

Distributed Systems

Christian J. Rudder

January 2025

Contents

Contents	1
1 Summary	5
Bibliography	9

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 —

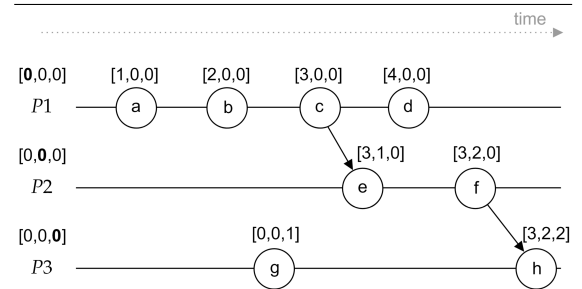
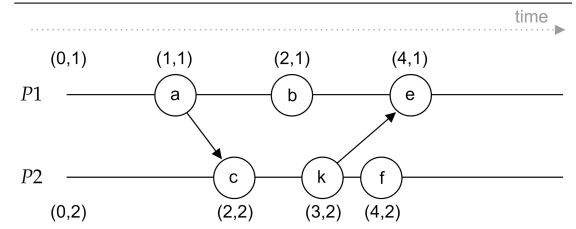
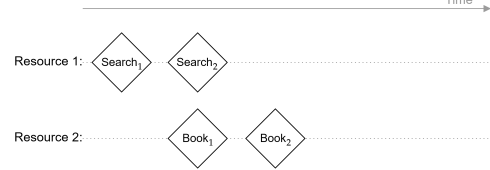
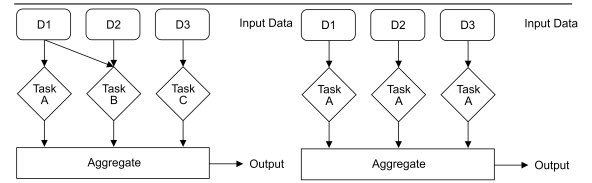
Summary

This page is left intentionally blank.

—**System—Threads:** Hardware threads (CPU cores) each may run one software thread (program) at a time. Switching, between threads is context switching (overhead). E.g., Go manages internal thread pools, offering it to the OS, reducing overhead. **Thread Comm:** Inter-process communication (IPC), threads communicate via shared memory (e.g., channels, pipes, virtual memory). **Conc. & Parallelism:** Concurrency shuffles tasks, parallelism runs tasks simultaneously (threads). **Dist. Conn:** A client connects to a server (client-server model), a TCP conn. (FIFO) secures the line, the RPC (remote procedure call) abstracts dist. communication. **Races & Deadlocks:** Data race, two threads manipulating shared data (reads-onlys are fine). **Mutexes:** Placing locks around shared data, stopping concurrent access. **RPC Fail Models:** At-least-once, client retries until a response is received. Reads are fine, writes cause race conditions. At-most-once, server handles dupes, clients send unique IDs (cached responses). Exactly-once, both at-least-once and at-most-once. **(a)sync:** Asynchronous, non-blocking, no waiting for a response. Synchronous, blocking, waiting for a response. **(un)buffered-channels:** Unbuffered, sender waits for receiver on some thread to receive message. Buffered, sender sends message(s) to a buffer, takes one at a time. **Task, Data, Pipeline Parallelism:** Task, same data, different tasks. Data, same task, different data. Pipeline, task split into dependent stages, independent stages run in parallel. **Time Accr:** Physical clocks drift due to hardware limitations. Atomic clocks, have insignificant drift. NTP (network time protocol), utilizes GPS satellites to sync time, to a ground truth clocks, which propagates time to other systems. —**Logical Time—Lamport Clocks:** Lamport Clocks, (t_p, p) , t_p time of process p , monotonically increases for each event/send (Total Ordering). Receivers q resolve time differences, $t_q = \max((t_p + 1), t_q)$ (send vs. local time). Given two events a & b , timestamps $t(a)$

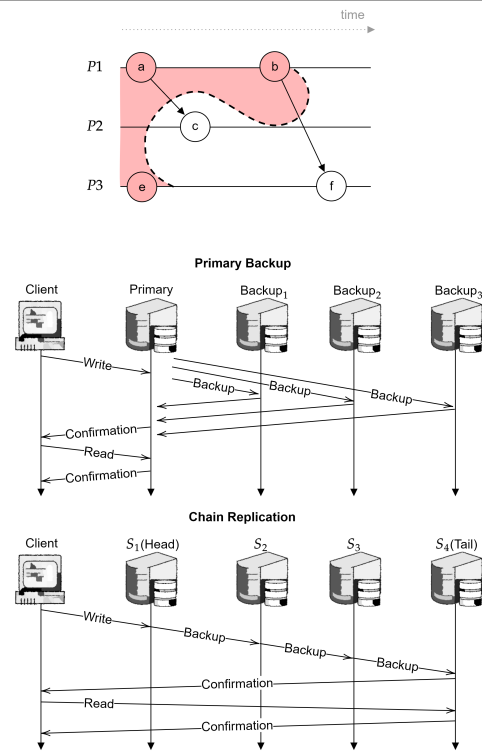
& $t(b)$, with r trace: $a \rightarrow_r b \implies t(a) < t(b)$ (causal ordering); $t(a) \geq t(b) \implies a \not\rightarrow_r b$ (concurrent); $(t(a) = t(b) \wedge a \gg b) \implies a \rightarrow_r b$, s.t., (\gg) rep. process order. **Non-causal:** $(a \ll b) := (a \not\rightarrow_r b) \wedge (b \not\rightarrow_r a)$ (concurrent). **Vector Clocks:** Operate as an array of Lamport clocks, index is process p_i , and value is t_{p_i} (time); However, sends do not increment t_{p_i} . Given timestamps a & b , if all indexes in a are larger than b , $a \rightarrow_r b$. If some in a are larger than b , vice versa, $a \not\rightarrow_r b$ (non-comparable, concurrent, partial ordering).

TaskPar. DataPar. PipePar. La&Ve.Clock:



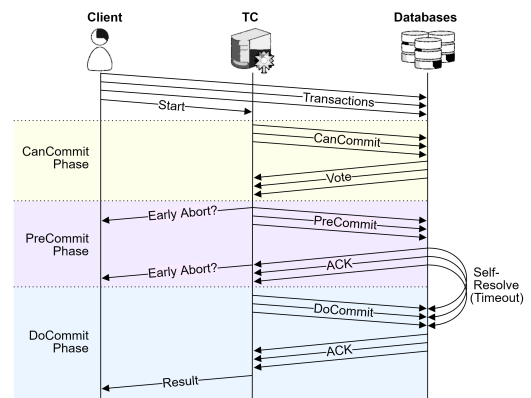
Snapshots: Consistent snap., captures causal dependencies, if $e_1 \rightarrow_r e_2$, and e_2 is in the snap., e_1 must also be present (otherwise it's inconsistent). If in the snap., e_1 sent a message to e_2 , and e_2 is not in the snap., replay the message on snap. recovery. **Chandy-Lamp.Snap:** Alg. for capturing consistent snaps. Reqs: No failures, FIFO channels, Strongly conn. graph, Single initiator. (1) Marker sent to all out-chans., record local state. (2) On marker retrieval, block (empty) the chan. from which it came. Record local state, except for empty chans. Send marker to all out-chans. (3) Completion, When all processes have received and sent a marker, the snapshot is complete. (every processes' incoming channel is empty). —**Replication**— **Def:** Maintaining multiple copies of the same data across distinct nodes (machines), providing fault-tolerance, load-balancing, and availability. **Active vs. Passive Rep:** Active, client sends reqs. to all replicas, must process in FIFO order. Passive, client sends reqs. to one replica (primary), which propagates to others (backups). **State vs. Req. Rep:** State Rep., forwards the entire state to backups. Request Rep., forwards individual reqs. to backups. **Primary Commit:** Client→Primary→Backups→Primary→Client (Commit Point). **Arbitrary Serv. (CFG):** The Configuration Service Provider (CFG) ensures a controlled **failover** (switching to backups) in the event of a primary failure. **Chain Rep:** An ordered chain of s_n replicas, writes propagate from s_1 to s_n , where s_n reports back to the client. Reads speak directly with s_n . For any failover, the next adjacent successor takes over. —**Consensus—State Machine:** Processes a seq. of inputs from a log, saving them in state. **Rep. State Machine:** Replicated logs across multiple machines, the processing of which is deterministic, generating the same state. **Consensus Model:** facilitates agreement of replicated logs between replicas. **Raft:** A consensus algorithm, with a central leader elected by the cluster in monotonically increasing terms. Liveness is measured by periodic heartbeats, if the leader isn't reachable, followers, run for election. Split votes are dealt with the current

candidates by timing out again (new timeout). **Log Matching Property:** Same index and term implies, same command and previous indexes are identical. **Log Correction:** The leader will find the index of the first mismatch, then overwrite the followers entries with its own. **Leader Completeness:** All proceeding leaders must have all committed entries of the previous leader. Leaders may only commit entries from their term. Candidates are rejected if they have a lower term, shorter log. **Timings:** $heartbeatReceipt \ll electionTimeout \ll failRate$. **Cluster Reconfiguration:** A joint consensus reconfiguration follows two phases: (1) propagate mix config of old and new, (2) propagate new config, which includes new servers or deletions. New servers enter with empty logs. **Log Compaction/Snapshotting:** Servers independently take snapshots, which truncates all committed logs, storing the last committed log index, term, and state (key-value pairs). Clients are always redirected to the leader. **2ReplaysSnap. Primary & Chain Rep.**



—**Failure Model Hierarchy**—Crash-Stop: Process halts, cannot resume (undetectable) \subset Omission: Process fails to properly communicate \subset Crash-Recovery: Process halts, but can recover \subset Byzantine: Process exhibits arbitrary or malicious behavior. —**Consistency Models**—**Def:** A distributed system’s method on validating operation orderings on shared data. **Global Total Order:** Order of observable events agreed upon by all clients. **Strong Consistency (Str):** The client observes nodes agree on order execution (all node reads appear to be identical). **Weak Consistency (Wck):** The client temporarily observes that nodes disagree on shared data values. **Linearizability (Str):** Global Total Order with respect to real-time ordering (operation time intervals unmovable, but execution choice within them are). E.g., Raft leader-commit (happy-path) and Chain Replication. **Sequential (Str):** There is some Global Total Order found when shifting operation time intervals. **Causal Consistency (Wck):** Same as Sequential; However, clients may observe their own view on a Global Total Order. **Eventual Consistency (Wck):** Given no new writes, replicas will eventually agree on the same value after some time. **Causal Implications:** Linearizability \Rightarrow Sequential \Rightarrow Causal \Rightarrow (FIFO/RYW). **Release Consistency (Wck):** Push updates to all nodes after releasing the lock. **Lazy-release Consistency (Wck):** Push updates to the next node who acquires the same lock. —**Transactions**—**ACID:** Atomicity, no partial effects, all or nothing. Consistency, data departing the database leaves unmutated. Isolation, no interleaving of transactions. Durability, transactions once committed, are permanent even after system failure. **Serializability (Str):** Ensures the outcome of concurrent transactions is the same as if they executed in some serial (sequential) order. Differs from linearizability, which deals with real-time ordering of single tasks, as opposed to whole jobs. **Strict-Serializability (Str):** Serializability with real-time ordering; In particular, Serializability \Leftarrow Strict-Serializability \Rightarrow Lin-

earizability. **Optimistic Concurrency Control (OCC):** Assumes conflicts are rare, proceeds without locking. (1) Prepare the transaction on all nodes, (2) Tell the coordinator to execute, validate the outcome, (3) Commit if Serializable, abort otherwise (isolation). **Timestamping OCC:** Helps with distributed OCC agreement on separated data, but aborts unnecessarily. **Two-Phase Commit (2PC):** Ensures atomicity, (1) Prepare Phase: After client’s request is received on all nodes, the Transaction Coordinator (TC) sends a prepare message to all nodes, awaiting YES votes. (2) Commit Phase: If all nodes reply YES, the TC requests for all nodes to commit. If any fail to reply, the TC is blocked, sending the commit request to that node indefinitely (can’t abort after this point). **3PC:** 2PC with an additional phase before the commit phase, ensuring people can commit (Pre-Commit Phase). If any nos or failure to respond in PreCommit, abort. If not partitioned, if the TC fails in the PreCommit phase, servers may attempt to reconcile with themselves about how to continue. **Pessimistic Concurrency Control (PCC):** Assumes conflicts are likely and prevents them by acquiring locks before any data access. It then preforms actions directly on shared data, ensuring serializability and isolation. The locks are released after the transaction is completed. 3PC.



Bibliography

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