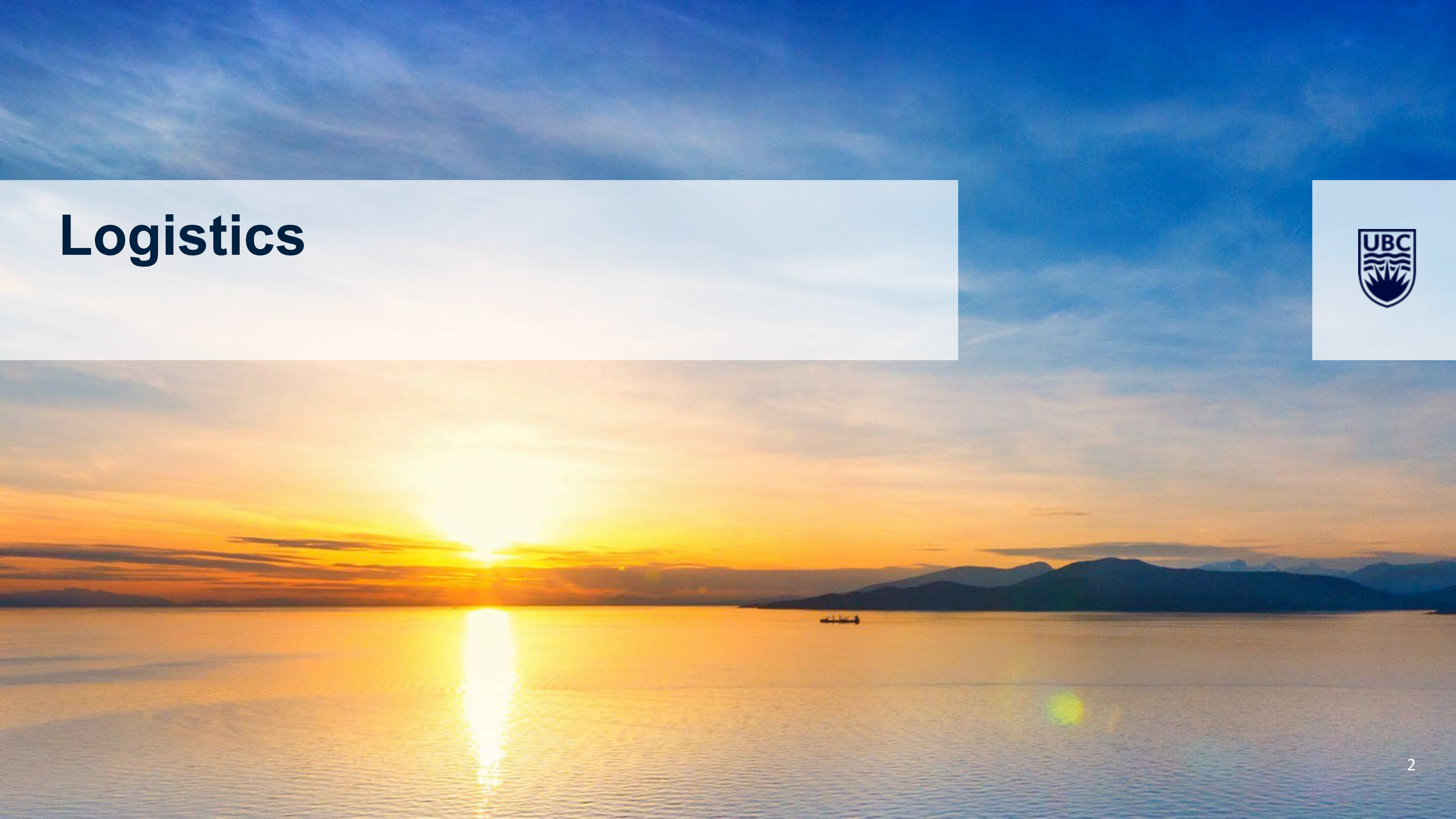
CPSC 416 Distributed Systems

Winter 2022 Term 2 (February 7, 2023)

Tony Mason (fsgeek@cs.ubc.ca), Lecturer



Logistics

Deadlines

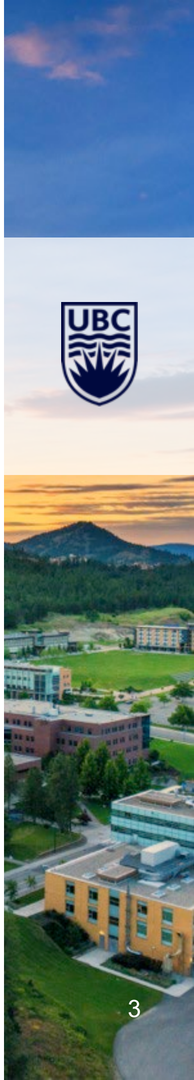
Project 2 Report: Pending Review

Project 3 Released. Initially Due: **February 13, 2023**. Next Monday

Project 4 Released. Initially Due: March 13, 2023

Project 5 Released Due: April 13, 2023

All project work is due April 13, 2023. Late projects have a 75% score cap.



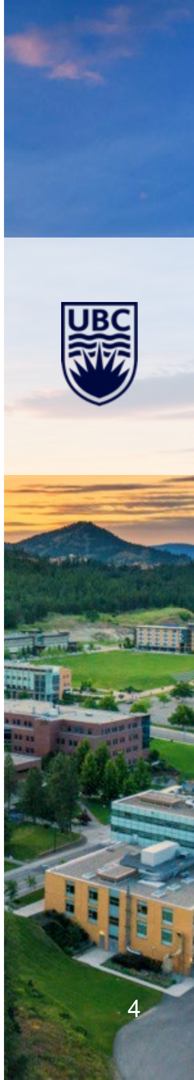
Deadlines

Alternate Path 1 & 2: Review in progress

- Piazza private threads created
  - Review
  - Respond to outstanding questions
- **Proceed according to your plan.**

Instructor Office Hours:

- Zoom Office Hours (Tuesday) @ 13:00-14:00
- Discord (Casual) Office Hours (Thursday) @ 14:00-15:00

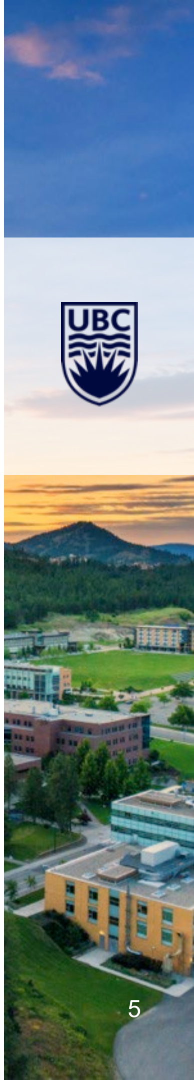


Readings

Required:

Recommended:

- Corfu: a distributed shared log
- Serverless Network File Systems
- Distributed Logging for Transaction Processing
- EngraveChain: A Blockchain-Based Tamper-Proof Distributed Log System
- Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems using Event Chaining Approach

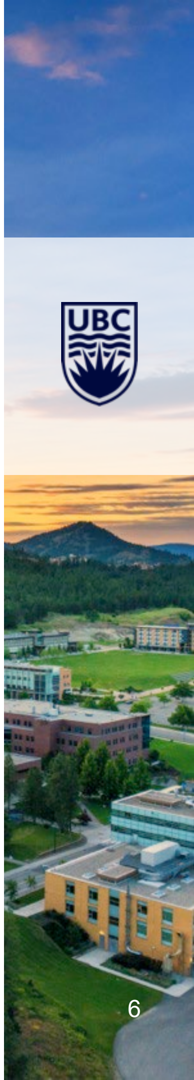


Questions?

Questions about the class?

Questions about the previous lecture?

Funny stories to share?



Today’s Failure

Google Cloud Platform

Event start: June 2, 2019 @ 11:45 PT

Event ends: June 2, 2019 @ 15:40 PT

TL;DR Version

- Network connectivity issues
- Degrade/Disrupt
  - Compute Engine (up to 33% packet loss)
  - App Engine (23.2% decrease in requests per second) – many timeouts
  - Cloud Endpoints – 50% of service configuration push workflows failed
  - Cloud Interconnect – packet loss up to 100%
  - Cloud VPN – multiple gateways unreachable
  - Cloud Console – slow/failed page loads
  - Etc. (6 more impacted services including G Suite)

Cause

Two “normally benign” misconfigurations

One software bug

(1) Network control plane jobs configured to be stopped for maintenance event
(2) Cluster management software instances included in “relatively rare mainentance event”
(3) Maintenance event software initiator bug: allows descheduling multiple independent software clusters *at the same time*. Clusters were geographically distributed.

Mitigations

Network data plane continues to run without control plane presence.

BGP routing stopped: connectivity interrupted (“can’t get there from here”)

Google engineering knew of the issue within *two minutes*.

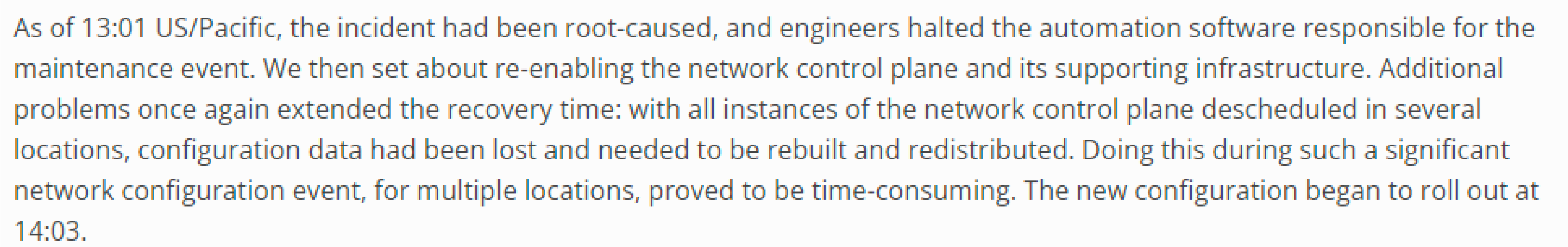
Tools don’t work: “Debugging the problem was significantly hampered by failure of tools competing over use of the now-congested network.”

Mitigations

When your communications fail…

“The defense in depth philosophy means we have robust backup plans for handling failure of such tools, but use of these backup plans (including engineers travelling to secure facilities designed to withstand the most catastrophic failures, and a reduction in priority of less critical network traffic classes to reduce congestion) added to the time spent debugging. Furthermore, the scope and scale of the outage, and collateral damage to tooling as a result of network congestion, made it initially difficult to precisely identify impact and communicate accurately with customers.”

Recovery



As of 13:01 US/Pacific, the incident had been root-caused, and engineers halted the automation software responsible for the maintenance event. We then set about re-enabling the network control plane and its supporting infrastructure. Additional problems once again extended the recovery time: with all instances of the network control plane descheduled in several locations, configuration data had been lost and needed to be rebuilt and redistributed. Doing this during such a significant network configuration event, for multiple locations, proved to be time-consuming. The new configuration began to roll out at 14:03.

Cascading failures – again. This is a common (and recurring theme) in distributed systems failures.

Lesson Goals

Distributed Logging

Review: transactions

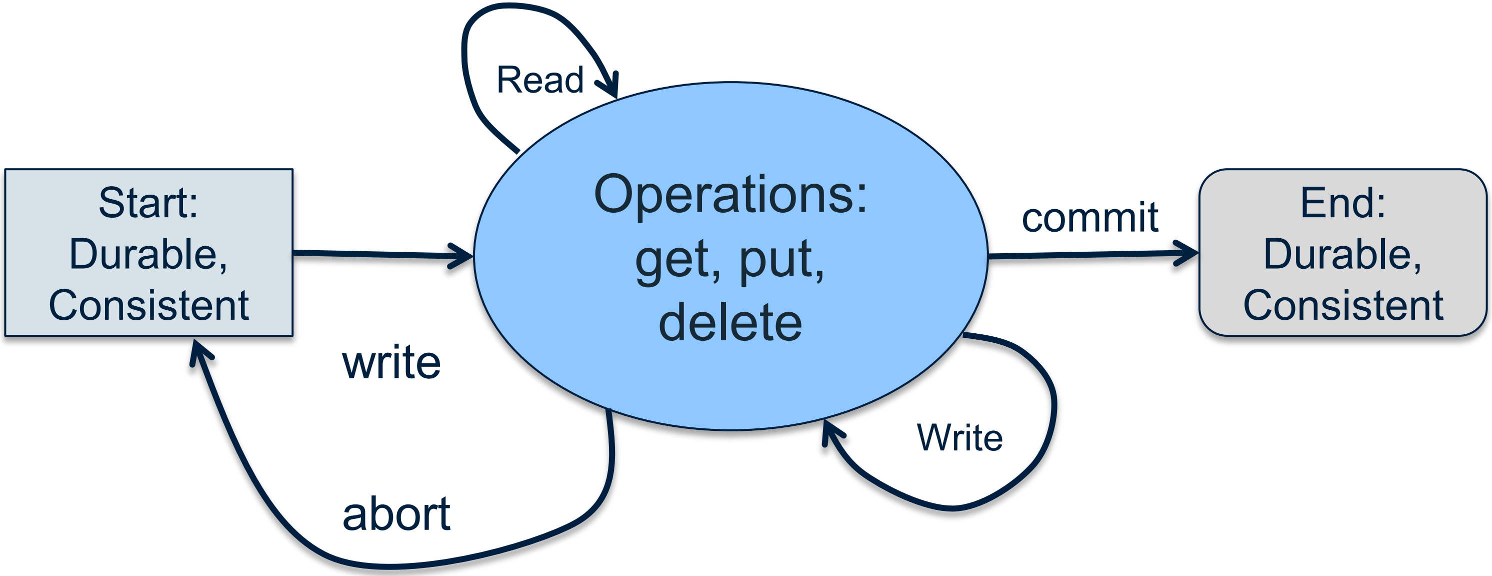
Types of Logs

How Logs implement Transactions

Centralized Logs

Distributed Logs

Review: Transaction



Start: Durable, Consistent

Read

Operations: get, put, delete

commit

End: Durable, Consistent

write

abort

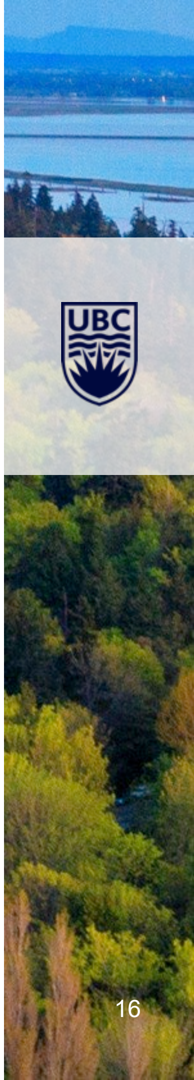
Write

Transactions

Transactions provide:

- Atomicity of *multiple* distinct operations
- Consistent state (beginning/end)
- Isolation (intermediate states *are not visible*)
- Durable (outcome is preserved)

Note: in the “real world” we often explore different ways of realizing these



Logs

Record persistent information

- Enables recovery (and abort)

Logging types

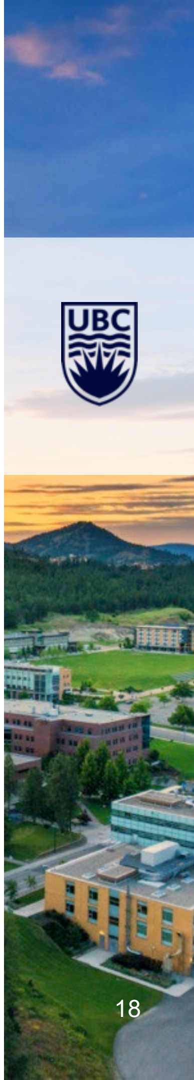
- Redo – roll forward (data mutations blocked until commit)
- Undo – roll back (log written before data mutations written)
- Write-ahead – combines undo/redo loggin

Why Logs?

Logs provide a *serialized* version of events

Tracking distributed state (vector clock) permits disjoint logs to be combined.

- Recall: serialization doesn’t have to be **the** order it is just **one possible order**
- Ordering is “flexible” iff outcome is the same.



Undo/Redo Logging

1979 – IBM Technical Report

- Transaction Logs
- 2 Phase Commit
- Undo/Redo Recovery

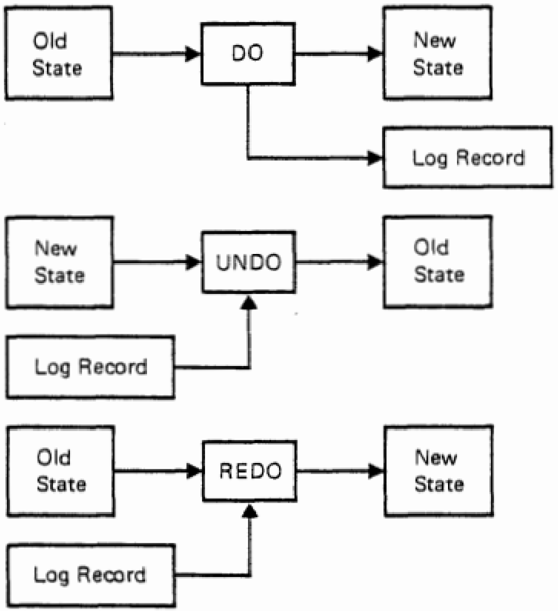


Figure 7. Three aspects of an action: Action DO generates a new state and a log record. Action UNDO generates an old state from a new state and a log record. Action REDO generates a new state from an old state and a log record.

Write-Ahead Logging

Goal: use undo and redo logging to ease write-back rules

Steps:

1. Start transaction
2. Write old values to log
3. Modify the data
4. Write new values to log
5. Commit transaction
6. Write data to disk
7. Truncate record (no longer needed)

Recovery

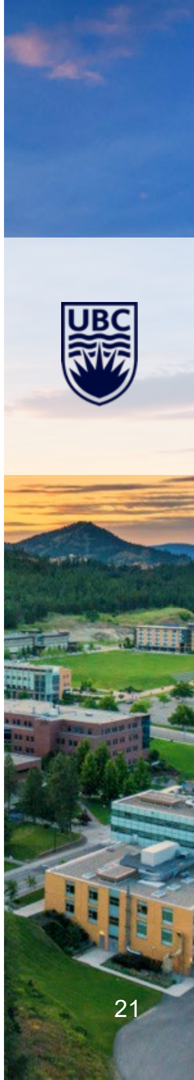
After restart

- Find head of log: most recent transaction
- Find tail of log: oldest *active* transaction
- For transactions with a commit but not truncated: write new values
- For transactions without a commit: write old values

Truncation occurs when:

- All data changes in a given transaction have been recorded.
- Can be *lazy*

Note: recovery must do *undo* before *redo*.



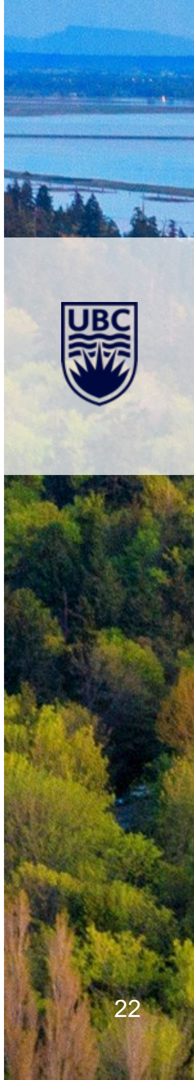
Distributed Logs

At least one log must define transaction outcome (commit/abort)

- Note: one log becomes a “single point of failure”
- Could use chain replication for redundancy

Coordinator handles distributed transactions

- Transaction identifier (TID) created – links transactions across nodes
- Knows all transaction participants
- Provides atomic commit
- Logs outcomes – used for recovery



Log: 2PC

Coordinator

- Begin Transaction
- Commit Transaction
- Abort Transaction

Messages between Nodes and Coordinators

- Node to Coordinator: request to commit
- Coordinator to all Nodes: prepare to commit
- Node to Coordinator: prepared/aborted
- Coordinator to Nodes: committed/aborted

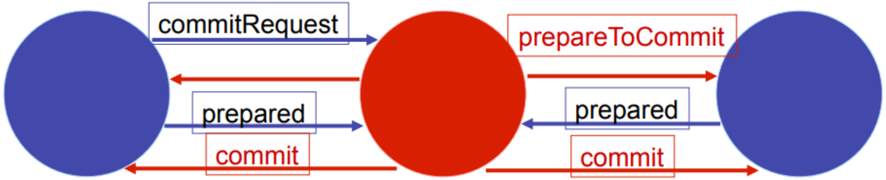
Two-Phase Commit Walk-through

Phase 1: Voting

- Node sends request to commit
- Coordinator sends prepare to commit
- Nodes prepare the request (write to their own log)

Phase 2: Transaction completion

- Coordinator: waits for all Nodes to answer
- Coordinator: writes commit to log
- Coordinator: tells Nodes the outcome
- Node: commits (writes to log)



commitRequest

prepareToCommit

prepared

prepared

commit

commit

[u,tid,xxx]
[p,tid]
[c,tid]

[b,tid]
[c,tid]

[u,tid,xxx]
[p,tid]
[c,tid]

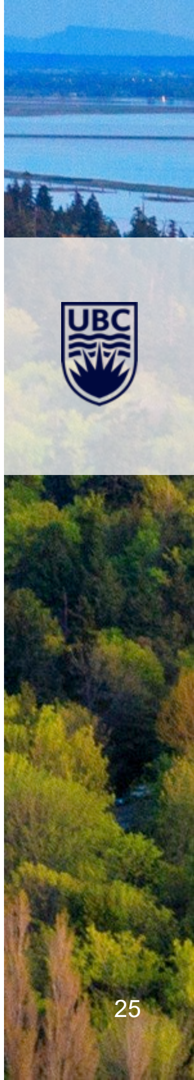
Node Recovery

Coordinator

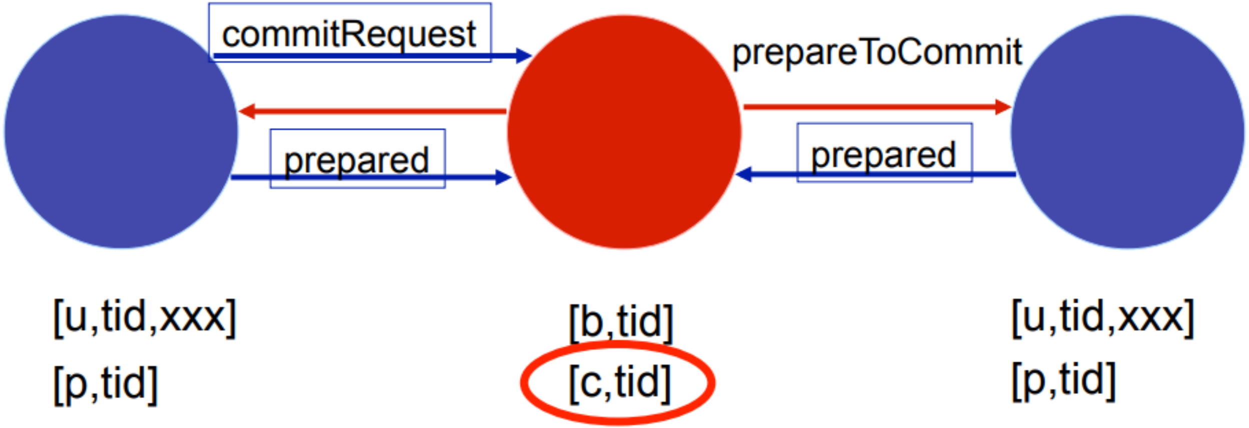
- Uses *timeout* mechanism
- Aborts pending transactions dependent upon the failed Node

Node

- Recovers log
  - Log records without a prepare to commit: abort
  - Log records *with* a prepare to commit, no commit/abort:
    - Query Coordinator for outcome
    - What if the Coordinator is unavailable?



2PC Commit (In flight)



commitRequest

prepareToCommit

prepared

prepared

[u,tid,xxx]

[p,tid]

[b,tid]

[c,tid]

[u,tid,xxx]

[p,tid]

This transaction has committed, but workers don't know yet

Coordinator Failure

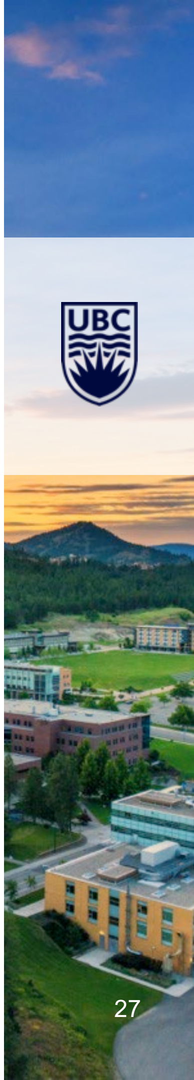
Node sending commit request:

- Timeout is a failure
- Abort transaction in local log

Node sends prepared response

- Timeout if no committed/aborted received
- Transaction outcome is *indeterminate*
    - Must ask Coordinator
    - What if Coordinator is permanently lost?

As I noted: single Coordinator becomes a single point of failure.



Replicated Log

Coordinator permanent failure is catastrophic

- Could replicate the coordinator log

Two possible approaches:

- Use quorum replication (Daniels, Spector, Thompson)
  - Multiple *log servers*
  - Distributed replication: weaker than *chain replication*
  - Coordinator is a single client
- Global shared log
  - Replication
  - Parallelism
  - Garbage collection via *sparseness* (“trim”)

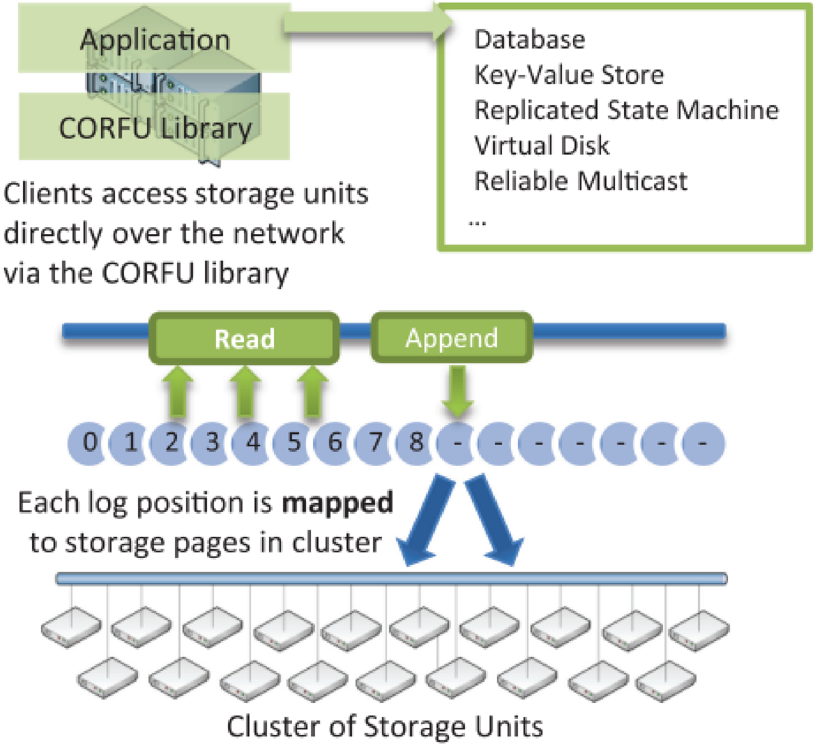
Global Log (Corfu)

Distributed Log

- Abstraction is a single log
- Append-only
- Sparse (“Trim”) – Compaction
- Write-Once (atomic write)
- Space Reservation (collison avoidance) – Optional
- Block level replication (chain)

Infinitely growing log

- Maps virtual to physical locations
- Deletes unneeded storage



Clients access storage units directly over the network via the CORFU library

Each log position is **mapped** to storage pages in cluster

Tamper-Proof Log

Strong audit requirements

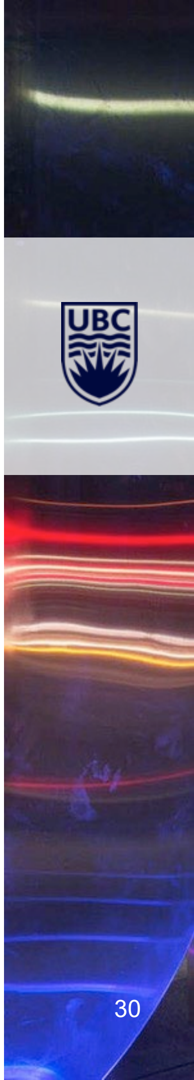
- Payment Card Industry (PCI) standards
- General Data Protection Regulation (GDPR)
- Health Care (PIPA & HIPPA)

Strong requirements

- Distributed
- **Byzantine** fault tolerant

EngraveChain

- Distributed, chained log (“block chain”) using Practical Byzantine Fault Tolerance



EngraveChain

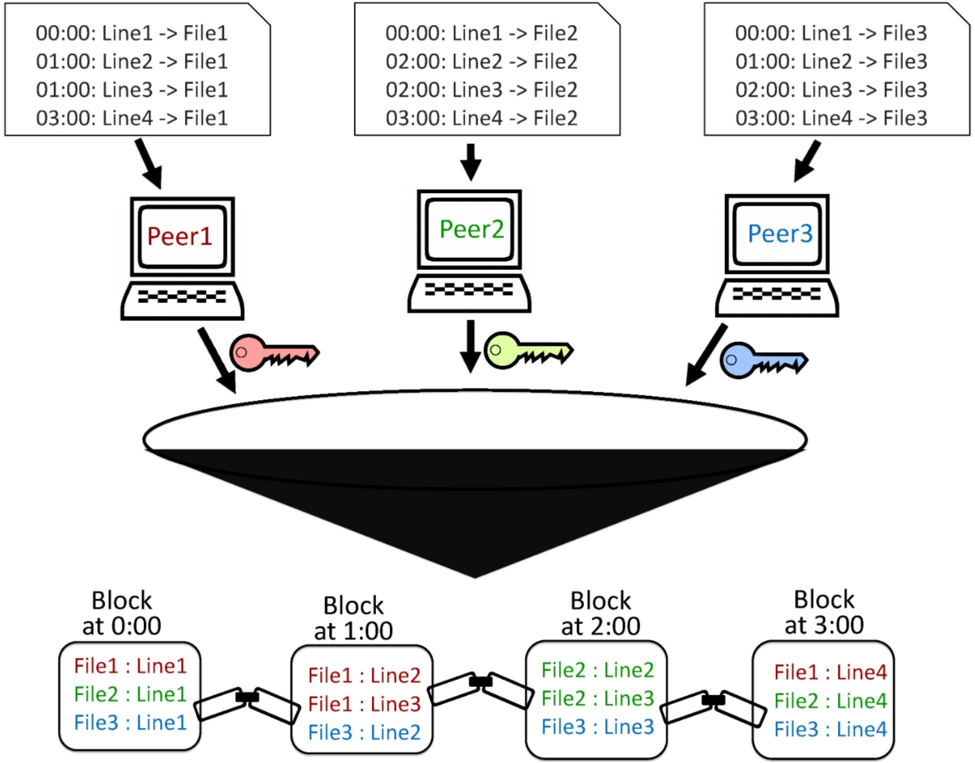
Write logs to blockchain

- Encrypt log with peer key
- Insert encrypted file into blockchain

Recover

- Linearized list of blocks
- Ask peer to decrypt

Can use hash of file contents for large files.



Lesson Summary

Distributed Logging

Mechanism for building distributed transaction

Undo/redo logs

Distributed Logs

Global Logs

Secure Logs

Questions?

UBC THE UNIVERSITY OF BRITISH COLUMBIA