Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Prof. Anna Arpaci-Dusseau**
and **Prof. Ioannis Liagouris** for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

— 1 —

Introduction

1.1 Consistency Models

1.1.1 Introduction

Moving on from consensus models, which handle agreement on data values, we now shift our focus to the ordering of operations and their validity:

Definition 1.1: Consistency Model

Within a distributed system, a consistency model acts as a contract between systems on valid operation ordering (e.g., reads or writes) on shared data within the network.

These models are critical for ensuring that a system behaves as expected when replicating data across multiple nodes. [2]

Definition 1.2: Global Total Order

A Global Total Order is a sequence of operations that is agreed upon by all nodes in a distributed system. This means that such sequence given some client inputs, could reproduce the same state on a single machine.

Now we begin to list some common consistency models:

Definition 1.3: Strong Consistency

The client observes that all nodes *appear* to agree on the order of execution. This means all node reads of shared data are identical (Global Total Order of operations).

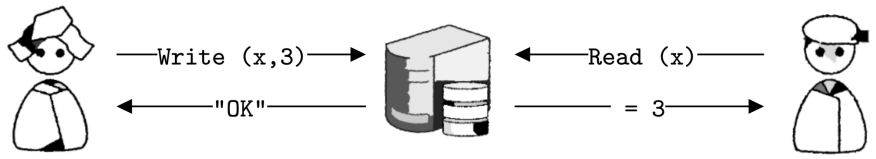
Definition 1.4: Weak Consistency

The client **temporarily** observes that nodes disagree on shared data values at some point in time (no Global Total Order).

Theorem 1.1: Strong Consistency vs Weak Consistency

Even though strong consistency is desirable, it can be costly on the network and difficult to implement. Weak consistency is easier to implement and may provide better performance, but at the cost of data integrity and difficulty in debugging interactions.

Consider this example interaction, which interaction has the weakest consistency model?



Find The Weakest Consistency

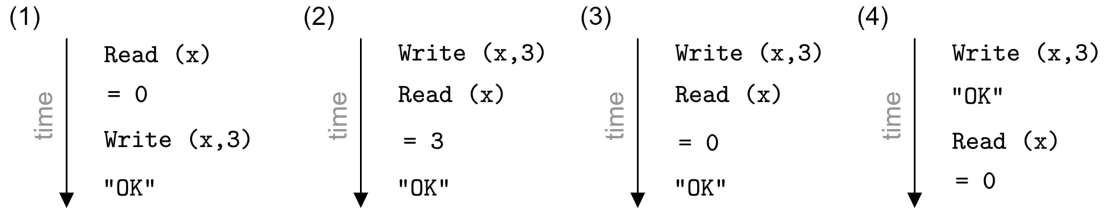


Figure 1.1: (1) Strong Consistency, as the read x could be 0, and the write responds with "OK". (2) This is okay, as our write of 3 is read. The "OK" does come a little late, but its order isn't bizarre. (3) Still okay, as the "OK" comes after the read of 0, meaning the write could not have happened yet. (4) The weakest, as our read completely disregards our acknowledged write of 3.

1.1.2 Strong Consistency Models: Linearizability & Sequential Consistency

We discuss two strong consistency models, Linearizability and Sequential Consistency:

Definition 1.5: Linearizability

Replicas produce a Global Total Order, which preserves real-time ordering of events. Moreover, every read must return the value of the **most completed** write. In particular:

- If operation A completes before B, then $A \to B$ in real-time.
- If tasks A and B overlap, there is no real-time ordering.

Theorem 1.2: Raft and Linearizability

Raft's leader-commit design provides Linearizability, **except** in specific scenarios such as:

- A committed log entry is lost within the majority of the cluster.
- The newly elected leader somehow bypasses the Leader Completeness property.

In all other cases, Raft is considered to have a strong consistency model.

Consider the following examples and determine whether it is linearizable:

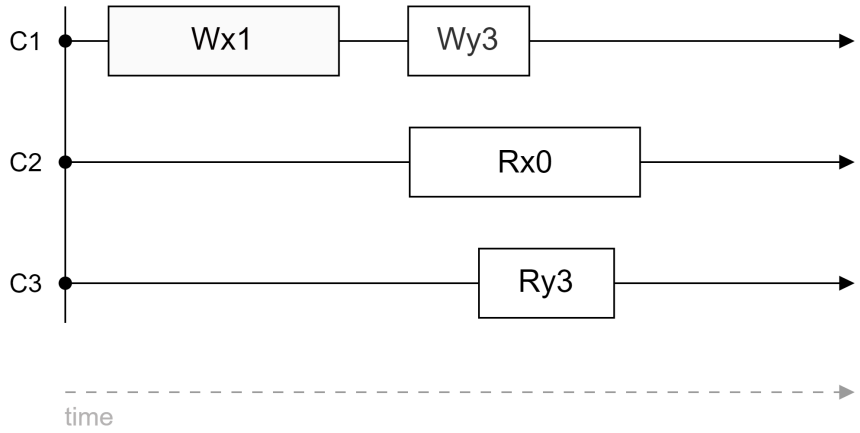


Figure 1.2: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx0$ reads, "0 read from x." This system is not linearizable because the read $Rx0$ should have returned 1, in respect to real-time.

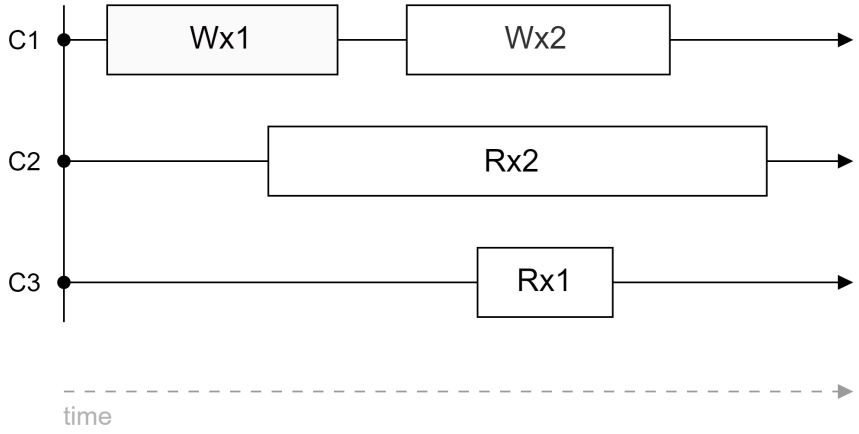


Figure 1.3: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x." This system is linearizable. This is because each box represents a period of time when the action could take place. Therefore, Wx1 and Rx1 can happen, while still having enough time for Wx2 and Rx2 to occur.

The **next page** includes an elaboration of the above.

Here we elaborate on Figure (1.3):

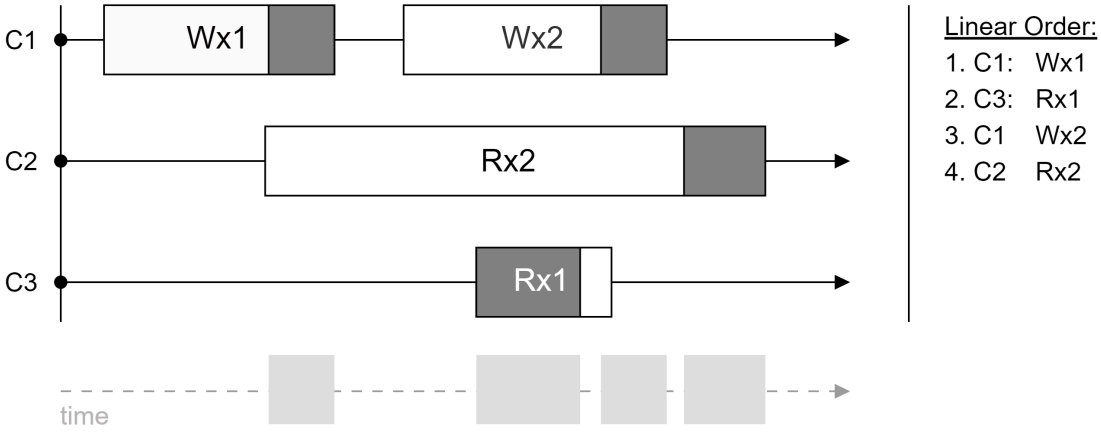


Figure 1.4: The gray boxes represent a mutex on the shared data, x. Here we see the order of operations, Wx1, Rx1, Wx2, and Rx2, from which **no overlaps occur** and logically make sense.

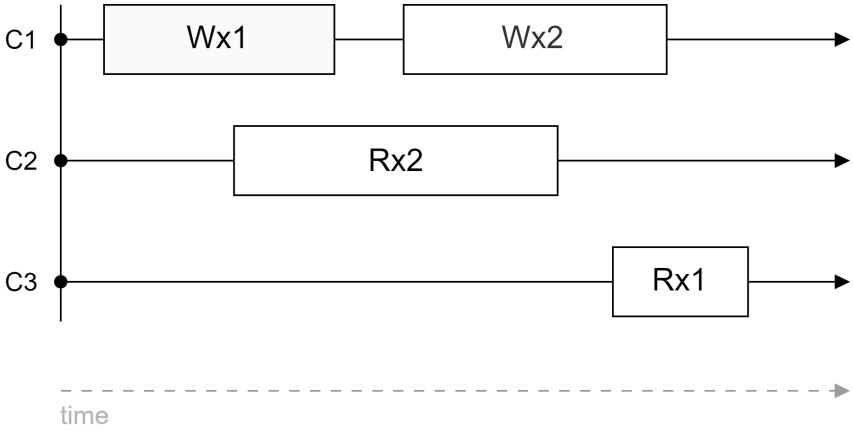


Figure 1.5: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x." This system is not linearizable. The Rx1 is not possible as it's too far disjointed from Wx1 after being overwritten by Wx2.

We've talked about another replication method before that is also linearizable:

Theorem 1.3: Chain Replication and Linearizability

Chain Replication is linearizable. The duration of a write operation extends until the tail, from which an acknowledgment is sent. Reads also adhere to reading the last completed write, and is thus—a strong consistency model.

The next model is Sequential Consistency:

Definition 1.6: Sequential Consistency

Weaker than Linearizability. All replicas execute all operations in **some** Global Total Order. Each client **observes the same order** once such order is agreed upon. Therefore a single machine may replicate the system if given such order. In particular:

- If a system process issues A before B, then $A \to B$ in the Global Total Order.
- If A and B happen on different processes, there is no real-time ordering (concurrent).

Consider the following examples and determine whether it is sequentially consistent:

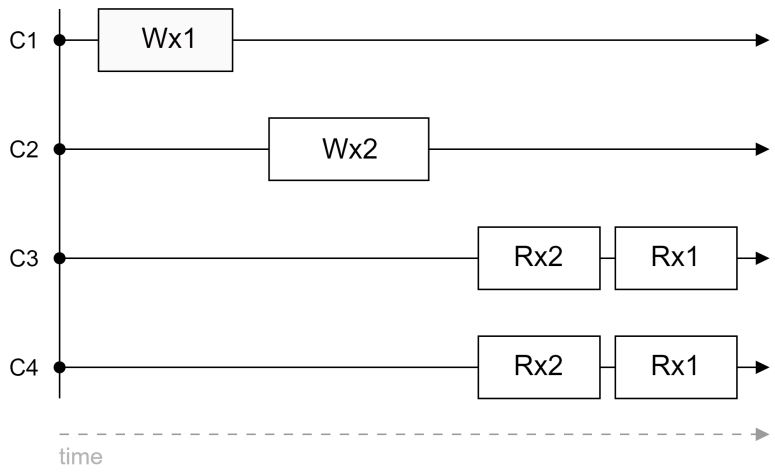


Figure 1.6: A distributed system with client C1, C2, C3, C4 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx0$ reads, "0 read from x." This figure may or may not be sequentially consistent.

Next page includes an elaboration of the above.

We elaborate on the previous example:

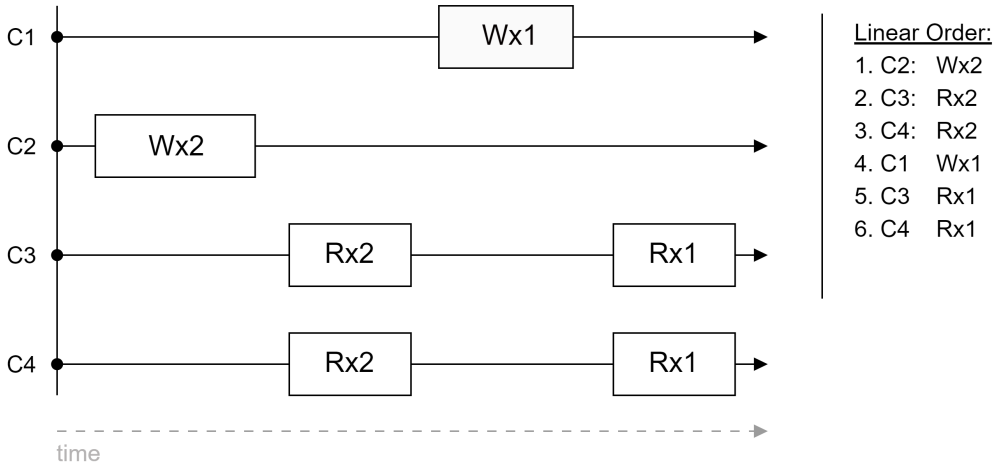


Figure 1.7: A distributed system with client C1, C2, C3, C4 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x.". This figure is sequentially consistent as there exists some Global Total Order that can be agreed upon. Here listed, Wx2, Rx2, Rx2, Wx1, Rx1, Rx1.

And lastly, consider the following example:

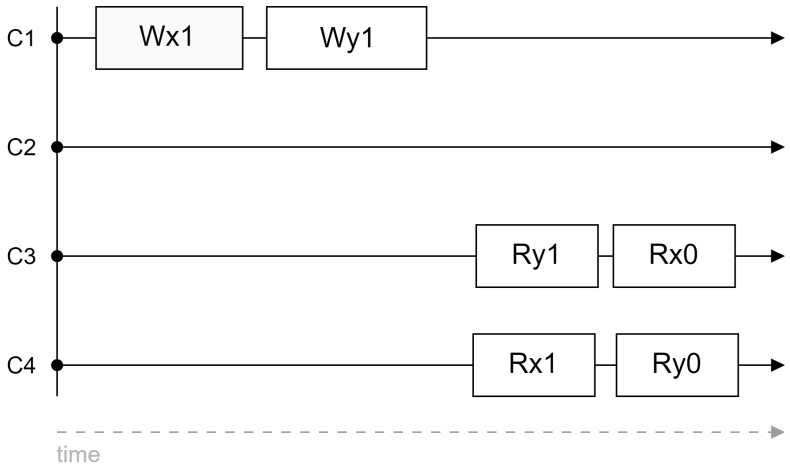


Figure 1.8: A distributed system with client C1, C2, C3, C4 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x.". This figure is not sequentially consistent as there is no Global Total Order that can be agreed upon. Here Rx0 cannot happen after Ry1 because of the relation Wx1 $\rightarrow$ Wy1.

Consider the following theorem:

Theorem 1.4: Linearizability vs Sequential Consistency

If a system is linearizable, it is also sequentially consistent. As, adhering to real-time order naturally satisfies sequential consistency. **However**, the reverse is not true. In particular:

- **Linearizability:** Relies on real-time.
- **Sequential Consistency:** Relies on program order.

1.1.3 Handling Shared Data via Mutex: Release & Lazy-release Consistency

Consider the following **Weak** methods of handling shared data:

Definition 1.7: Release Consistency vs. Lazy-release Consistency

These methods require explicit use of locks to propagate updates:
Release Consistency: Push updates to all nodes after releasing the lock.
Lazy-release Consistency: Push updates to all nodes after the lock is acquired. *Though this may provide less stress to the system, if the client does not lock data, it may become stale (weak consistency). Alternatively, if the client locks all data (strong consistency) though it may be costly.* **These both are weak consistency models.**

Determine whether the following system is release or lazy-release consistent:

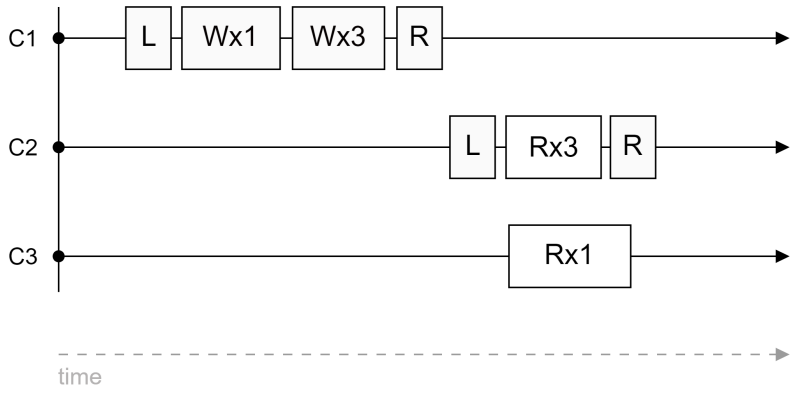


Figure 1.9: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x." With L (lock) and R (Release). This is Lazy-release consistent, as C3's read of x wasn't updated to 3 in absence of a lock.

1.1.4 Weak Consistency Models: Causal & Eventual Consistency

We now discuss two weak consistency models, Causal Consistency and Eventual Consistency:

Definition 1.8: Causal Consistency

Weaker form of Sequential Consistency. That the Global Total Order of operations adhere logically to their causal dependencies. In particular:

- If A causes B, then $A \rightarrow B$ should be in the Global Total Order.

I.e., Causal Consistency is just Sequential Consistency, differing slightly in that Clients in Causal Consistency may observe different Global Total Orders.

For example, only one of these sentences makes causal sense:

(1)

1. Alice: Lunch?
2. Bob: Yes.
3. Carl: No.

(2)

1. Bob: Yes.
2. Alice: Lunch?
3. Carl: No.

(3)

1. Alice: Lunch?
2. Carl: No.
3. Bob: Yes.

Here, statements (1) and (3) are causally consistent, as the responses are in order of the question. Take that same intuition for this next example and determine whether the below system is causally consistent:

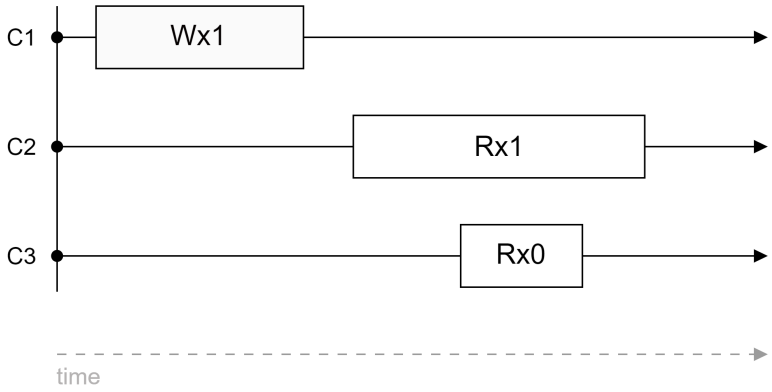


Figure 1.10: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x." This system is causally consistent as the Global Total Order of operations adhere to their causal dependencies. Here, Wx1 $\rightarrow$ Rx1, where Rx0 must come before Wx1.

Consider the following example and determine whether it is causally consistent:

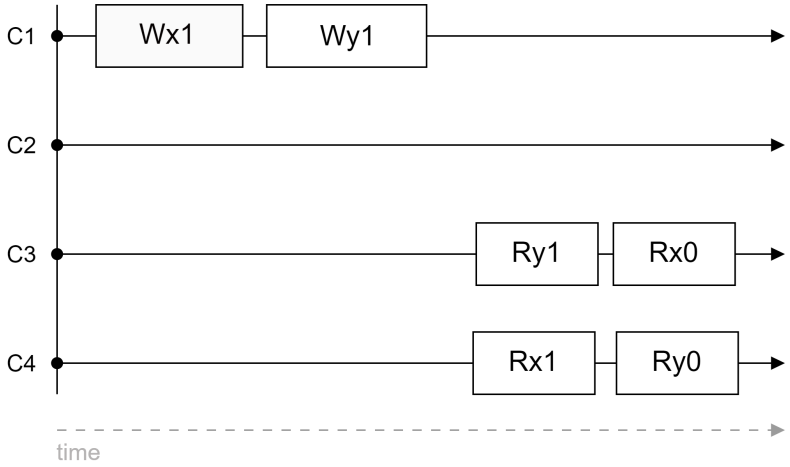


Figure 1.11: This A distributed system with client C1, C2, C3, and C4 interactions. This is not causally consistent, as $Wx1 \rightarrow Wy1$, but $Ry1 \rightarrow Rx0$ does not adhere to this relationship.

Definition 1.9: Eventual Consistency

Weaker than Causal Consistency. Given there are no new writes, replicas will **eventually** agree on the same value after some period of time.

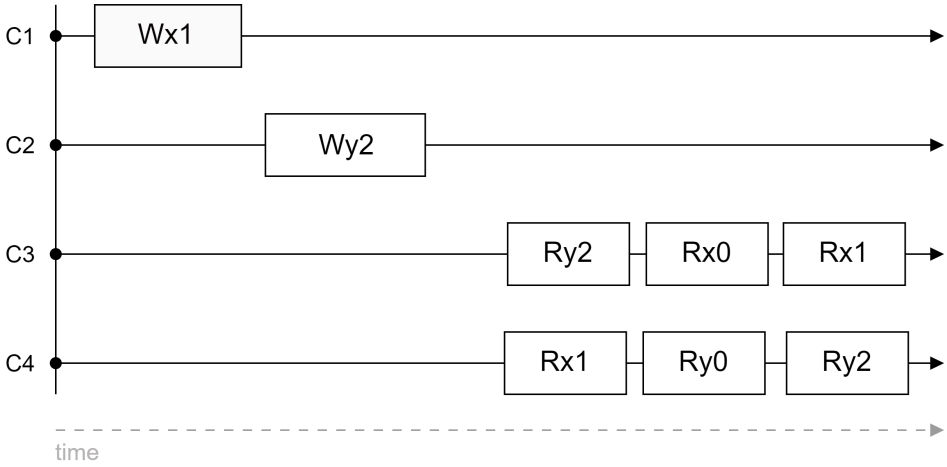


Figure 1.12: This system is eventually consistent as it eventually agrees on x and y.

Causal Consistency implies two properties:

Theorem 1.5: Causal Consistency → FIFO & RYW

Causal Consistency implies FIFO (First-In-First-Out) and RYW (Read-Your-Writes):

- **FIFO:** All writes are read in the order they were issued.
- **RYW:** If the client writes to x, then their next read of x must return the value they just previously wrote.

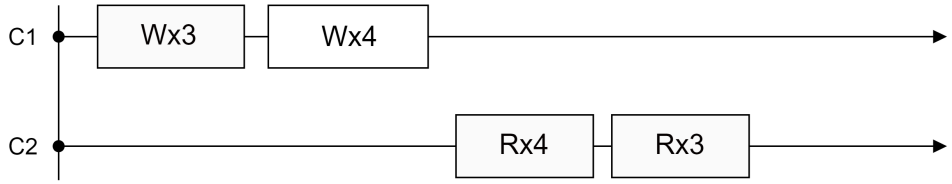


Figure 1.13: This system is not FIFO nor Causally Consistent. Writes **must be read in monotonic order**. Hence, Rx3 must come before Rx4.

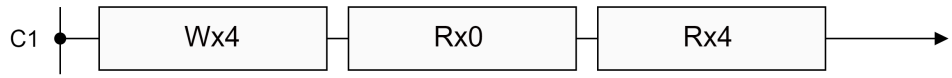


Figure 1.14: This system is not RYW nor Causally Consistent. As the next read should be Rx4 alone. The order, Rx0 → Wx4 → Rx4, satisfies RYW and is Causally Consistent.

Tell whether the following is linearizeable, sequential, causal, and or read your writes.

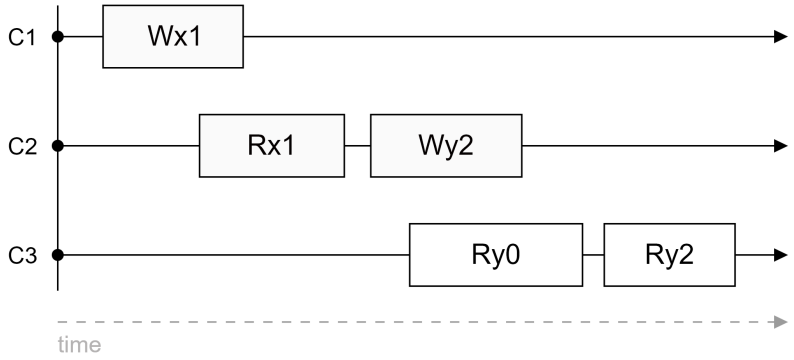


Figure 1.15: This system is linearizable, sequential, causal, and RYW.

Recall that linearizability deals with real-time ordering of locks.

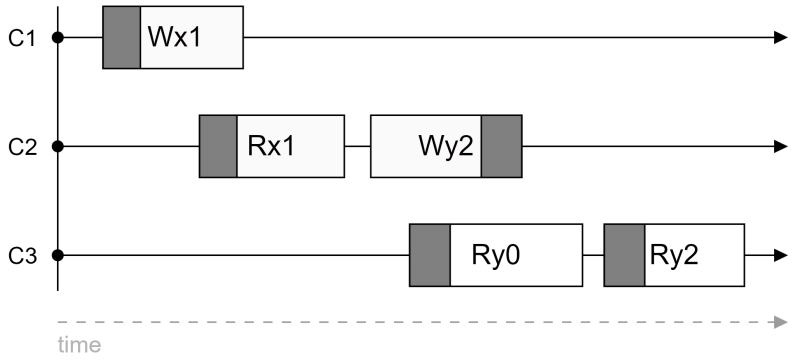


Figure 1.16: Here, locks are claimed in non conflicting order, which follows a real-time order.

To emphasize:

Theorem 1.6: consistency Model Implications

The following implications hold among consistency models:

linearizability → sequential → causal → read-your-writes → FIFO → eventual

1.2 Transactions and Concurrency Control

1.2.1 Optimistic Concurrency Control (OCC)

Say the backbone of our stock trading application is a distributed database. The system may conduct complicated stock trades based on server stock prices.

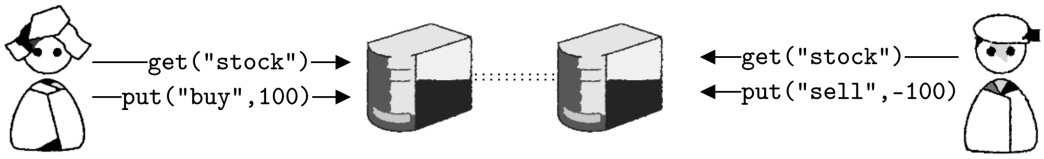


Figure 1.17: Two users checking the stock prices and making a trade based on such information.

It is critical that the stock price is consistent through all servers, and more important that if any trade fails, the system can recover to a consistent state.

Definition 2.1: Transaction

A **transaction** is a sequence of operations that are treated as a single unit of work. A transaction must satisfy the **ACID** properties:

- **Atomicity**: A transaction is either fully completed or dropped entirely.
- **Consistency**: A transaction must leave the database in a consistent state.
- **Isolation**: Transactions must be isolated from each other.
- **Durability**: Once a transaction is committed, it remains so even after system failure.

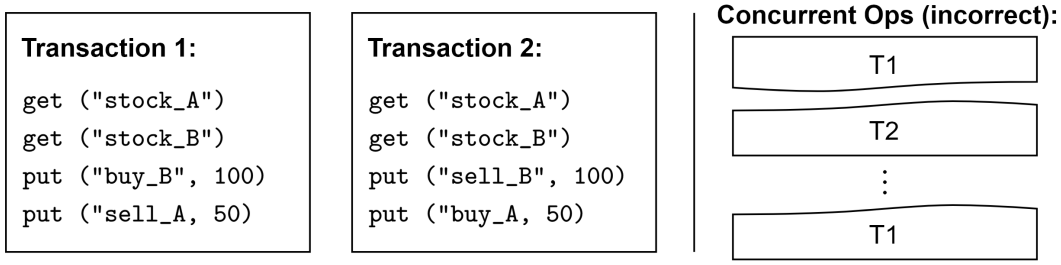


Figure 1.18: Two transactions which whose operations are interleaved.

Interleaving transactions **violates the isolation property**. This is problematic in Figure (1.18) as T2's (transaction 2) operations may depend on the server state before T1's transaction. Additionally partially completed transactions leave the system in an inconsistent state, violating **atomicity** (e.g., T1's "buy_B" fails, but "sell_A" succeeds).

We discuss another consistency model which will help us in this settings:

Definition 2.2: Serializability

Serializability is a strong consistency model that ensures the outcome of concurrent transactions is the same as if they were executed in some sequential (serial) order.

This differs from **linearizability**, which focuses on the real-time ordering of individual operations. Serializability instead concerns the logical order of **entire transactions**, independent of their timing.

However, **strict-serializability** does care about real-time ordering in addition to the logical order of transactions. This implies linearizability, but not vice versa.

We consider the following model to help us preform transactions:

Definition 2.3: Optimistic Concurrency Control (OCC)

Optimistic Concurrency Control (OCC) assumes conflicts are rare and proceeds without locking. It follows four main steps:

- **Prepare**: The system reads the transaction request and creates a backup or temporary copy of the state.
- **Modify/Validate**: The transaction modifies the temporary state. Then The system checks whether the transaction is **serializable**.
- **Commit/Rollback**: If valid, commit; Otherwise, abort transaction and rollback to previous state.

This only solves **isolation**, as it does **not** guarantee atomicity.

Definition 2.4: Transaction Coordinator and Database Servers

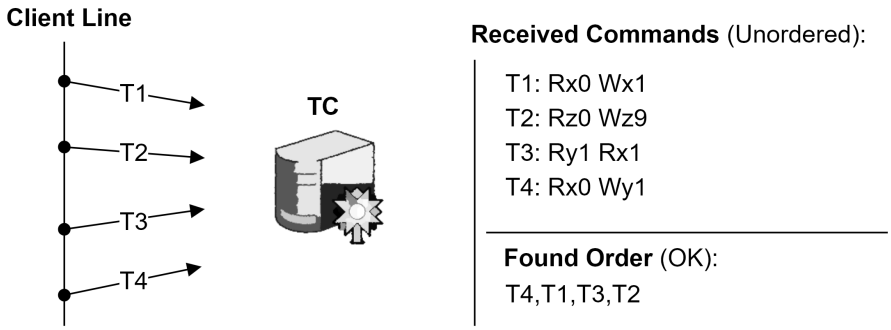
OCC maintains two necessary components:

- **Transaction Coordinator (TC)**: The validation server responsible for checking whether a transaction is **serializable**. It receives requests from clients and responds with either:
  - `OK`: if the transaction is serializable,
  - `ABORT`: if it conflicts with prior transactions.
- **Database Servers (DB)**: Receives transaction operations, executing it on local state. Then, on **OK**—commit state, on **ABORT**—rollback to the previous state.

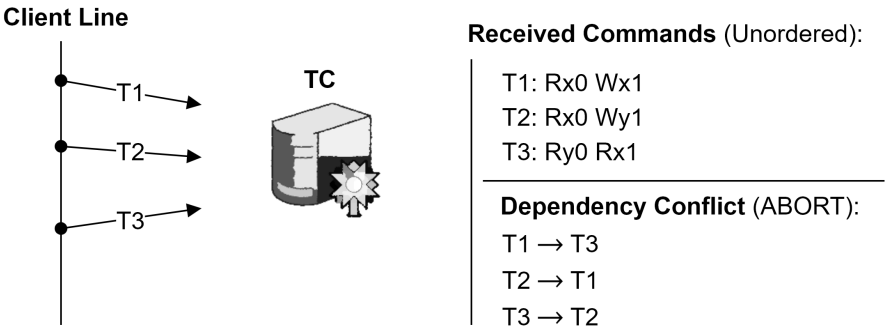
We consider one model, which where multiple clients interact with one TC.

Example 2.1: Centralized OCC

Consider these two examples with a single TC and multiple clients on the network line:



Here, clients on the line send transactions T1–T4 to the TC. The TC then checks the transactions for serializability. In this case an order is found (T4,T1,T3,T2), which the TC communicates to the DBs to commit.



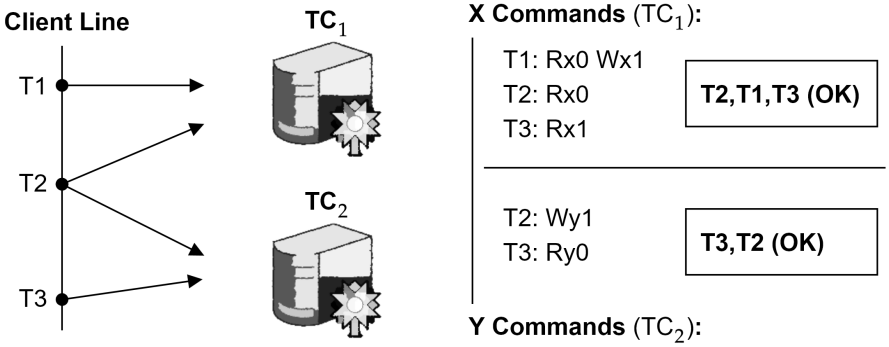
Here, transaction requests, T1, T2, and T3, do not have a serial order. As we build, T1 → T3 (T1 then T3) makes logical sense. Then T2 → T3, giving us the order T2 → T1 → T3; However, it appears T3 must come before T2. This causes a cycle (T2 → T1 → T3 → T2). Hence, the TC must must abort all transactions. ■

Tip: If familiar with **Directed Acyclic Graphs (DAG)**, one can think of the transactions as nodes and the edges as the dependencies between them. If the graph is a DAG, then there is some serial order (OK). If not, then there is a cycle, so we must abort.

Though we run into an issue when there are multiple TCs.

Example 2.2: Distributed OCC

Consider two TCs responsible for different parts of our system data.

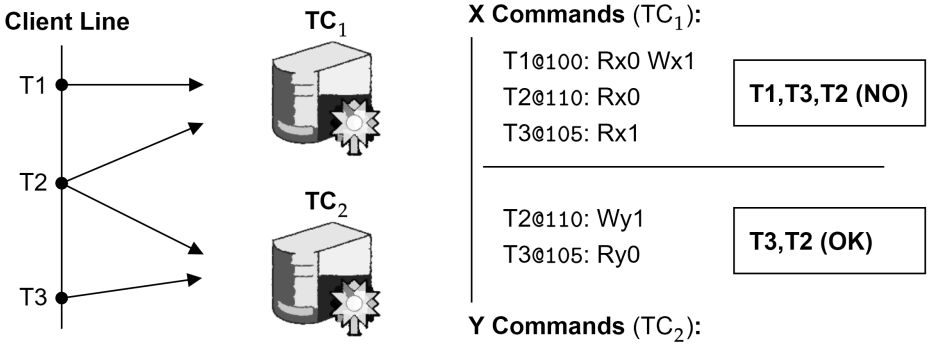


The problem occurs as TC_1 and TC_2 pick **different serial orders** for the transactions. ■

Theorem 2.1: Timestamping Distributed OCC

Timestamping is a method where each transaction is assigned a unique timestamp (ID), which aids order agreement between TCs. The downside: **unnecessary aborts.**

Example 2.3: Timestamping Distributed OCC



Timestamps (@#) are only serve as IDs. Here TC_2 OKs the order (T3,T2). TC_1 sees this, enforces the order, but is not able to serialize (T1,T3,T2). Hence, it aborts (NO). ■

1.2.2 Two-Phase Commit (2PC)

We introduce another method to help us with this problem, though it **does not ensure isolation**:

Definition 2.5: Two-Phase Commit (2PC)

Two-Phase Commit (2PC) ensures **atomicity**. The client sends the transactions to the DBs (participants). There after, the client tells the TC start the commit process, involving two phases:

- **Prepare Phase**: The coordinator sends a prepare request to all DBs; Each respond:
  - `YES`: if it can commit the transaction.
  - `NO`: if it cannot commit the transaction.
- **Commit Phase**: If all voted `YES`, the coordinator sends a `COMMIT` request to all DBs. If any participant voted `NO`, the coordinator sends an `ABORT`. The TC then responds to the client with the final outcome of the transaction.

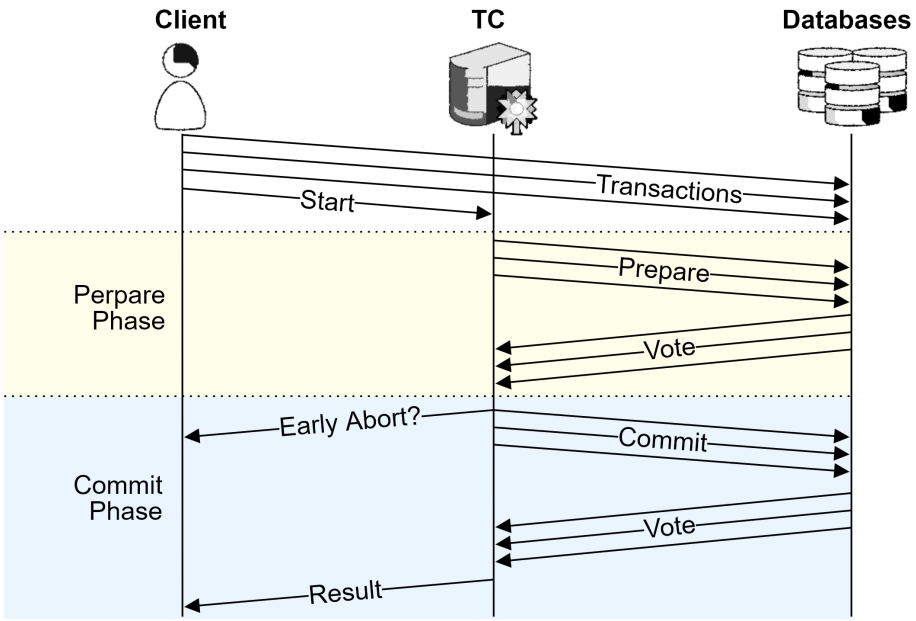


Figure 1.19: Two-Phase Commit (2PC) process. The client sets up the transaction with the TC and DBs. Then, the TC starts the commit process, starting with the prepare phase, and then ends with the commit phase.

Though there is a pitfall with this method:

Definition 2.6: 2PC Blockout

In most cases if there's a timeout the TC or DBs will abort the transaction. However, if the TC times out after a DB has voted `YES`, the DB is left in an uncertain state, and must **wait** for the TC to respond.

Preparation Phase	
Case	**Action**
TC timeout for yes/no vote	**Abort** — transaction coordinator did not receive votes in time.
DB timeout for prepare	**Abort** — database did not receive prepare request in time.
Commit Phase	
Case	**Action**
DB timeout for commit/abort and DB voted NO	**Abort** — since DB already voted NO, it's safe to abort.
DB timeout for commit/abort and DB voted YES	**Block** — DB must wait for TC decision to preserve atomicity.

Definition 2.7: 2PC Persistent & Volatile State

In the Two-Phase Commit (2PC) protocol, both the Transaction Coordinator (TC) and Database (DB) participants must persist critical information to recover correctly after a crash.

- **Database (DB):**
  - Must persist the result of any vote (`YES` or `NO`).
  - If the DB voted `YES` and then crashes, upon recovery it must contact the TC to learn the final decision (`COMMIT` or `ABORT`).
- **Transaction Coordinator (TC):**
  - Must persist the result of any vote and the final decision (`COMMIT` or `ABORT`).
  - If the TC crashes, it must resume the commit protocol from where it left off.

1.2.3 Three-Phase Commit (3PC)

2PC's main problem was **availability**. We can fix this by adding an additional phase:

Definition 2.8: Three-Phase Commit (3PC)

Three-Phase Commit (3PC) is an extension of 2PC that adds a third phase to avoid blocking in case of failures:

- **CanCommit Phase**: The coordinator sends a `CANCOMMIT` request to all participants. Each participant responds with either `YES` or `NO`.
- **PreCommit Phase**: If all participants respond with `YES`, the coordinator sends a `PRECOMMIT` request to all participants. Participants prepare to commit sending back an acknowledgment (`ACK`).
- **DoCommit Phase**: If all participants `ACK` the precommit, the coordinator sends a `COMMIT` request. Otherwise, it sends an `ABORT` request.

Theorem 2.2: 3PC Non-Blocking Nature & Self-Resolution

The reason this fixes the blocking issue in 2PC lies within the `PRECOMMIT` phase. In regular 2PC, if a DB votes `YES` it must wait for the TC to respond as it is uncertain whether another DB has committed or not.

In 3PC, such DB need not worry, as no other DB can commit until the TC sends a `PRECOMMIT` request. Henceforth, DBs can safely either terminate or resolve between themselves that if everyone else has the `PRECOMMIT` request, they can commit.

Theorem 2.3: 3PC Persistent & Volatile State

Both the TC and DBs may crash at any time, for which they must persist the state of the transactions. Though it is fully possible for the TC to only persist the original transaction request, and instead query **all** DBs for their state to recover the transaction.

Theorem 2.4: 3PC Tradeoffs

- (−) If the network becomes partitioned, inconsistencies may arise within self-resolution.
- (−) Adds another round trip to the commit process (3 in total).
- (+) Removes the blocking issue of 2PC, as the TC can always recover from a crash.

Below is a diagram of the 3PC process:

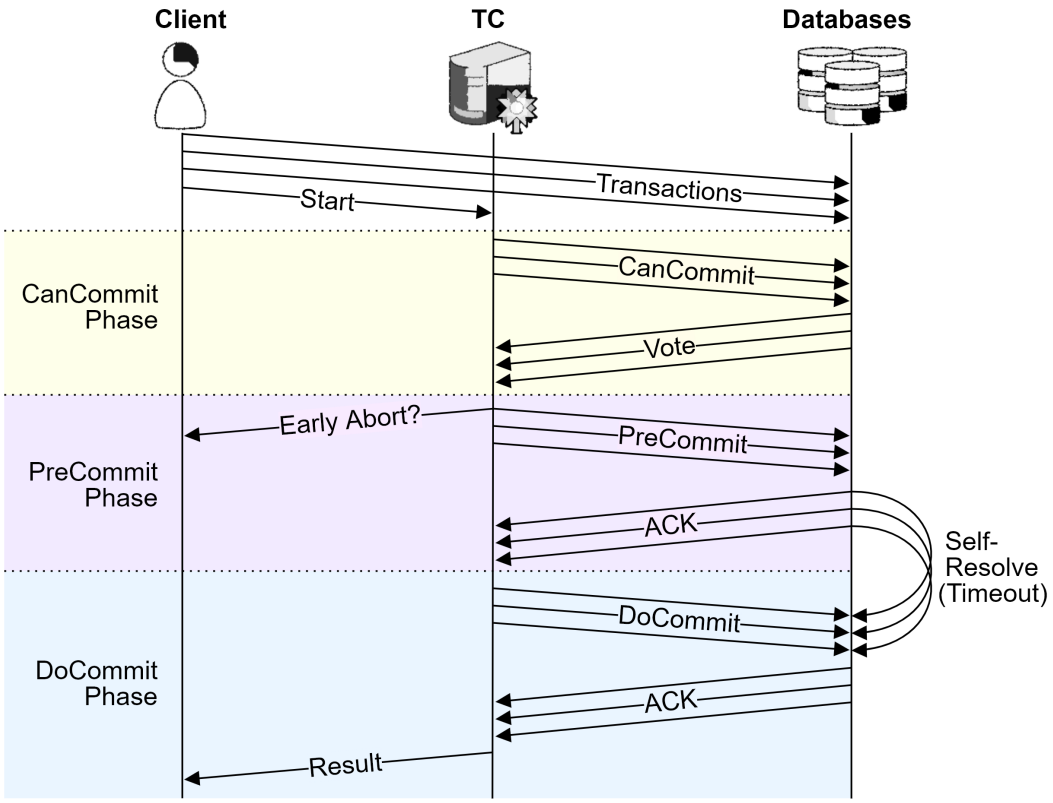


Figure 1.20: Three-Phase Commit (3PC) protocol message flow. The transaction progresses through three phases: **CanCommit**, where participants vote; **PreCommit**, where participants acknowledge readiness to commit; and **DoCommit**, where the final decision is executed. In the event of coordinator failure, participants may invoke **Self-Resolve** after a timeout, based on whether they received a `PreCommit` message, ensuring non-blocking recovery. **Note:** The Databases do not communicate with the client after resolution, though this could depend on the implementation.

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.

[2] ScyllaDB. Consistency models definition, 2025. Accessed: 2025-03-25.