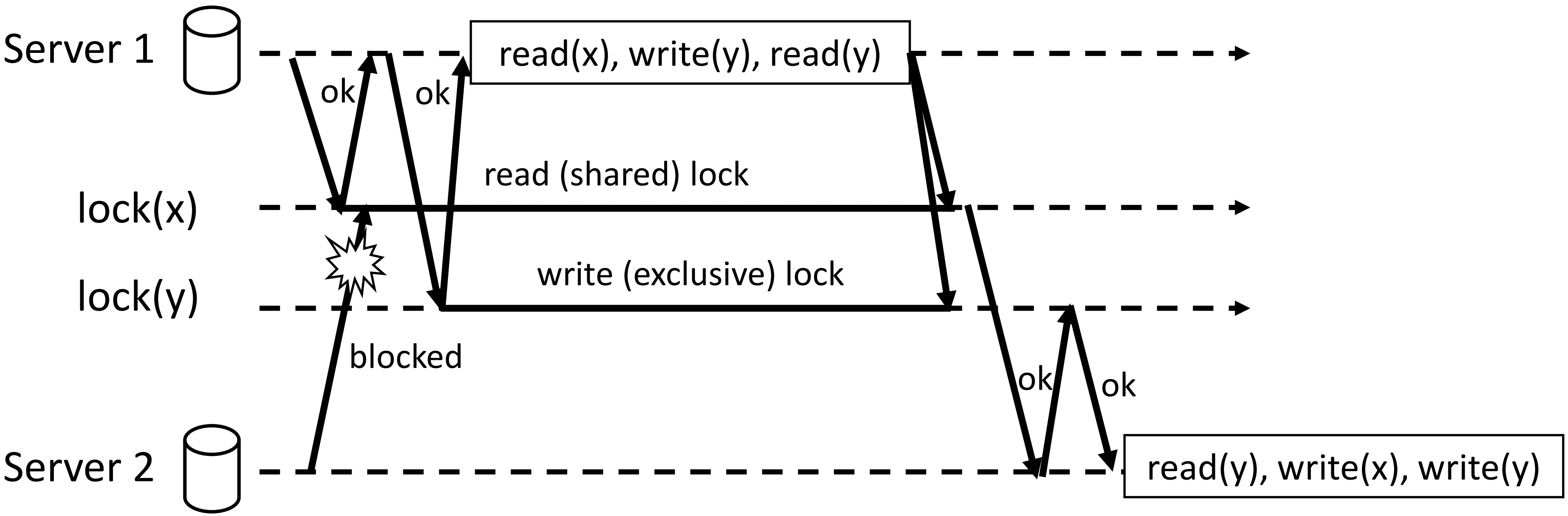


No constraint of ordering between concurrent transactions.

Serializability vs Linearizability

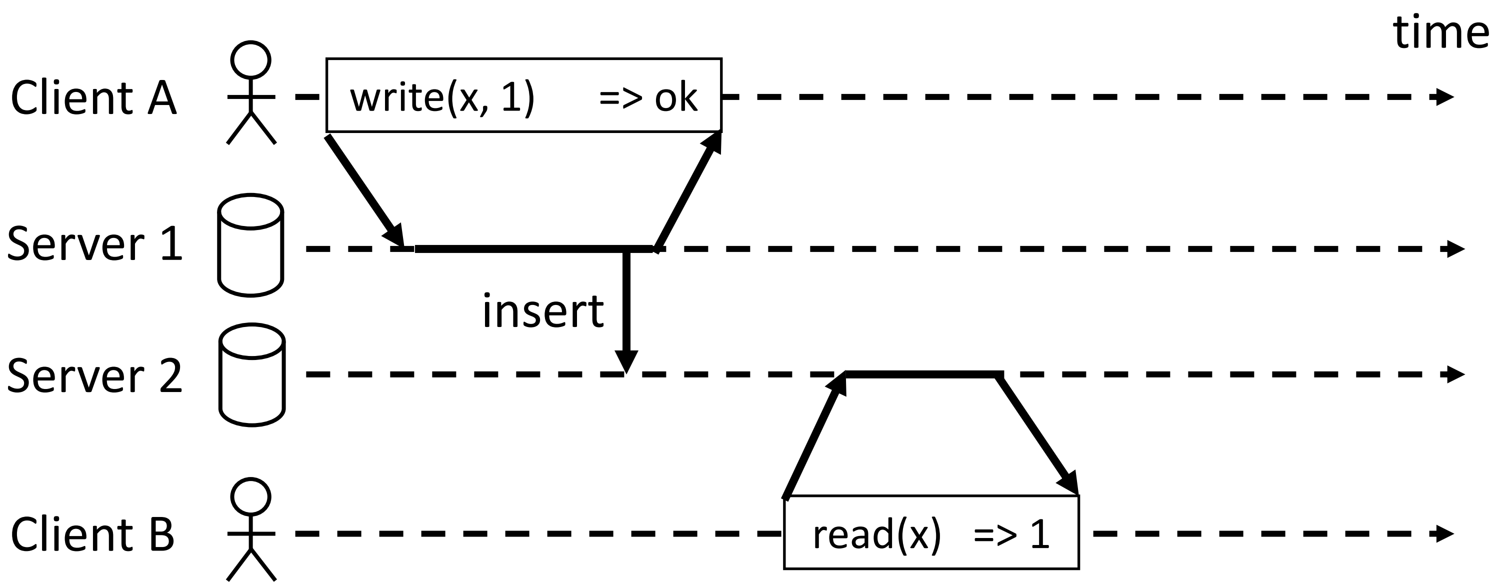
- Actually, serializability can be implemented by linearizability (i.e., a lock service)

Two-phase lock (2PL) ➔ Irrelevant to two-phase commit (2PC)



Serializability vs Linearizability

- **Linearizability**
  As soon as a write operation finishes, the whole system should see the latest data.

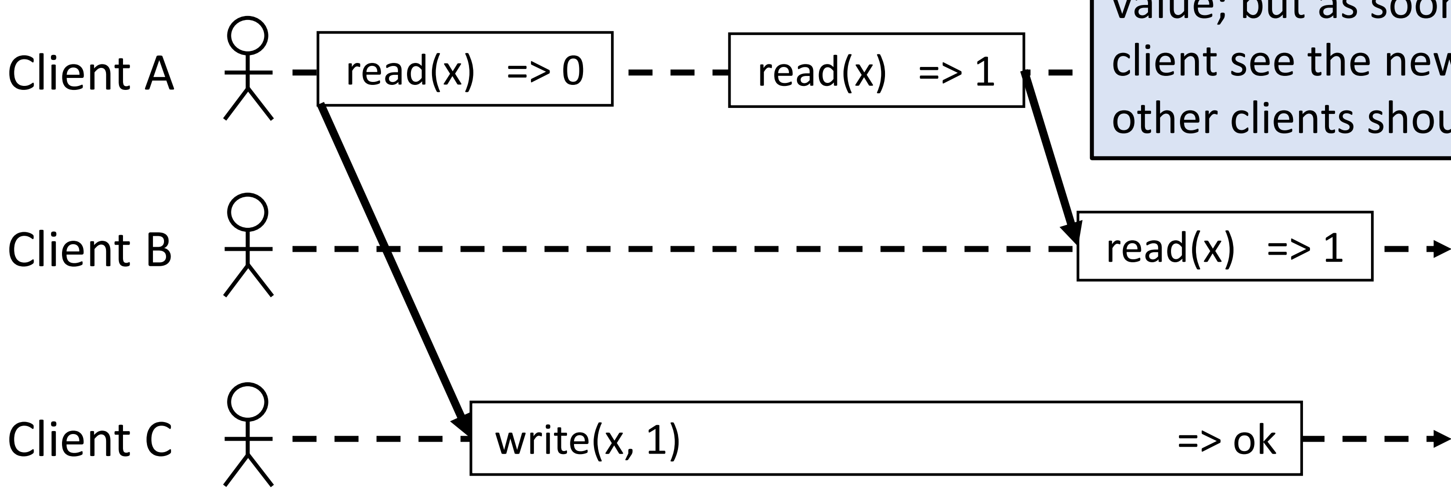


More on Linearizability

- Reads concurrent with a write

Any read that begins before the write must see the old value.

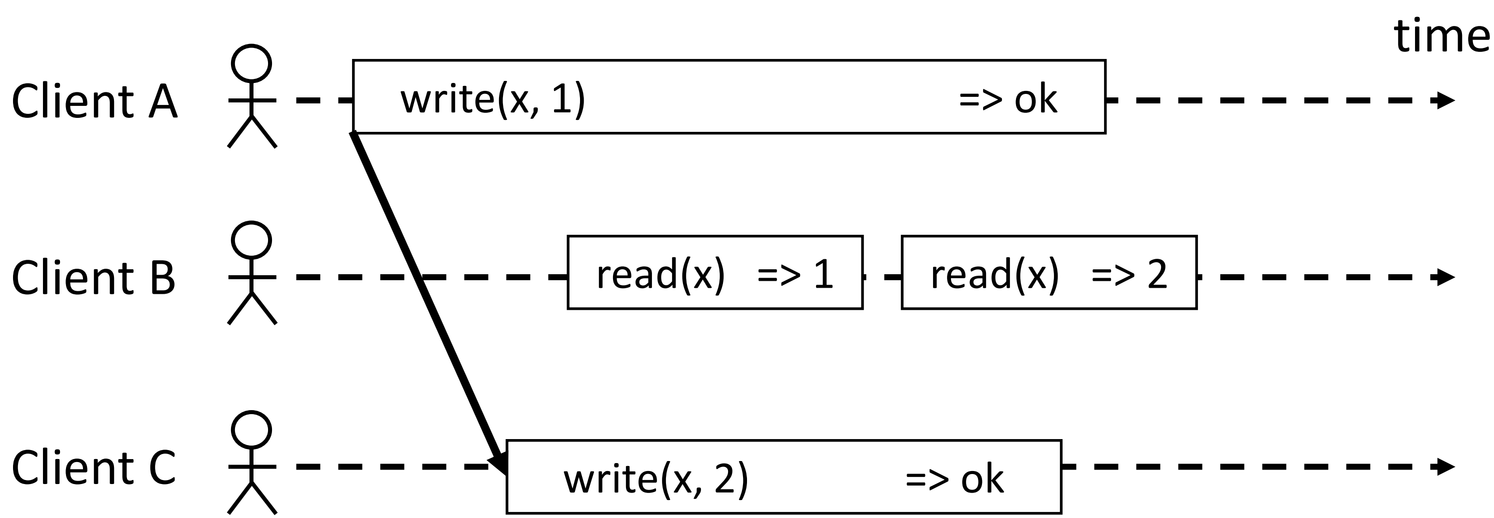
Any read that begins AFTER the write may see old or new value; but as soon as one client see the new value, all other clients should too.



More on Linearizability

- Write concurrent with another write

Any write begins before another write must be committed first.

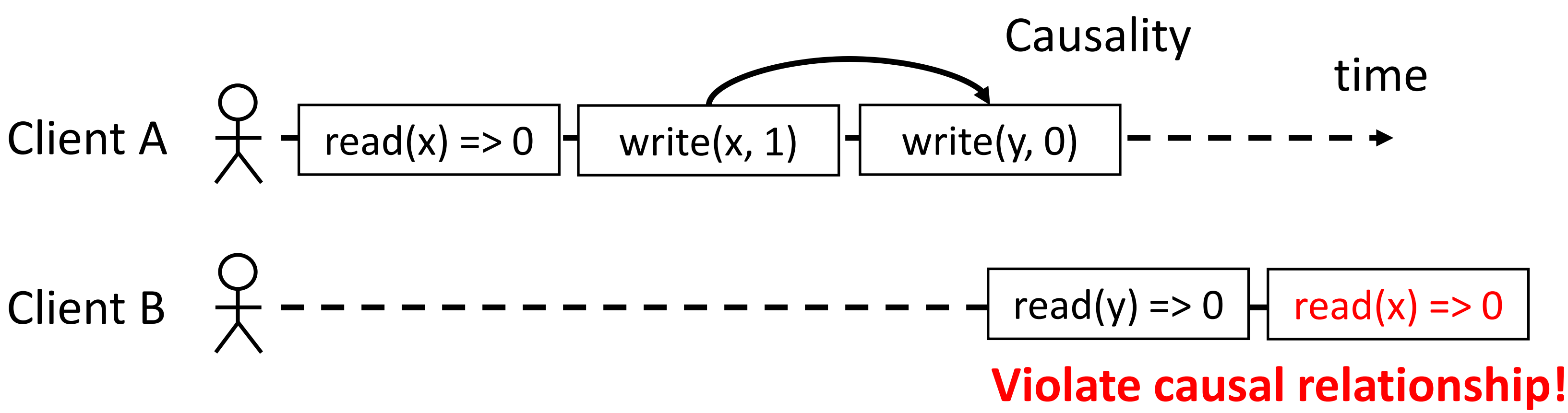


Linearizability vs Causality

- **Linearizability** implies **causal consistency**

  (1) Causality is based on **happens-before** relationship in a client

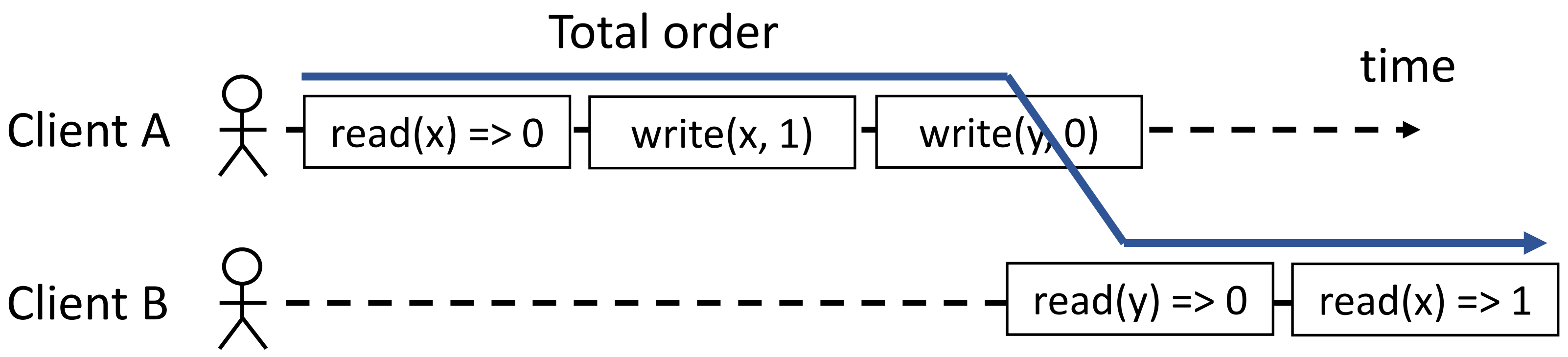
If client sets x = 1 and sets y = 0, then the two values have causal relationship.



Linearizability vs Causality

- **<u>Linearizability</u>** implies **<u>causal consistency</u>**

(2) Linearizability implies **<u>total ordering</u>** of each object, so automatically preserves happens-before.



A linearizable system doesn’t have to do anything to preserve causal consistency.

How to Implement Linearizability?

- Read/write from single leader
  - Failover to a replica may lose linearizability
- Consensus algorithms
  - Using two-phase commits (2PC) or Paxos
  - Prevent stale replicas
- Most likely unlinearizable: multi-leader replication

Q: How does linearizability apply to CAP Theorem?

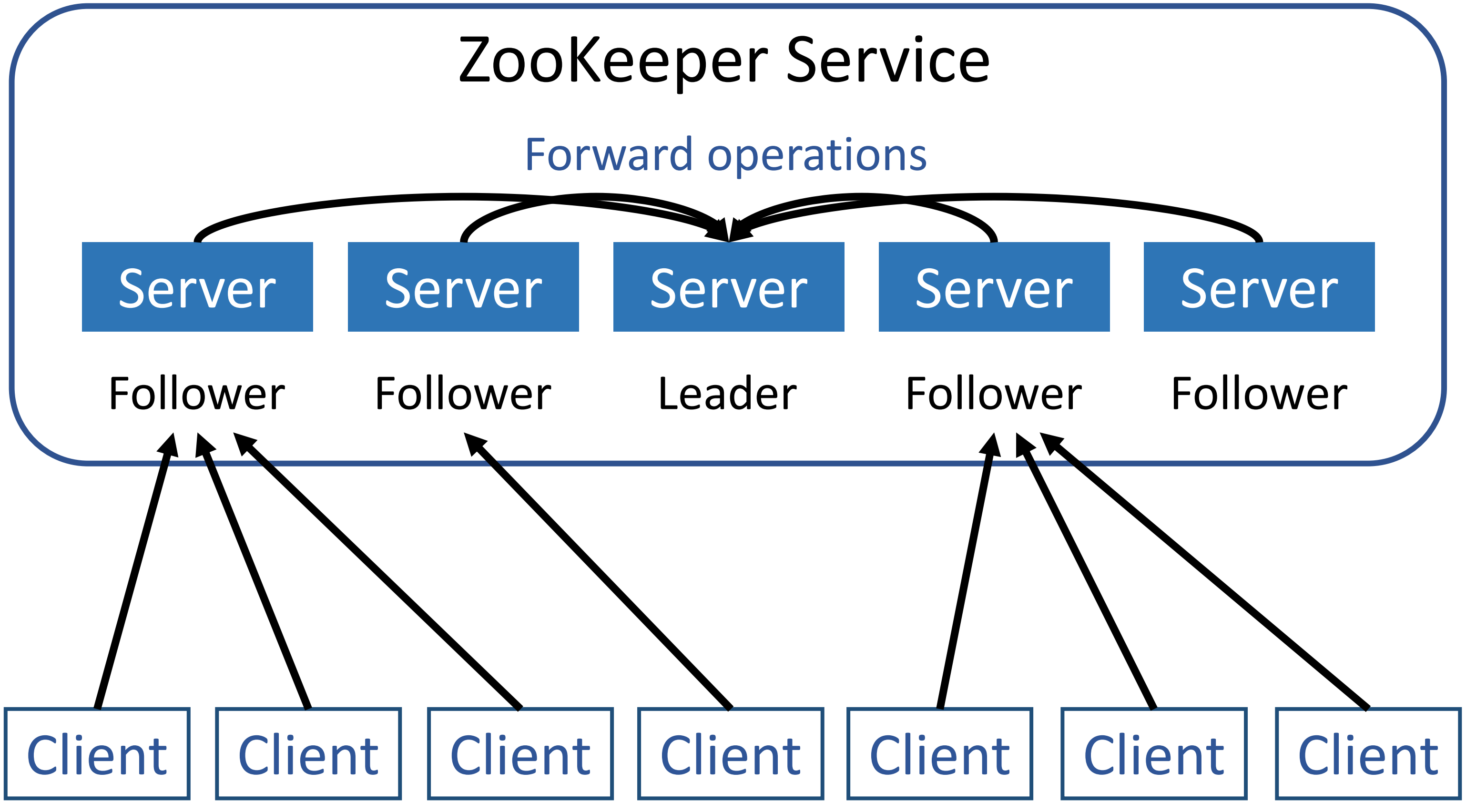
Linearizability in CAP Theorem

- Linearizability = Strong C
  - Read/write through single leader: lose A & P
  - Read through replicas, write through leader: lose P
- With **<u>consensus</u>**, linearizability can be:
  - Partition tolerant when ½ of replicas are connected
  - Fair availability with wait-free operations and fast, lossless leader recovery

Apache ZooKeeper

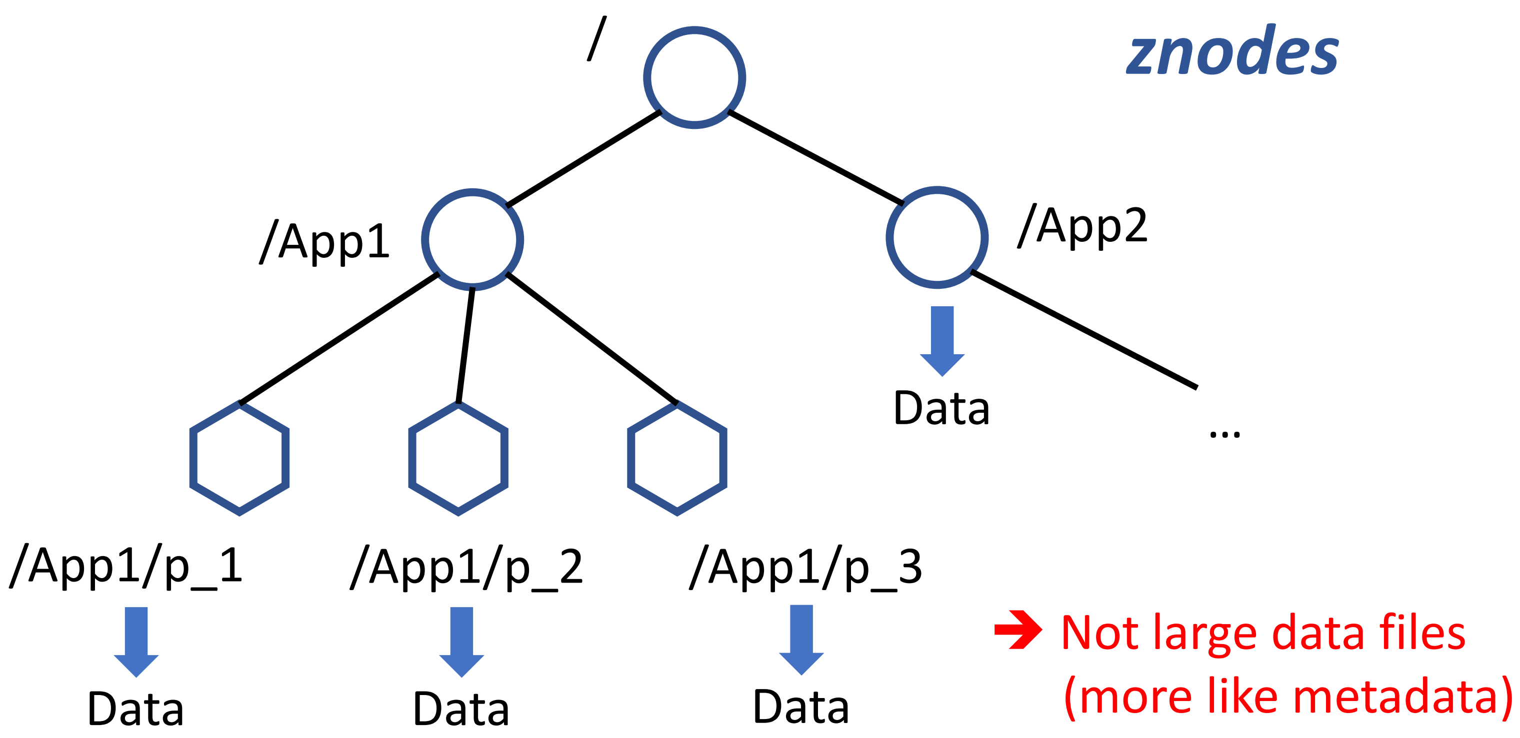
- A **coordination service** for all use cases of strong consistency in a distributed system
- **Wait-free**: any operation will not block on other slow or failed clients
- ZooKeeper has no API for locking, but can be used to implement any locking mechanism

System Overview



Namespace

- ZooKeeper uses a filesystem-like namespace



Namespace

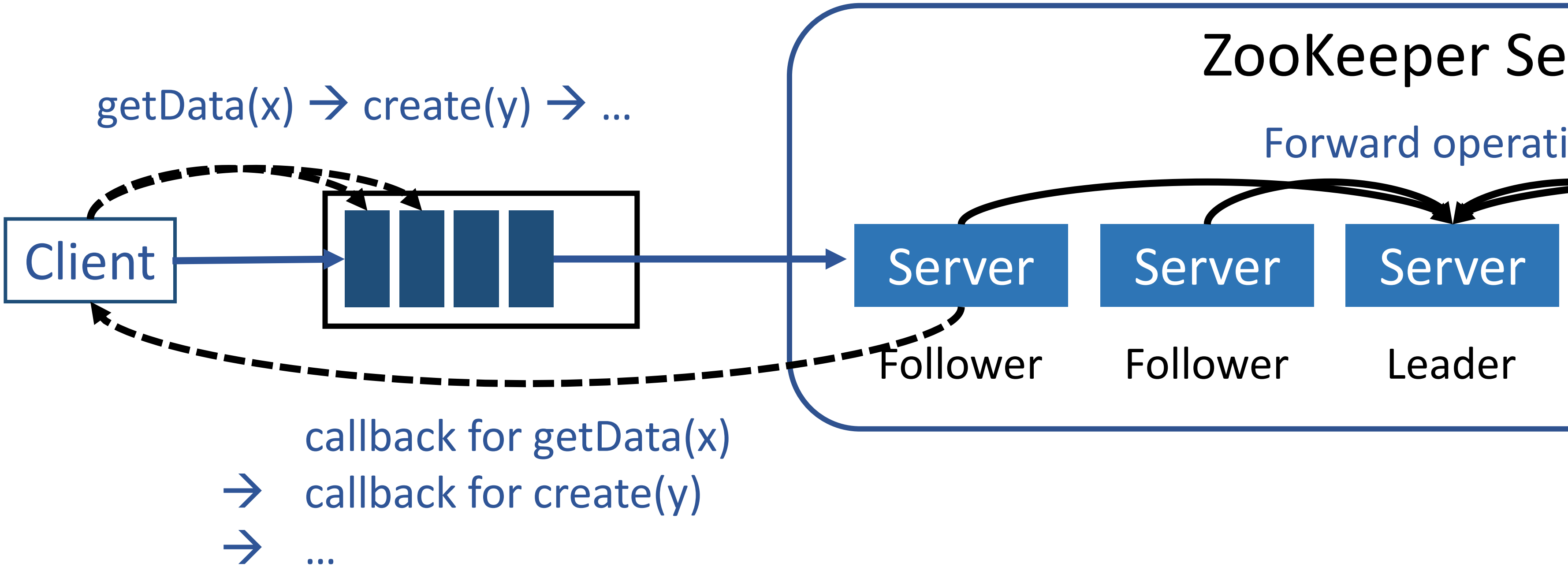
- ZooKeeper has two types of znodes (paths)
  - **Permanent (regular)**: clients explicitly create and delete the znodes
  - **Ephemeral**: clients create the znodes, and either delete them explicitly or let the system automatically deletes them when client sessions timeout.

API

- `create(path, data, flags)`
  - flags: regular/ephemeral, sequential (appending a seqnum)
- `getData(path, watch) -> (data, version)`
- `setData(path, data, version)`
- `delete(path, version)`
- `exists(path, watch) -> true/false`
- `getChildren(path, watch) -> [paths]`
- `sync(path)` path is ignored now

Asynchronous Operations

- Operations can be synchronous or asynchronous
- Client can queue up multiple asynchronous operations
- Server responds by invoking callbacks



Event Notification

- All operations except **sync** are wait-free
- No locking API but clients can implement locks using **watch events**

Server will push events to client when the path is updated

Lock (very naïve version)

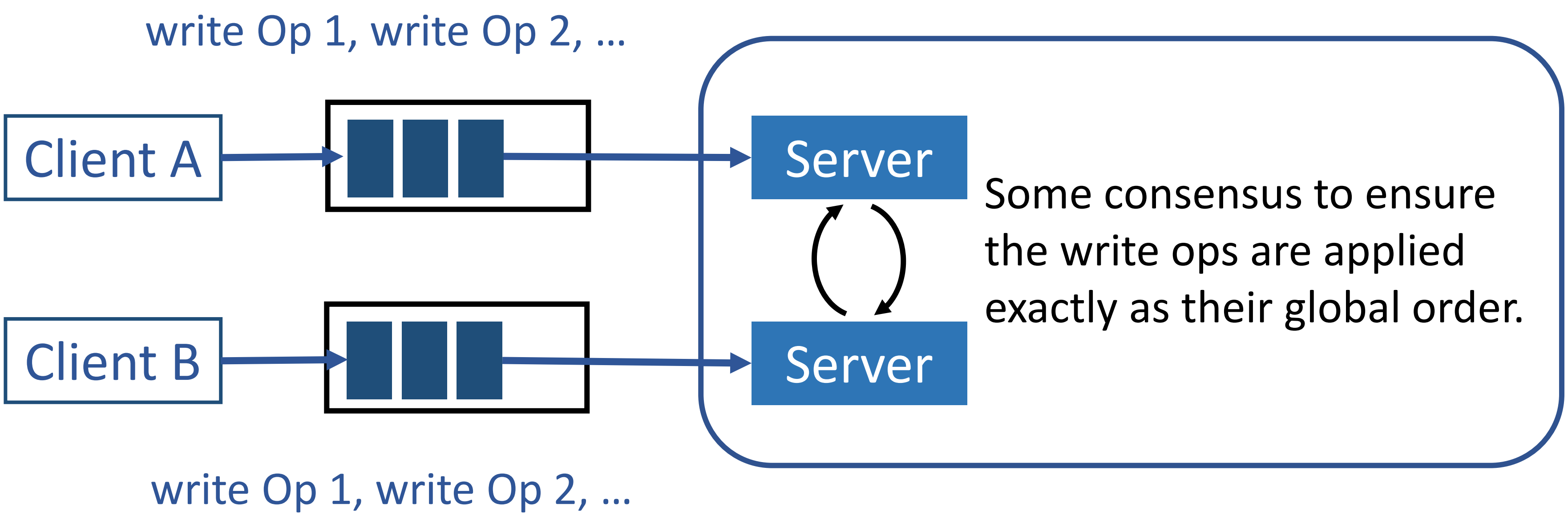
```
1   l = “/my-lock”;
2   if exists(l, watch=true) then wait for watch event;
3   n = create(l, EPHEMERAL);
4   if n is error then goto 2;
```

Unlock

```
1   delete(l);
```

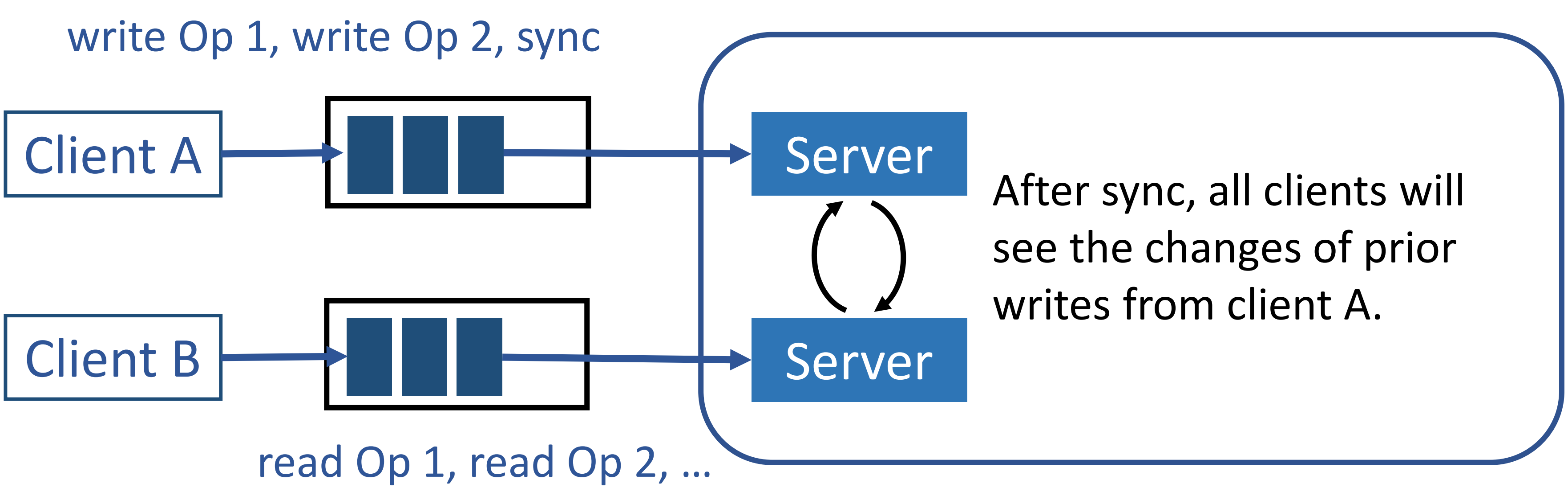
A(asynchronous)-Linearizability

- Local order: all operations from the same client are processed FIFO (first-in-first-out)
- Global order: Linearizable writes



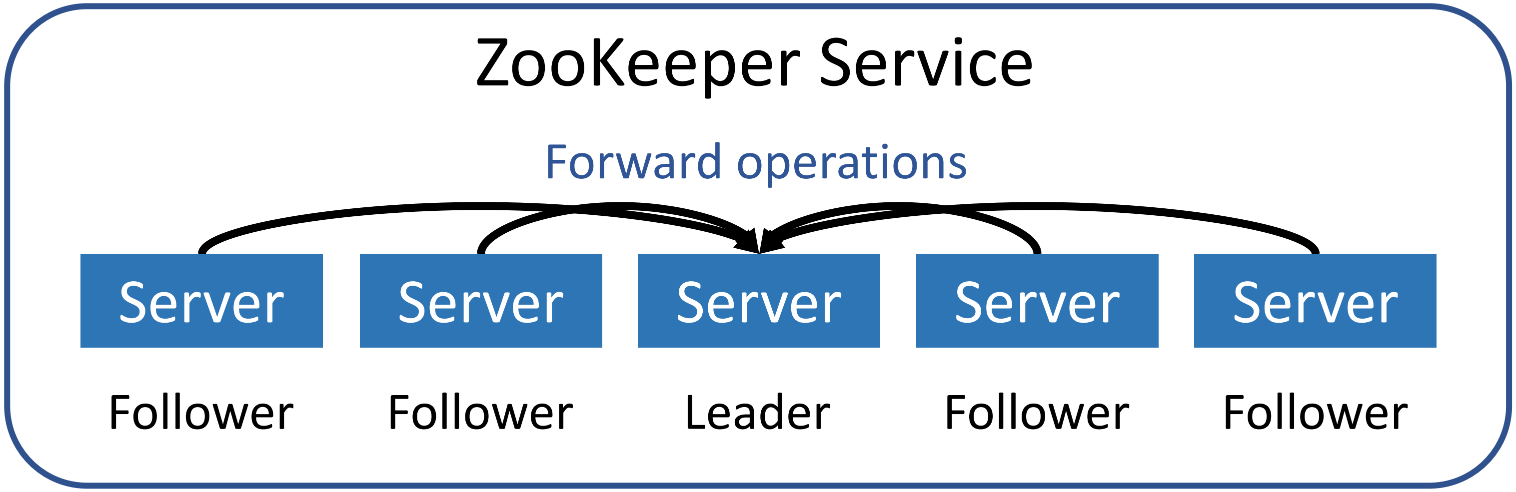
A(asynchronous)-Linearizability

- Reads are not linearized (may not see latest state)
  - Read directly from server (identical replicas)
  - Other servers may have pending writes in queues
  - Solution: **<u>sync after writes</u>**



Zab (Atomic Broadcast)

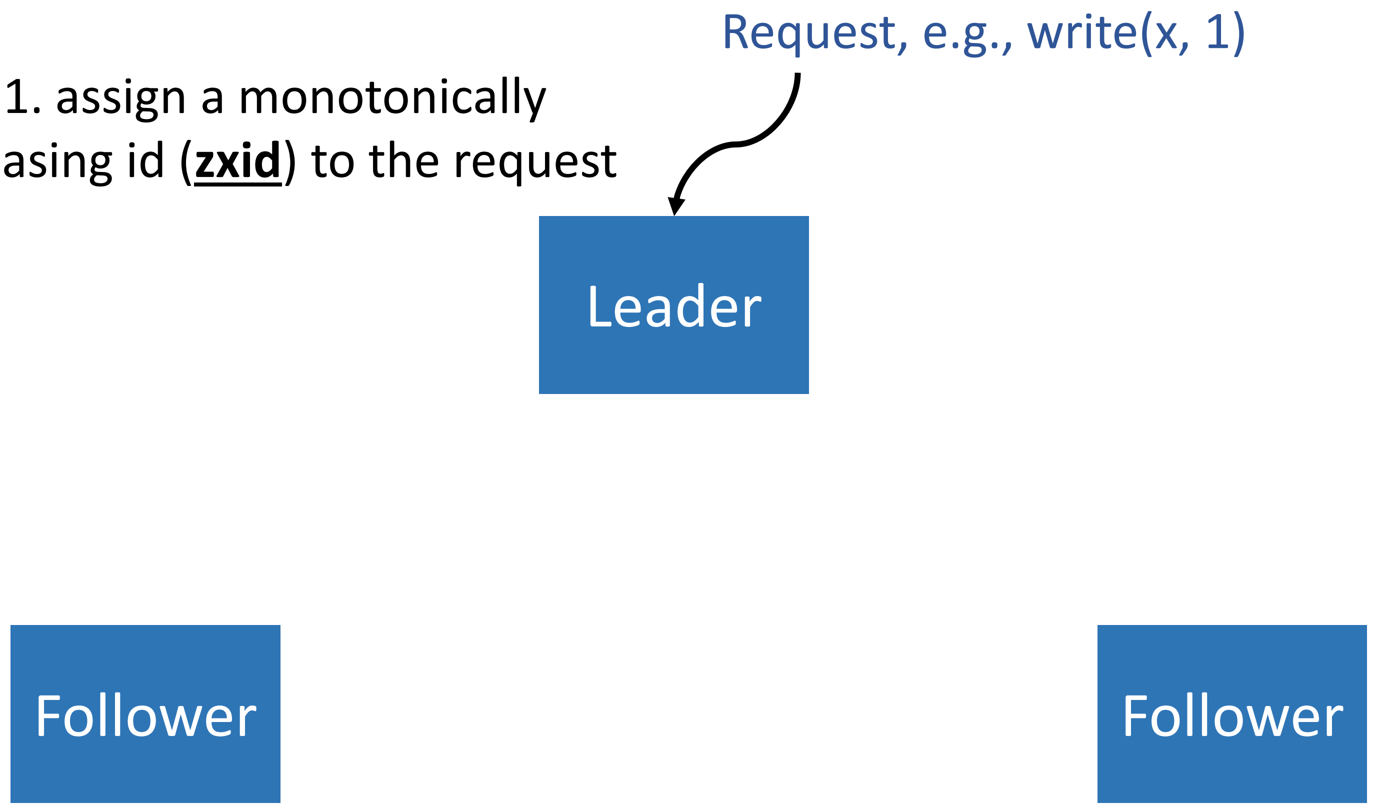
- All servers forward messages to a single leader
  - Important role as a **sequencer**
  - Leader can change if partitioned or failed
- The leader broadcasts (proposes) the messages to be delivered to all followers



Zab (Atomic Broadcast)

- **<u>Two-phase commit</u>** (2PC)

Step 1. assign a monotonically increasing id (**<u>zxid</u>**) to the request

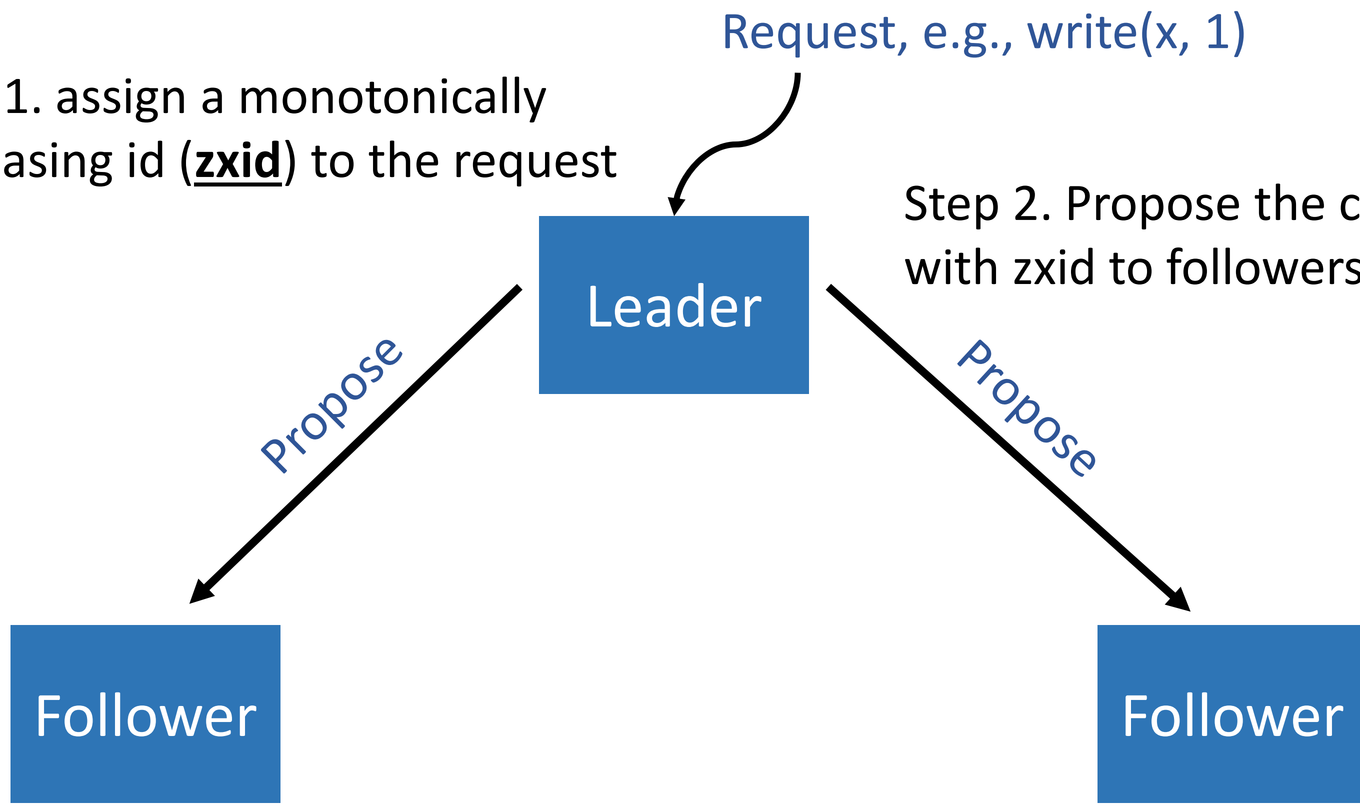


Zab (Atomic Broadcast)

- **<u>Two-phase commit</u>** (2PC)

Step 1. assign a monotonically increasing id (**<u>zxid</u>**) to the request

Step 2. Propose the change with zxid to followers



Zab (Atomic Broadcast)

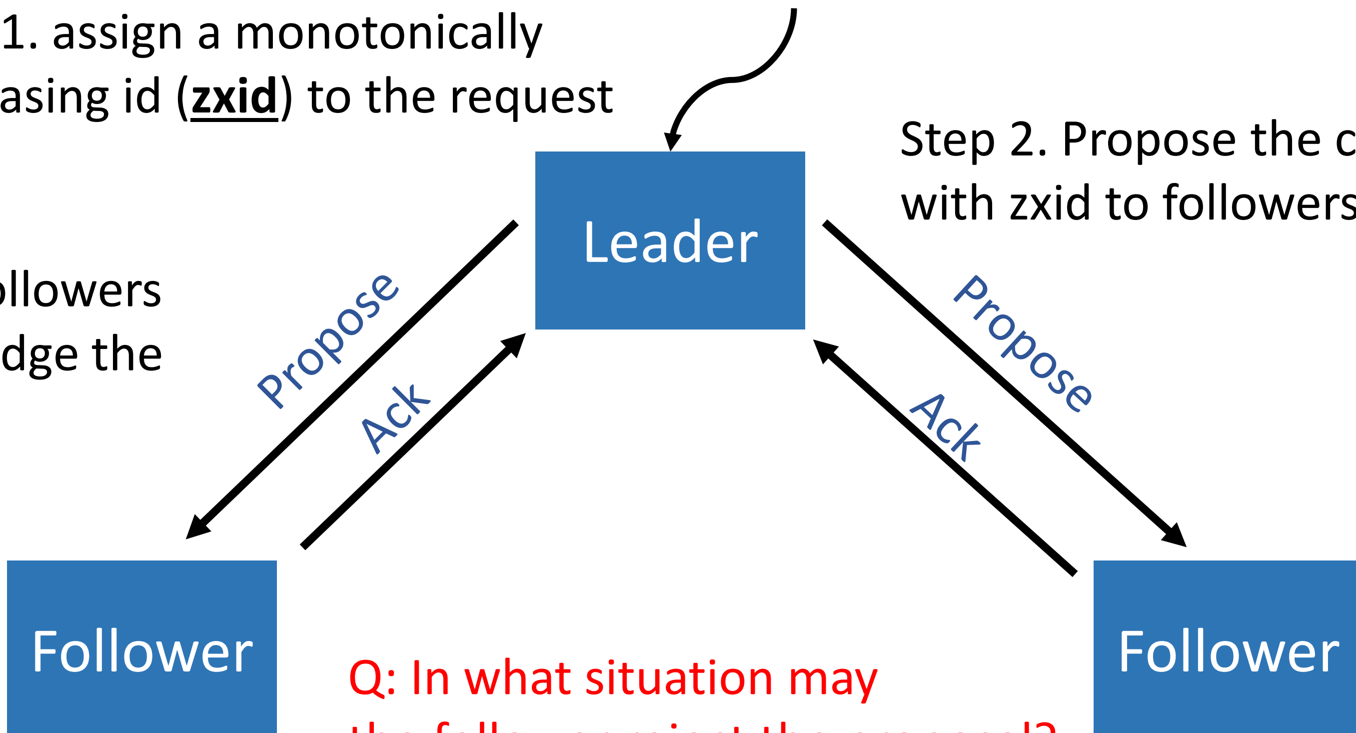
- **<u>Two-phase commit</u>** (2PC)

Request, e.g., write(x, 1)

Step 1. assign a monotonically increasing id (**<u>zxid</u>**) to the request

Step 2. Propose the change with zxid to followers

Step 3. Followers acknowledge the proposal.



Q: In what situation may the follower reject the proposal?

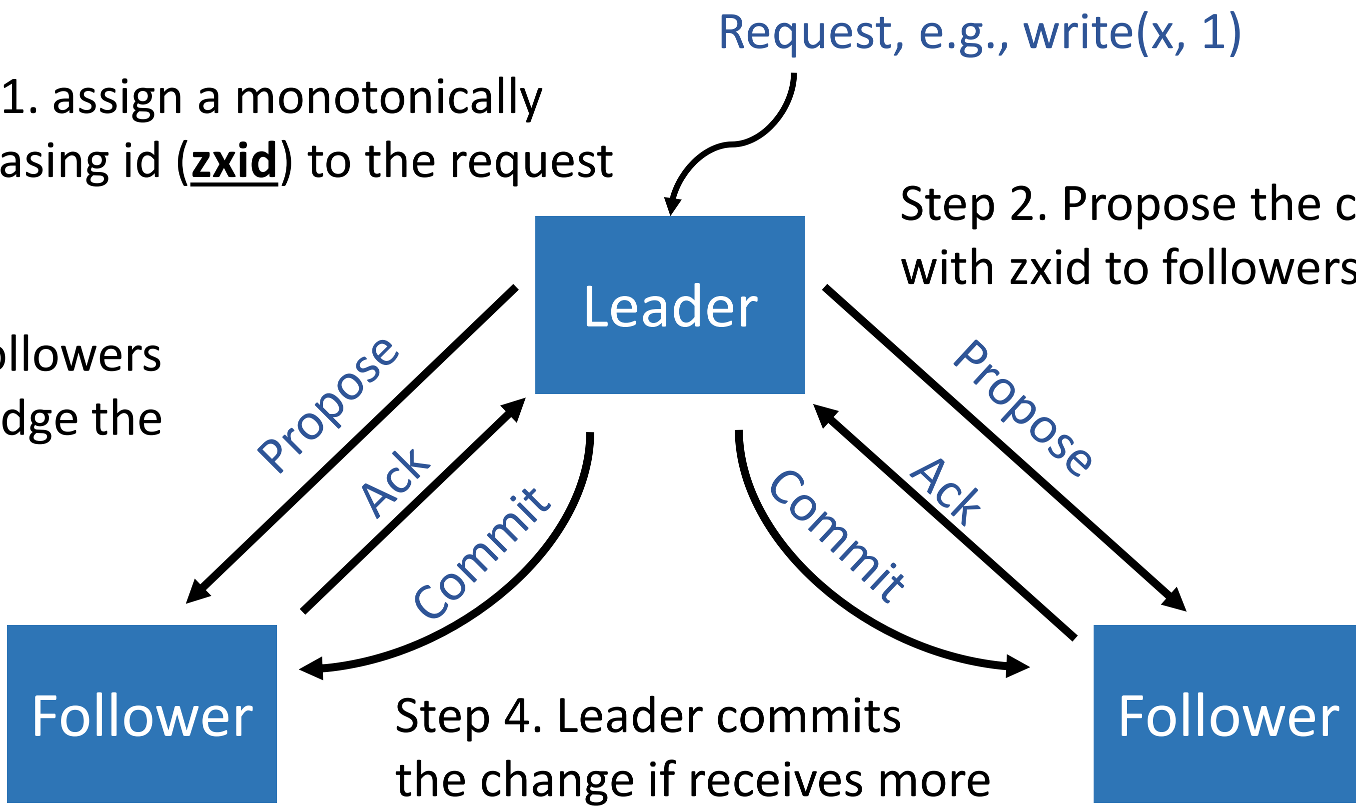
Zab (Atomic Broadcast)

- **<u>Two-phase commit</u>** (2PC)

Step 1. assign a monotonically increasing id (**<u>zxid</u>**) to the request

Step 2. Propose the change with zxid to followers

Step 3. Followers acknowledge the proposal.

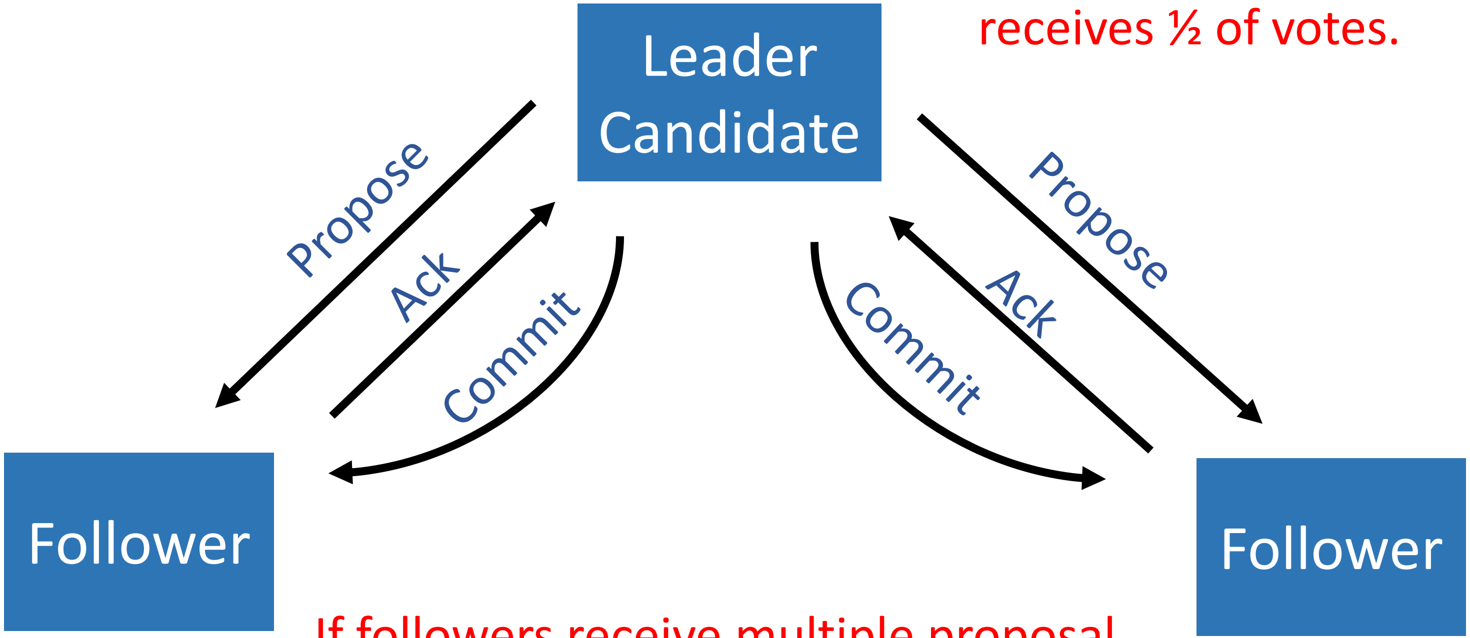


Step 4. Leader commits the change if receives more than ½ of acks (votes).

Leader Election

- If a server finds the leader disconnected or failed, it tries to become the leader (**<u>same 2PC protocol</u>**).

Becomes the leader when receives ½ of votes.



If followers receive multiple proposal, they vote for the candidate with the highest zxid.

References

- "ZooKeeper: Wait-free coordination for Internet-scale systems," USENIX ATC '10 (by Hunt et al.)
- Zab protocol: "A simple totally ordered broadcast protocol", LADIS '08 (by Reed and Junqueira)
- "Linearizability: A Correctness Condition for Concurrent Objects", TOPLAS 1990 (by Herlihy and Wing)
- "Designing Data-Intensive Applications", O'Reilly 2017 (by Martin Kleppmann)