CS 60002: Distributed Systems

T11: Replication

Department of Computer Science
and Engineering

INDIAN INSTITUTE OF TECHNOLOGY
KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

Content courtesy: Lecture slides by Prof. Arobinda Gupta

Broad Idea of Replication

- Multiple copies of data kept in different nodes
  - A set of replicas holding copies of a data
    - Nodes can be physically very close or distributed all over the world
  - A set of clients that make requests (read/write) of the data in a replica

Broad Idea of Replication

- Multiple copies of data kept in different nodes
  - A set of replicas holding copies of a data
    - Nodes can be physically very close or distributed all over the world
  - A set of clients that make requests (read/write) of the data in a replica

- Why replicate?
  - **Fault Tolerance**
    - Service can be provided from a different replica if one replica fails
  - **Load Balancing**
    - Load can be shared by multiple replicas (ex. web servers)
  - **Reduced latency**
    - Replicas placed closer to request source for faster access, example: geo-replicated data sources

Broad Idea of Replication

- Ideally, the user should think that there is a single copy of the data (**replication transparency**)
  - Requires that a write at one replica to propagate instantaneously to another replica as if it is done on a single copy
    - Impossible if real time ordering of read/write operations is to be maintained as in a single copy

Broad Idea of Replication

- Ideally, the user should think that there is a single copy of the data (**replication transparency**)
  - Requires that a write at one replica to propagate instantaneously to another replica as if it is done on a single copy
    - Impossible if real time ordering of read/write operations is to be maintained as in a single copy

- Then how should we keep replicas updated in the presence of writes?
  - Should all copies of the data have the same value always, or are intermediate differences allowed?
  - Depends on the **consistency model** to be satisfied depending on the application need
    - What is a consistency model?

Design Issues

- Consistency model to be enforced
- Where can updates happen?
  - One designated replica or any replica?
- When to propagate updates?
  - Eager (immediately, before response to client) or lazy (sometime after response is sent to client)?
  - Depends on consistency model to be supported
- How many replicas to install?
- When to Install a replica
  - Static or On-demand?
- Where to place the replicas?

Consistency Models

Consistency Models

- Defines what guarantees are provided on reads on a shared data in the presence of possibly interleaved/overlapped access
- Replicas of a data accessed at multiple sites can be viewed as a single shared data
- Tradeoff
    - Should be strong enough to be useful
    - Should be weak enough to be efficiently implementable

Consistency Models

- Examples of consistency models
  - Linearizability
  - Sequential consistency
  - Causal consistency
  - Eventual consistency
- Many other models exist…
- Why so many models?
  - Application requirements are different
  - Stronger models require more overheads to implement, so many weaker models have evolved if strong guarantees are not needed for an application
  - Even within a single application, different types of data may require different consistency models

Linearizability

- Satisfied if there exists some sequential ordering of the reads and writes in which
  1. Operations of individual processes are ordered in the same way as in the actual order
  2. For two operations by two different processes, if times in actual order are t1 and t2, and times in the sequential order are t1' and t2', then if t1 < t2, then t1' < t2'
  3. Each read gets the value of the latest write before it in the sequential ordering

- Time can be based on any global timestamping scheme
- Used mostly for formal verification etc.

Linearizability

P1: Write (x) -> a

P2: Read (x) <- a

Linearizable

Linearizability

P1: Write (x) -> a

P2: Read (x) <- a

Linearizable

P1: Write (x) -> a

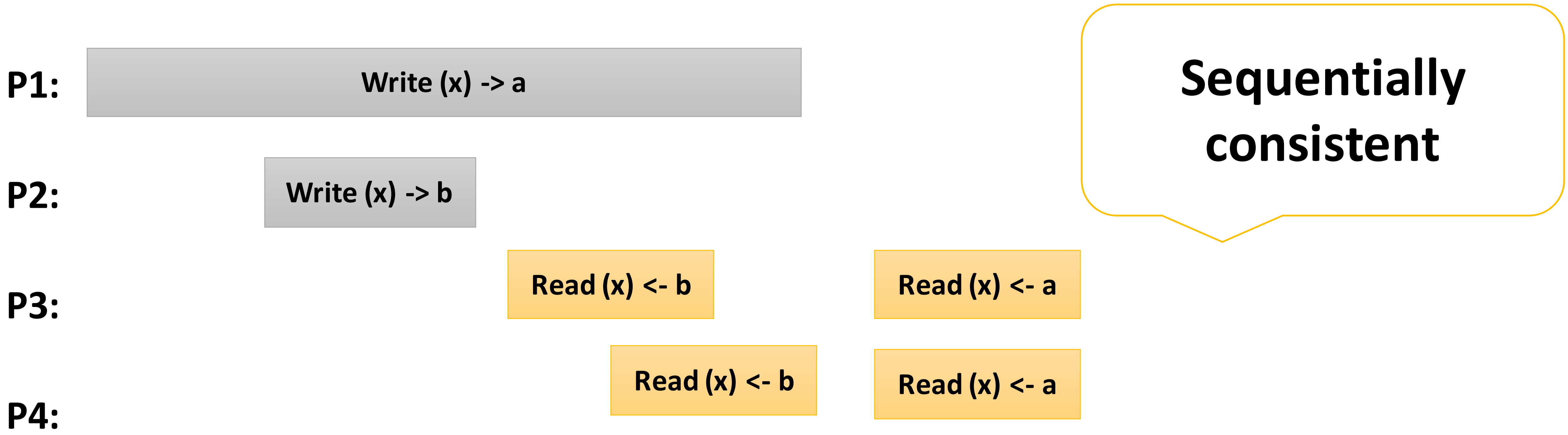
P2: Read (x) <- NULL   Read (x) <- a

Not linearizable

Sequential Consistency

- Requires only the first and third conditions of Linearizability
  - No ordering of events at different processes required
  - Linearizability implies sequential consistency but not vice-versa
- No notion of time, and hence no notion of "most recent" write
- Ordering of events at different processes may not be important as they could have happened in some other order in practice anyway due to different reasons (server speed, message delays,…)
- Widely used in practice
- Still costly to implement

Sequential Consistency



Sequential Consistency

P1: Write (x) -> a

P2: Write (x) -> b

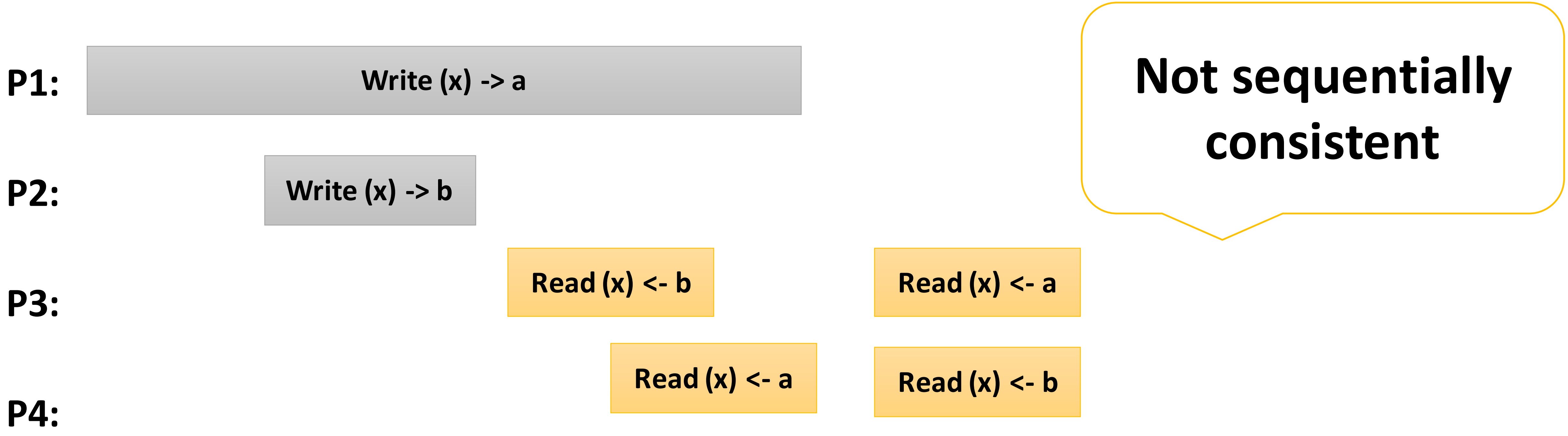
P3: Read (x) <- b   Read (x) <- a

P4: Read (x) <- b   Read (x) <- a

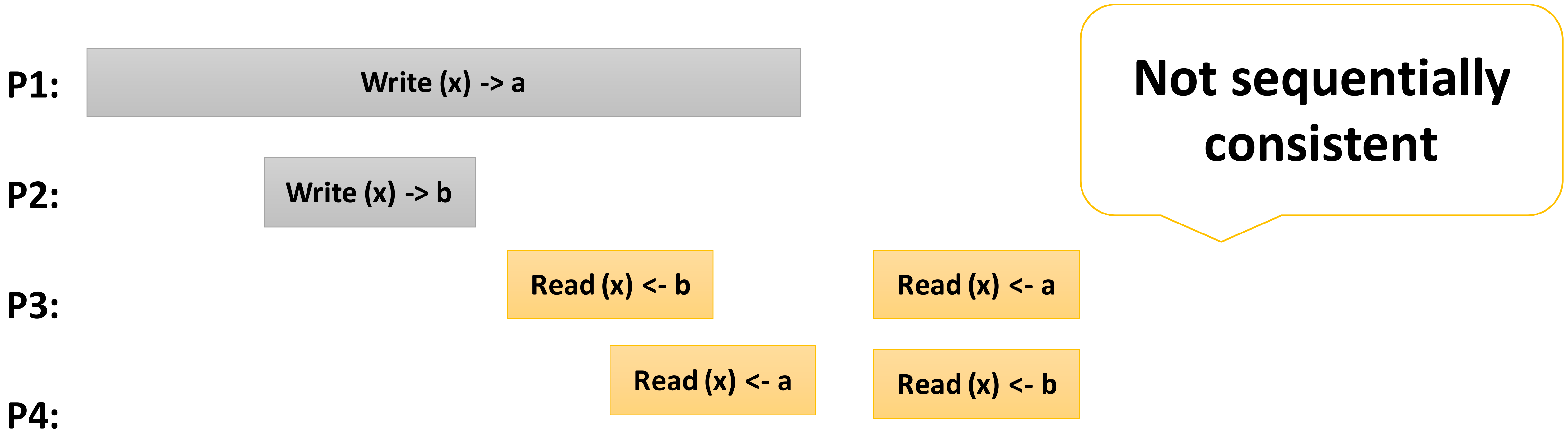
Sequentially consistent

Are the events linearizable?

Sequential Consistency



Sequential Consistency



Sequential consistency means all processes see all writes in the same order ("seeing" means the results returned by reads)

Causal Consistency

- **Causally related writes**
  - Writes in the same process are ordered in the same way as in the actual order
  - Writes in different processes linked by reads in between
- Writes not causally related are **concurrent**
- All writes that are causally related must be seen (results of read) by every process in the same order
- Writes that are not causally related can be seen in any order by different processes
- Value returned by the reads must be consistent with this causal order

Causal Consistency

P1:	Write (x) -> a			Write (x) -> c		
P2:		Read (x) <- a	Write (x) -> b			
P3:		Read (x) <- a			Read (x) <- c	Read (x) <- b
P4:		Read (x) <- a			Read (x) <- b	Read (x) <- c

Causal Consistency

P1:	Write (x) -> a			Write (x) -> c	
P2:		Read (x) <- a	Write (x) -> b		
P3:		Read (x) <- a		Read (x) <- c	Read (x) <- b
P4:		Read (x) <- a		Read (x) <- b	Read (x) <- c

Is this sequentially consistent?

Causal Consistency

P1:	Write (x) -> a			
P2:		Read (x) <- a	Write (x) -> b	
P3:			Read (x) <- b	Read (x) <- a
P4:			Read (x) <- a	Read (x) <- b

Is this causally consistent?

Causal Consistency

P1:	Write (x) -> a			
P2:		Write (x) -> b		
P3:			Read (x) <- b	Read (x) <- a
P4:			Read (x) <- a	Read (x) <- b

Is this causally consistent?

Eventual Consistency

- Only requires that all replicas are eventually consistent
  - If no further updates to a data happens, all reads of the data at any replica should eventually get the same value
  - Temporarily, different clients can see different values
    - Say client X updates at replica A, and client Y reads from replica B before the update by X propagates to B
  - Temporarily, even the same client can see different values
    - Say client X updates at replica A, and then reads from replica B before the update propagates to B
    - So may not even guarantee that a single client always sees its last write
  - Other intermediate models exist
  - Good if most operations are read, writes are infrequent, and some temporary inconsistencies can be tolerated
    - Good for many applications. Ex. DNS, NIS,….

Implementing Consistency Models

Replication Architecture

- Consistency models are fine, but how do systems implement them?
  - Depends on replication architecture and the specific model

Replication Architecture:

- **Passive Replication**
  - All requests made to a single replica (primary)
- **Active Replication**
  - Requests made to all replicas

Passive Replication

- Each client requests to a single replica (**primary**)
  - A unique identifier assigned by primary for each request
- Other replicas are **backup**
- Master-slave like relation between primary and backups
- Reads are returned from primary
- On write,
  - Primary executes the write and sends the updated state to all replicas
  - Receive reply from all replicas
  - Reply success to client
- Primary also sends periodic heartbeat messages to all backups to indicate it is alive

Passive Replication

- If primary dies (no heartbeat message detected at backup)
  - Backups elect a new leader that starts to act as primary
  - Client may fail to access a service during the duration between primary crash and new primary election (**failover time**)
- Need to ensure
  - Exactly one primary at all the times (except failover time)
  - All backups agree on the primary
  - No backups respond to client requests
- **Problem**: what happens if there is a failure during update of replicas?
- Consistency models enforced in passive replication
  - Linearizability, as primary acts as sequencer, serializing all access to the data
  - Enforcing Linearizability implies sequential consistency

Active Replication

- No master-slave relation among replicas
- A client makes requests to all replicas
  - In practice, client can send request to one replica, that replica can act as front end to send requests to all replicas
  - Client must know what replica to go to if the front end fails
- All replicas replies to client, client can take
  - First response for crash failure model (requires f+1 replicas to tolerate f faults)
  - Majority for byzantine faults (requires 2f+1 replicas to tolerate f faults)
- Need to ensure that all replicas agree on the order of client requests
  - If all requests are applied in the same order at all replicas, their final state is consistent
  - Consensus problem

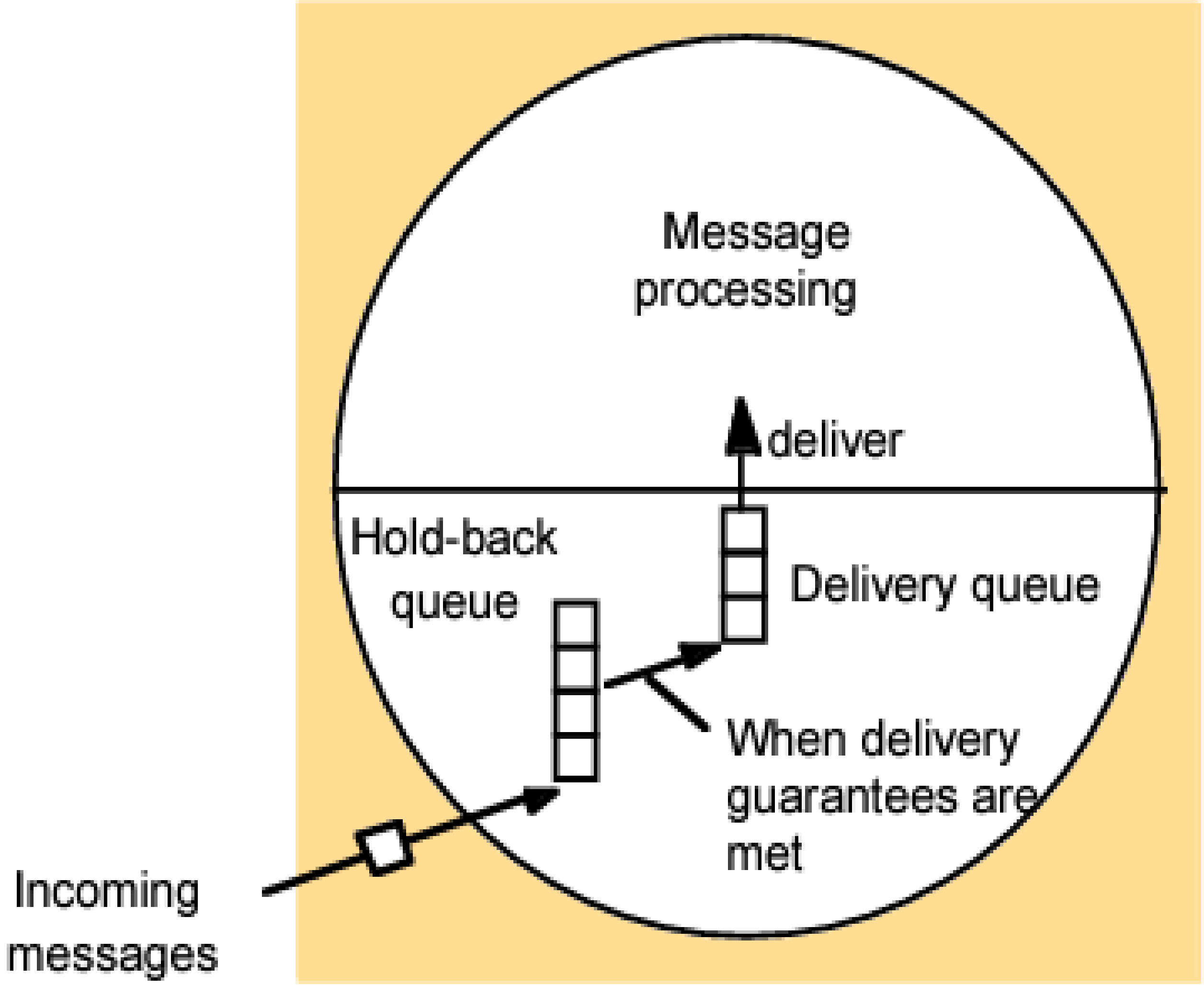
State Machine Replication

- A general strategy proposed to build fault-tolerant systems by replication
  - Basis for active replication in practice
- Each replica is represented by a state machine
- All replicas start with the same initial state
- Client requests are made to the state machines
- Need to ensure
  - All non-faulty state machines receive all requests (**Agreement**)
  - All non-faulty state machines processes the requests in the same order (**Order**)

Ensuring Agreement and Order

- Using atomic multicast
  - Read/write request sent to all replicas using atomic multicast
  - A communication primitive that delivers messages to multiple groups of processes according to some total order
    - **Total order:** Each process agree on the same order of the message (may be different from FIFO or Causal Order)
  - How to implement atomic multicast?
    - Incoming messages are held back in a queue until delivery guarantees can be met (**hold back queue)**

Hold Back Queue



Ensuring Agreement and Order

- Using atomic multicast
  - Read/write request sent to all replicas using atomic multicast
  - A communication primitive that delivers messages to multiple groups of processes according to some total order
    - **Total order:** Each process agree on the same order of the message (may be different from FIFO or Causal Order)
  - How to implement atomic multicast?
    - Incoming messages are held back in a queue until delivery guarantees can be met (**hold back queue)**
  - Atomic multicast is equivalent to consensus
    - Processes need to agree on the order of the messages
- Using other specialized consensus protocols (Paxos/Raft)
  - We have seen Raft

Implementing Linearizability

- Client makes read/write request
- Read/write request sent by local replica to all others using atomic multicast
- On receiving this, replica servers (a) update copy on write and send back an ack, (b) only send an ack on read
- On completion of total order multicast,
  - The local copy is given to the client on read, or
  - A success message returned to client on write

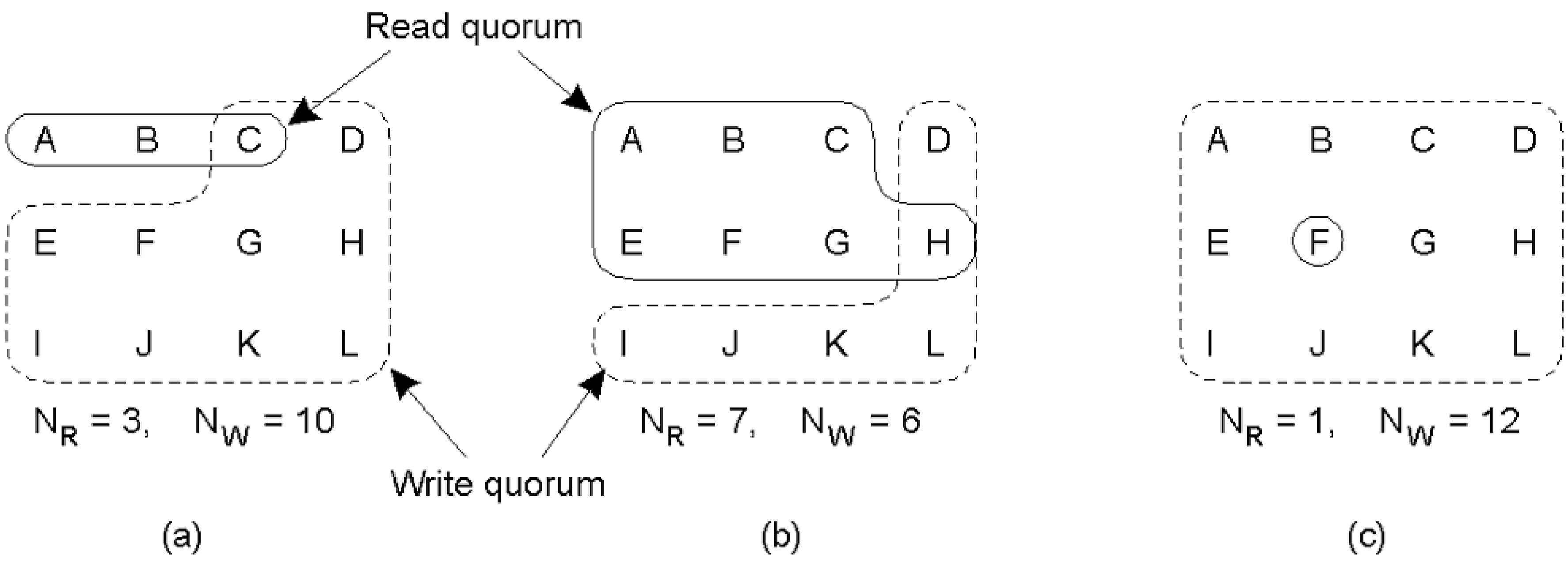
Implementing Sequential Consistency

- Using atomic multicast
  - Client makes read/write request
  - On read, just return the local copy (no atomic multicast)
  - On write, request sent by local replica to all others using atomic multicast
    - On receiving this, replica servers update copy on write and send back an ack
    - On completion of atomic multicast, a success message is returned to client on write
- Using Quorum-based protocols/Voting protocols

Voting Protocols

- Each replica is given a number of votes
- Let V = sum of all replica votes
- Choose **read quorum** Q_r and **write quorum** Q_w such that
  - $Q_r + Q_w > V$,   and
  - $2 * Q_w > V$
- To read, each replica must get Q_r votes
- To write, each replica must get Q_w votes
- Guarantees
  - A read and a write do not occur together
  - Two writes do not occur together

Choice of Quorums



- Three examples of the voting algorithm (each replica has one vote):
  a) A correct choice of read and write set
  b) A choice that may lead to write-write conflicts
  c) A correct choice, known as ROWA (read one, write all)

Voting Protocols

- Data items are tagged with a version
  - Incremented on each write
- To read data
  - client gets read quorum from replica sites
  - chooses the copy with the highest version number from those replicas
  - Most updated copy (why?)
- To write data
  - Client gets read quorum from replica sites
  - Chooses the copy with highest version number (say t)
  - Client gets write quorum from replica sites
  - Writes the data with version number > t in all replicas of write quorum

Two Special Cases

- Let N = no. of replicas
- **ROWA (Read One Write All)**
  - Each replica has one vote, $Q_r = 1$, $Q_w = N$
  - Fast reads, slow writes
  - Cannot tolerate even one replica failure for writes
- **RAWO (Read All Write One)**
  - Each replica has one vote, $Q_r = N$, $Q_w = 1$
  - Slow reads, first writes
  - Cannot tolerate even one replica failure for reads
- **Majority**
  - Each replica has one vote, $Q_r = Q_w = N/2 + 1$
  - Equal read/write overhead

Some Questions

- How to assign votes?
  - More reliable servers can be assigned more votes
  - More powerful servers may be assigned more votes
- What if one or more replicas fail?
  - No combination of votes may satisfy the quorum constraints
- What if there is a network partition?
  - No majority may exist in any of the partitions

Problems with Sequential Consistency

- Not scalable
  - Atomic Multicast requires all replicas to contact all other replicas
  - Voting based protocols still require a replica to contact a large no. of other replicas (majority in the worst case)

- Good for small systems that require such strong consistency

- Many systems do not need such strong consistency guarantees

Implementing Eventual Consistency

- Easy if a client always connects to a single replica
- Hard otherwise
  - Ex. mobile systems
- Generally, goal is to ensure that all replicas have the same state eventually
  - Epidemic protocol to replicate
  - Replication topology
    - Form a replication topology to decide who replicates from who
  - Scheduled Replication
  - Push vs. pull models
  - Gossip protocol

Replica Placement

- Mirroring
  - Static replicas created a-priori
  - May mirror all data or part depending on need
- Server-generated
  - Dynamic replicas created by a server when load increases
- Client caching
  - Replicas created by caching frequently used data at/near client
- Pros/Cons of each? When should you use what?

Happy Learning