Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Ioannis Liagouris**
and **Dr. Anna Arpaci-Dusseau** 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 Transactions and Concurrency Control

1.1.1 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.

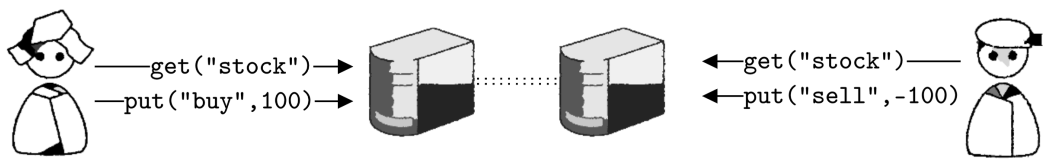


Figure 1.1: Two users checking the stock prices and making a trade based on such information.

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 1.1: 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.

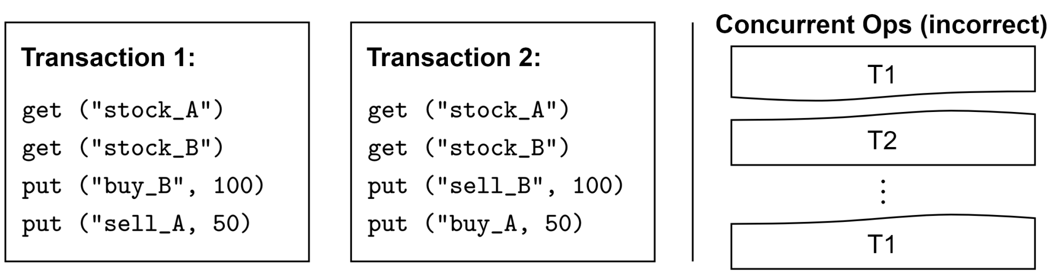


Figure 1.2: Two transactions which whose operations are interleaved.

Interleaving transactions **violates the isolation property**. This is problematic in Figure (1.2) 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 1.2: 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 1.3: 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 1.4: Transaction Coordinator and Database Servers

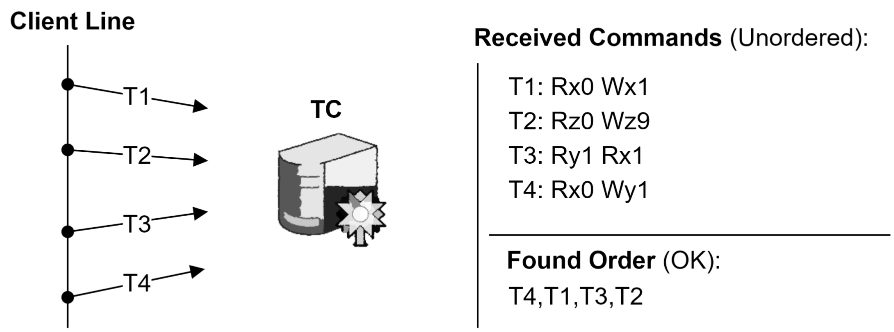
OCC maintains two necessary components:

- **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**: Receives transaction operations, executing it on local state. Then, on **OK**—commit state, on **ABORT**—rollback to the previous state.

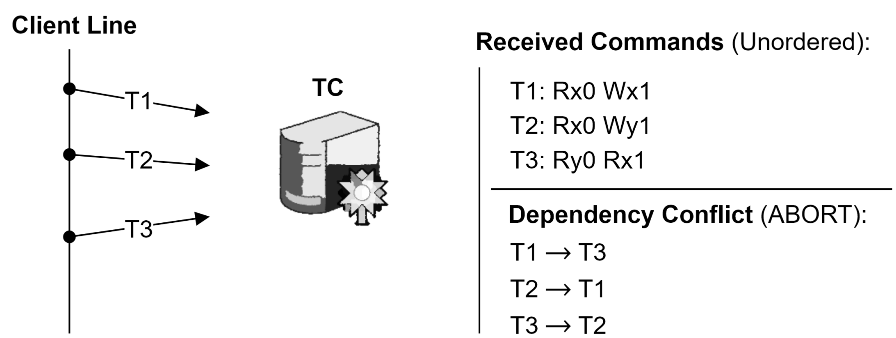
We consider one model, which where multiple clients interact with one TC.

Example 1.1: 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.



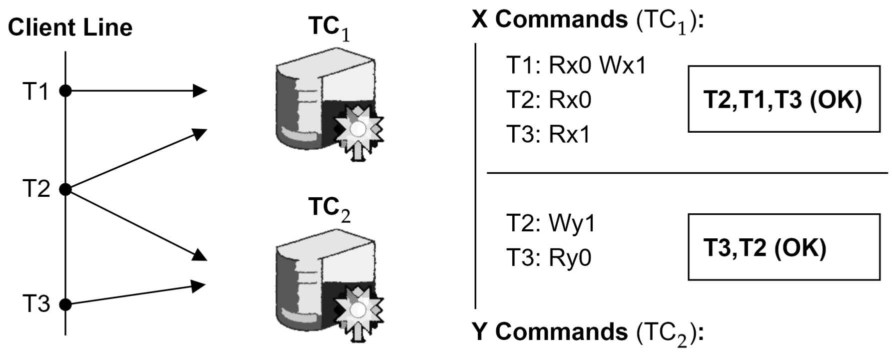
Here, transaction requests, T1, T2, and T3, do not have a serial order. As we build, T1 → T3 (T1 then T3) makes logical sense. Then T2 → T3, giving us the order T2 → T1 → T3; However, it appears T3 must come before T2. This causes a cycle (T2 → T1 → T3 → T2). Hence, the TC must must abort all transactions.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

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 1.2: Distributed OCC

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

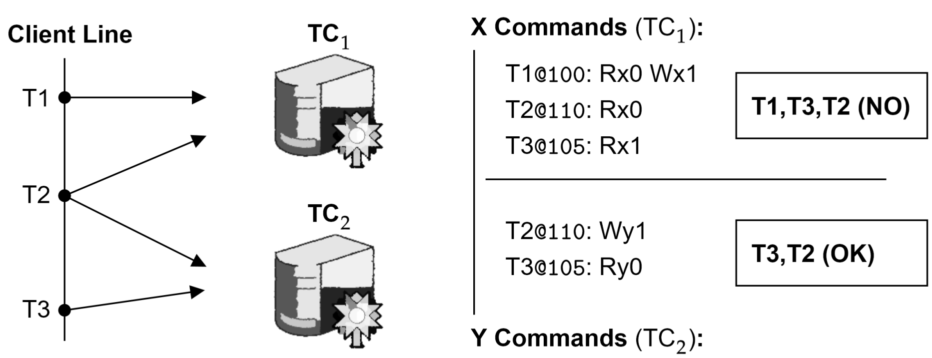


The problem occurs as TC_1 and TC_2 pick **different serial orders** for the transactions. ■

Theorem 1.1: Timestamping Distributed OCC

Timestamping is a method where each transaction is assigned a unique timestamp (ID), which aids order agreement between TCs.

Example 1.3: 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). ■

1.1.2 Two-Phase Commit (2PC)

Bibliography

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