DSC204A: Scaleable Data Systems Spring 2025

26: DISTRIBUTED CONSENSUS WITH RAFT

Instructor: Umesh Bellur *Scribe: Bhaavya Naharas, Caroline Zhang, Shreyasi Nath*

Recap

- **Logical Time**: In distributed systems, logical clocks (e.g., Lamport timestamps) are used to capture the order of events without relying on physical clocks. This helps reason about causality between operations.
- **Lack of Shared Memory**: Unlike centralized systems, distributed systems do not have a global shared memory. Processes communicate and coordinate through message passing, which introduces challenges in synchronization and consistency.
- **Message Ordering via Logical Time**: To maintain consistency and detect causality, distributed systems often rely on logical timestamps to order messages. This allows processes to agree on the sequence of events even in the absence of synchronized physical clocks.

Consistency and Replication in Distributed Systems

Transparency and Resource Abstraction

In distributed systems, two key abstractions help users interact with replicated resources seamlessly.

- **Location transparency** ensures that users do not need to know where a resource, such as a file chunk, is physically stored.
- **Replication transparency** hides the fact that a resource may be accessed and modified concurrently by multiple users. A classic example is Google Docs, which enables real-time collaboration without requiring users to manage synchronization.

Motivation for Replication

Replication serves two primary purposes: availability and performance. Redundancy through replication ensures that if a node or connection fails, the system can continue operating by using alternative replicas. Performance improves when replicas are placed closer to users, reducing latency and enabling faster access to data.

Challenges Introduced by Replication

While replication brings benefits, it also introduces significant complexity. The key challenges include:

- **Consistency**: How to ensure all replicas reflect the correct and up-to-date state.
- **Failure Handling**: Addressing issues like partial writes or dropped messages.
- **Synchronization**: Coordinating updates across nodes.

Consistency Scenarios and Examples

Suppose a shared variable X is replicated across multiple nodes. If process P_1 sets $X = 5$ and process P_2 concurrently sets $X = 7$, the system must define what value a subsequent read should return. Without defined semantics, the behavior is unpredictable, motivating the need for a well-defined consistency model.

Consistency Models

A consistency model defines the rules by which updates to shared resources are observed across processes. The strength of a model depends on how quickly updates propagate and the order in which changes are observed. Stronger models simplify application development but are harder to implement and more expensive in terms of system resources.

- Interleaving in Consistency Models refers to the global sequence of operations as perceived by processes in the system.
- In **sequential consistency**, all processes must observe the same interleaving, though it need not match real-time order.
- An interleaving is **valid** if it respects:
  - The program order of operations within each process.
  - The fact that once a process sees a newer value, it should not later see an older value.
- There is no global total order required—only a consistent view across processes.

Strict Consistency

Strict consistency is the strongest model. It requires that every read operation returns the result of the most recent write, regardless of which node the read is performed on. This implies instantaneous update propagation and the existence of a global clock—both of which are impractical.

For example, as shown in Figure 1, if process P_1 sets $X = 5$, any process reading X an instant later must return 5. If another process sees a different value, the system has violated strict consistency. Although strict consistency is not realizable in practice, it serves as a useful benchmark.

Consider an inventory management system. Suppose process P_1 updates the inventory of DVD players to $D = H$ (high), and then process P_2 reads D and observes the new value. However, if another process still reads $D = L$ (low) after the update, it implies that the update has not yet propagated, violating strict consistency.

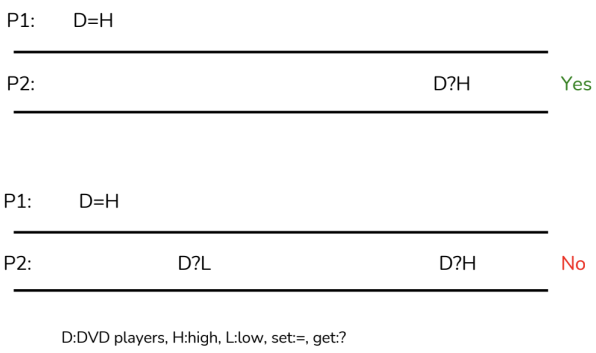

Figure 1: Strictly consistency

Implementable Model: Causal or Sequential Consistency

Causal and sequential consistency are more relaxed than strict consistency but are achievable in practice. These models ensure that writes from a single process are observed in the same order across all processes. For example, if a process writes $X = 5$ followed by $X = 8$, no other process should observe the value 8 before 5. This model preserves the logical order of operations without requiring instant propagation.

While causal consistency does not guarantee immediate visibility, it ensures that the relative ordering of writes is maintained. This makes it a practical compromise for many distributed systems.

Sequentially Consistent

Sequential consistency guarantees that operations by a single process appear in order, and all processes see operations in the same sequence, although this sequence may not reflect real-time order. The important constraint is that once a process observes a newer value, it must not observe an older one afterward. This ensures a coherent view of the system across processes.

Consider the examples shown in Figure 2 (a) and (b). In (a), process P3 reads value b and then a, while P4 does the same. This ordering is inconsistent across processes, violating sequential consistency. In contrast, (b) shows both processes observing the same sequence (a then b), maintaining sequential consistency. Achieving this requires delaying certain message deliveries to preserve a consistent interleaving.

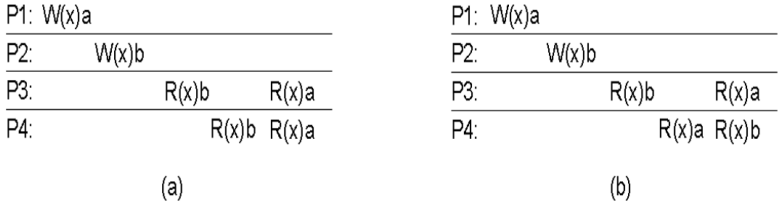

Figure 2: Sequential consistency

Cost of Sequential Consistency

Implementing sequential consistency introduces overhead. Systems must coordinate reads and writes to maintain ordering, which may involve delaying operations or buffering updates. This increases latency and complexity, particularly in geographically distributed environments. Read/write operations must wait longer than basic message transfer times to ensure correctness.

Causal Consistency

Causal consistency strengthens guarantees by enforcing ordering on causally related operations across processes. If one process reads a value and subsequently performs a write based on that value, the write is causally dependent on the read. Such dependencies must be preserved system wide.

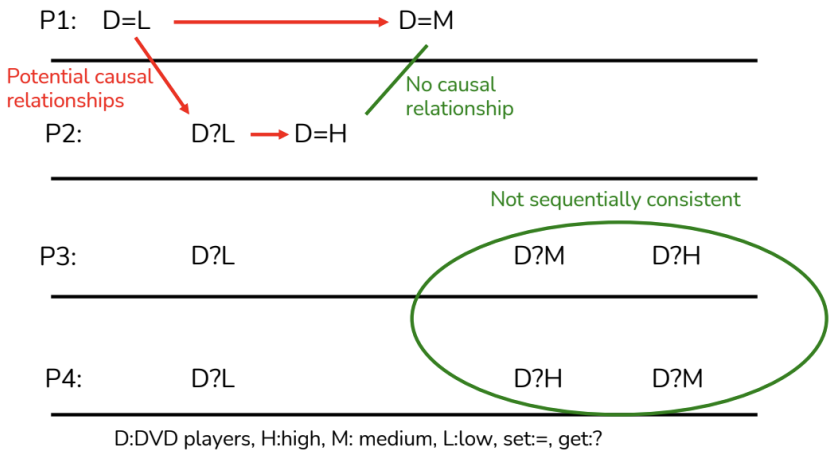

Figure 3: Causal Consistency - Case 1

In Figure 3, P1 sets D = L, which is read by P2 before it sets D = H. These two writes are causally related. Any process that sees D = H must have already seen D = L. However, Figure 4 illustrates a causal consistency violation: one process sees D = H before D = L, reversing the causal order and breaking the model.

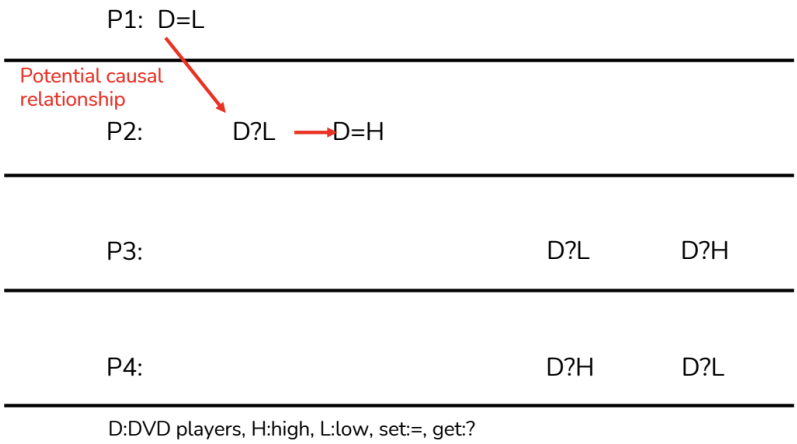

Figure 4: Causal Consistency - Case 2

Causal consistency respects FIFO order per process and adds ordering constraints across causally linked writes, but it does not enforce order for concurrent (unrelated) writes. For these, the system makes no guarantees that is different processes may observe different orderings.

Linearizability

Linearizability is a stronger consistency model that ensures operations appear to occur atomically in real time, respecting both total order and timing constraints. If one operation completes before another starts, all processes must observe them in that order. Unlike sequential consistency, linearizability incorporates real-time ordering into the model.

To enforce linearizability, systems need synchronized clocks or a way to infer the start and end times of operations. This model is useful in scenarios where users expect strong consistency aligned with actual time, such as financial transactions.

Refer to Figure 5: although both cases are sequentially consistent (all processes agree on an order), only the right-hand case is linearizable. The left-hand case shows a read of b followed by a read of a, which implies observing an older value after a newer one violating linearizability.

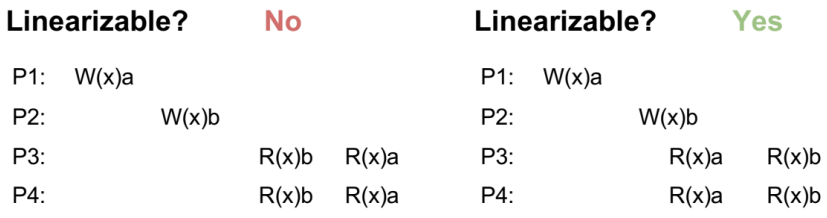

Figure 5: Linearizability

Types of Data-Centric Consistency

These consistency models focus on the data store itself and ensure consistency across all clients. They are distinct from client-centric models, where each client may have a different view of the data based on local interactions.

In data-centric consistency:

- The system ensures that reads and writes behave predictably for all clients.
- Programmers can use explicit synchronization (e.g., barriers or locks) to ensure consistency for critical operations.

For instance, issuing a synchronization operation S before a read ensures that the most recent value is returned. This may involve querying multiple replicas and returning the majority or freshest value.

Eventual Consistency

Under eventual consistency, the system guarantees that, in the absence of new updates, all replicas will eventually converge to the same state. This model does not ensure immediate consistency and is common in systems where availability and responsiveness are prioritized over strict correctness.

Applications like Gmail or social media feeds can tolerate temporary inconsistencies. For example, seeing a reply before the original email. Systems like DNS or content delivery networks (CDNs) also follow this model, trading off consistency for scalability and performance.

Exercise

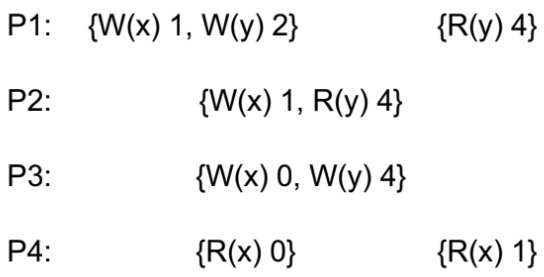

Figure 6: Exercise 1

Consider the operations by processes P1 through P4. Although all consistency models eventually return a ”Yes” for safety, the execution is not strictly consistent because a read of x returns a stale value that does not reflect the most recent write. However, the system satisfies causal, sequential, linearizable, and strict conditions, as each operation respects the causal and program order constraints, and no read observes a write that hasn't logically occurred yet. This illustrates how different consistency models provide layered guarantees, with strict serializability being the most demanding and eventual consistency the most permissive.

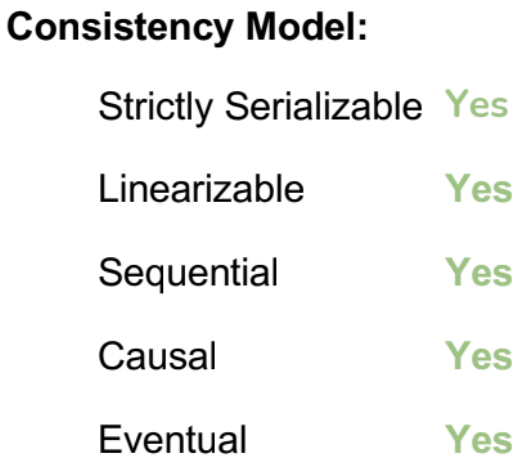

Figure 7: Exercise 1 - Results

Next Steps

- Study Paxos vs. Raft: leader election, log replication.
- Implement a toy Raft cluster for hands-on experience.
- Explore hybrid models: e.g., Google Spanner's TrueTime for cross-datacenter strong consistency.