

CPSC 416 – Distributed Systems

Lecture 06 — Single-Node Failure and Recovery

UBC Vancouver

2026-01-22

Failure-first question

A node crashes mid-operation. What state is it in? What would “recovery” mean?

What can go wrong?

Failure models — a taxonomy of what we assume can fail:

- **Crash failure:** Process stops, loses volatile state
- **Omission failure:** Message sent but never received
- **Timing failure:** Response outside expected time bounds
- **Byzantine failure:** Arbitrary/malicious behavior

Today: **crash failures** — the simplest and most common model.

Crash failure: The mental model

A crash-stop process:

- Executes correctly until it crashes
- Loses all volatile state (memory) on crash
- May retain durable state (disk) after crash
- Never performs incorrect actions (no “half-writes”)

Key assumption: Processes are either correct or dead — no middle ground.

Is this realistic? *Mostly* yes. Hardware typically fails cleanly; bugs are the bigger problem.

Safety and liveness

Two properties we care about in every distributed system:

Safety: “Nothing bad ever happens.”

- Invariants are never violated
- Examples: no data loss, no inconsistent state, no split brain

Liveness: “Something good eventually happens.”

- The system makes progress
- Examples: requests eventually complete, recovery eventually finishes

The tension: Safety is easier if you do nothing. Liveness requires action. Recovery must provide *both*: restore safety without blocking forever.

Studio 1: Where can things go wrong?

Scenario: ATM withdrawal transaction

Steps:

- 1 Check account balance
- 2 Debit account (subtract amount)
- 3 Dispense cash
- 4 Update transaction records

Your task (5 min): Where can a crash leave the system in a bad state?

Think about: What invariants should hold? What state is “inconsistent”?

Studio 1: Failures you might have missed

Common answers:

- Crash after debit, before dispensing cash
- Crash after dispensing, before updating records

Subtler failures:

- **Torn write:** Crash mid-write, partial data on disk
- **Power failure during log write itself**
- **Write in disk cache vs. on platter** — is it durable?
- **Memory corruption before flush**
- **Disk controller lies about flush completion**

Takeaway: Multi-step operations are *not* atomic by default.

The atomicity problem

Definition: An operation is **atomic** if it appears to execute either completely or not at all.

The problem:

- Real operations take multiple steps
- Crashes can happen between any two steps
- After a crash, the system is in an *intermediate* state
- Intermediate states may violate invariants

Question: How do we make multi-step operations atomic?

The log: Write intention before action

Key insight: Before doing something, *write down what you intend to do*.

The pattern:

- 1 Write intention to durable log
- 2 Perform the actual operation
- 3 Mark operation as complete in log

On crash:

- Read the log
- For incomplete operations: redo or undo based on log contents
- System returns to consistent state

Why does logging work?

The log is the source of truth.

- Log writes are *sequential* (append-only)
- Sequential writes can be made atomic (single sector write)
- The log survives crashes (durable storage)
- Recovery reads log to determine what to do

Fundamental trade-off: Extra writes during normal operation → correct recovery after crash.

Write-ahead logging (WAL)

Rule: Write the log record *before* modifying the actual data.

1. Write "BEGIN T1" to log
2. Write "T1: set $X = 5$ (old: 3)" to log
3. Modify X in database
4. Write "COMMIT T1" to log

On recovery:

- See COMMIT? → Redo any changes not yet applied
- No COMMIT? → Undo any partial changes

What is “consistent state”?

Consistency = the system satisfies its invariants.

Examples of invariants:

- “Account balance ≥ 0 ” (banking)
- “Every debit has a matching credit” (double-entry accounting)
- “File size matches inode metadata” (filesystem)
- “Foreign keys reference existing records” (database)

You define what “consistent” means. The system just enforces it.

Recovery as state transition

Inconsistent State $\xrightarrow{\text{recovery}}$ Consistent State

Recovery is not “magic” — it’s a well-defined state transition:

- 1 Read the log
- 2 Determine which operations were in-flight at crash
- 3 For each: complete it (redo) or roll it back (undo)
- 4 Result: all invariants satisfied

The log gives us enough information to make this transition.

Logging variants: Same principle, different trade-offs

The log pattern has many implementations:

- **Redo logging:** Log new values, replay committed ops
- **Undo logging:** Log old values, rollback uncommitted ops
- **Undo/redo (WAL):** Log both — most flexible, used by most databases
- **ARIES:** Industrial-strength (checkpoints, fuzzy snapshots, physiological logging)

The principle is always the same:

Log $\rightarrow$ Recovery $\rightarrow$ Consistent State

Further reading:

- DDIA Chapter 7 (Transactions)
- Mohan et al., “ARIES” (1992) — the classic paper

The pattern: failure $\rightarrow$ log $\rightarrow$ recovery $\rightarrow$ consistent state

The First Turtle

- ① **Failure model:** Crash (stop, lose volatile state)
- ② **Mechanism:** Log (durable record of intentions/actions)
- ③ **Recovery:** Read log, redo/undo to restore invariants
- ④ **Result:** Consistent state

This pattern works for a **single node**.

But what happens when you have **multiple nodes** that need to agree?

Studio 2: The distributed lock problem

Scenario: You're building a distributed lock service.

- Primary server holds lock state
- Backup server maintains a replica
- Client A requests the lock

Your task (5 min): Walk through the steps. Where can this fail?

Specifically: What happens if the primary grants the lock to A, then crashes before the backup knows?

Studio 2: The problem

The failure case:

- 1 Client A requests lock from Primary
- 2 Primary grants lock, writes to its log
- 3 Primary crashes before telling Backup
- 4 Backup becomes new Primary
- 5 Client B requests lock from (new) Primary
- 6 New Primary grants lock (doesn't know about A!)

Result: Both A and B think they have the lock. *Split brain.*

Logs on each node aren't enough. **The nodes need to agree.**

The question for next time

How do multiple nodes agree on a decision before any of them commits to it?

Spoiler: We need **another turtle**.

The pattern: failure $\rightarrow$ log $\rightarrow$ recovery $\rightarrow$ consistent state

We'll apply this same pattern at the *coordination* level.

Next lecture: Two-Phase Commit (2PC) — coordination as layered recovery.

Logistics

- Readings + quiz (Canvas → Readings page)
- Self-assessment opens at 13:00, due next class 11:30 (Pacific)
- Upcoming DP/capstone milestones (Canvas → Assignments)

Key concepts from today:

- Failure models (crash failure as starting point)
- The log pattern for atomicity and recovery
- Consistency = satisfying system invariants
- Single-node recovery is solved; multi-node is harder