Exactly Once RecordAppend Semantics: GFS

DISTRIBUTED SYSTEMS COURSE PROJECT

Umang Patel
Vyakhya Gupta

OVERVIEW

- Objectives
- Google File System
- Exactly Once vs At Least Once Semantics
- Implemented Features
- Solution Design
- Benchmarks
- Future Work

PROJECT OBJECTIVES

Core System Implementation

- **Functional**: Build a distributed file system following the GFS Master-Chunkserver architecture.

- **Operations**: Support standard file operations: Create, Read, Write, RecordAppend and Delete across multiple nodes.

Append Semantics

- **Functional**: Implement Atomic Append to ensure multiple clients can append concurrently without race conditions.

- **Non-Functional**: Guarantee Idempotency; the system must identify and discard duplicate requests caused by network retries.

Other Requirements

- **Fault Tolerance:** Failures of servers should be handled seamlessly

- **Availability**: Should not be compromised too much

- **Scalability** of architecture for multiple clients

GOOGLE FILE SYSTEM (GFS) OVERVIEW

Architecture:

- Master: Manages metadata (namespace, access control, chunk mapping).
- Chunkservers: Store actual fixed size data blocks (Chunks) on local disk.
- Client: Interacts with Master for metadata and Chunkservers for data.

Characteristics:

- Optimized for large files and high throughput.
- Assumes commodity hardware (frequent component failures).
- Data Flow: separated from Control Flow (Data flows linearly between chunkservers).
- RecordAppend Operation supported: Atomic and At Least Once

EXACTLY ONCE VS. AT LEAST ONCE SEMANTICS

(Standard GFS)
At Least Once RecordAppend

Behavior: If an append times out, the client retries.

Result: If the original write actually succeeded in some chunkserver but the ack was lost, the data is written twice.

Use Case: acceptable for log crawling or situations where duplicates can be filtered later.

(Our Project)
Exactly Once RecordAppend

Behavior: The operation is performed exactly one time, regardless of network failures or retries.

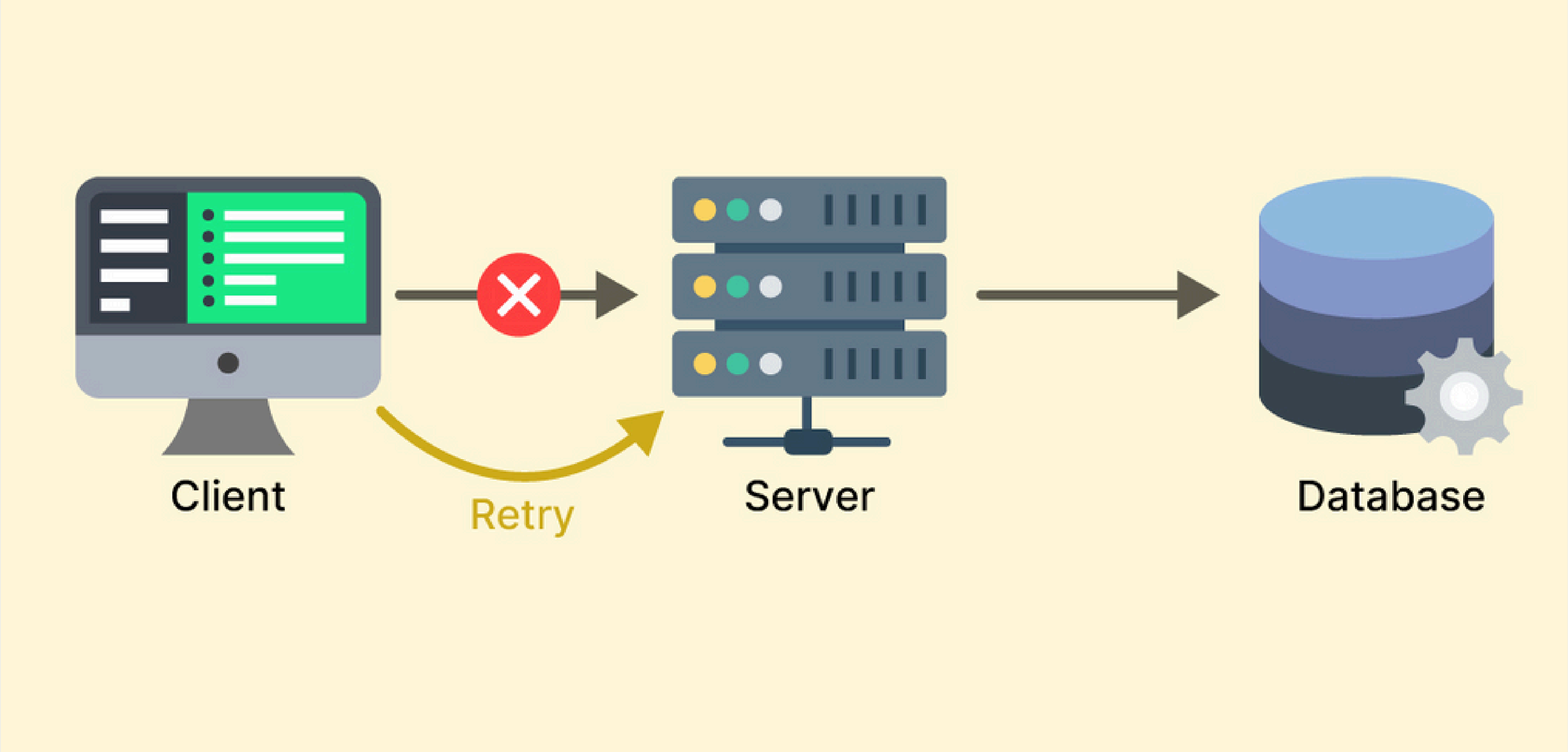
Result: No duplicates, consistent file size across replicas.

Use Case: Essential for financial transactions, counters, or strict ordering.

SOLUTION: IDEMPOTENCY

To achieve Exactly-Once semantics, we modified the standard GFS append flow:

- **UUID** Assignment: The Client generates a generic Unique Identifier (UUID) for every append request.
- **Idempotency Check**: Chunkservers maintain a log of recently processed UUIDs with TTLs.
- If a Request UUID exists in the log → Return cached success (do not rewrite).



SOLUTION: 2 PHASE COMMIT

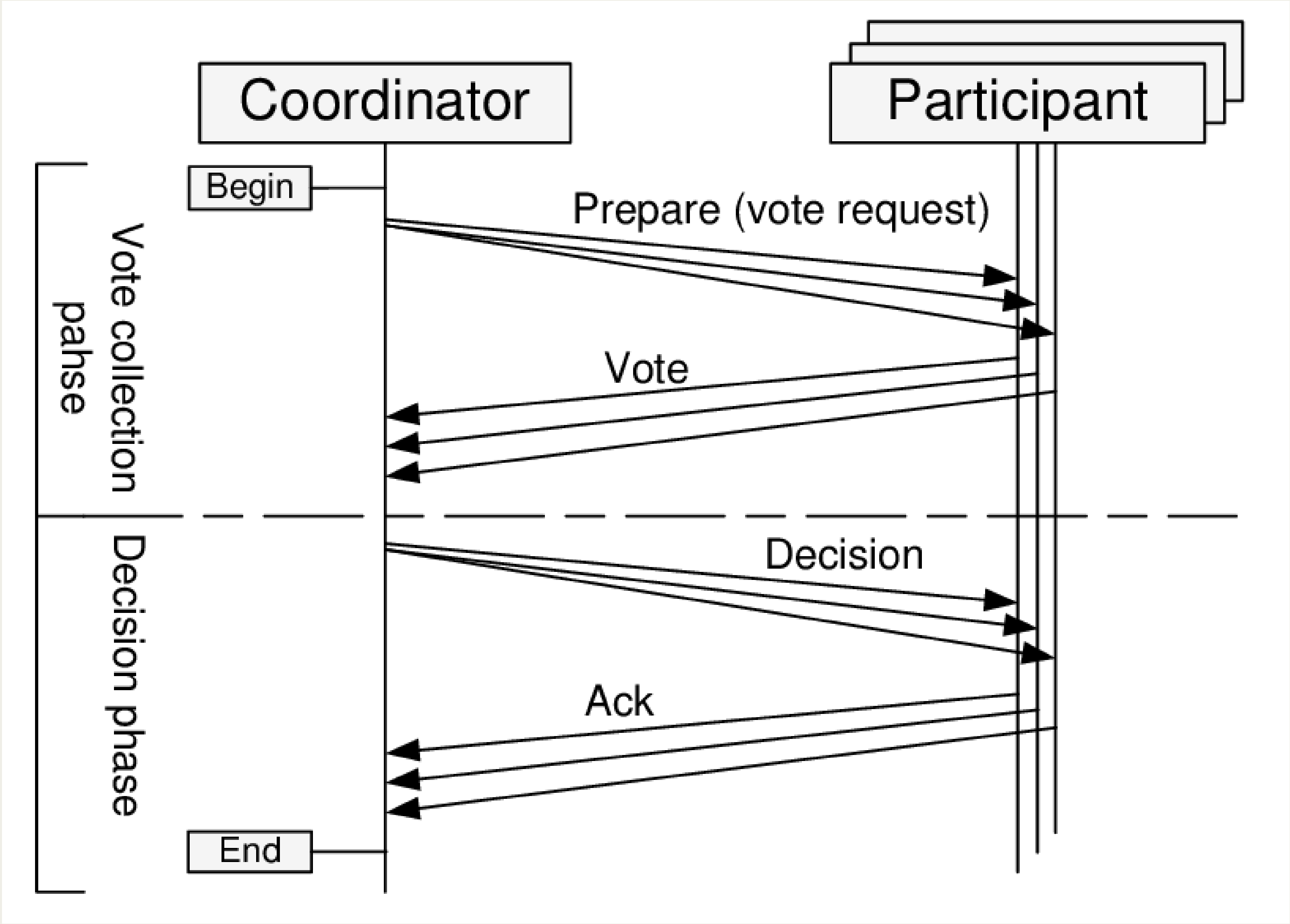
The Primary Chunkserver acts as the Coordinator.

Phase 1: Prepare

- Primary sends data + UUID to all Secondary replicas.
- Secondaries validate checks (disk space, UUID uniqueness) and write .
- Secondaries vote: COMMIT or ABORT.

Phase 2: Commit

- If all vote COMMIT → Primary tells all to apply changes to memory/disk.
- If any vote ABORT (or timeout) → Primary initiates Rollback (discard data).



TECHOLOGY USED

Golang

Provides efficient, native concurrency primitives and high performance, making it ideal for handling parallel network operations and distributed state management at scale.

gRPC

Enables low-latency, strictly typed communication between Master and Chunkservers using high-performance Protocol Buffers for serialization.

IMPLEMENTED FEATURES

Operations
Read, Write, Append, List, Rename File, Create File, Delete File

Garbage Collection
Lazy deletion of orphans, deleted via background process

Stale Replicas
Filtered via Chunk Versions

Failure Handling
Heartbeat monitoring

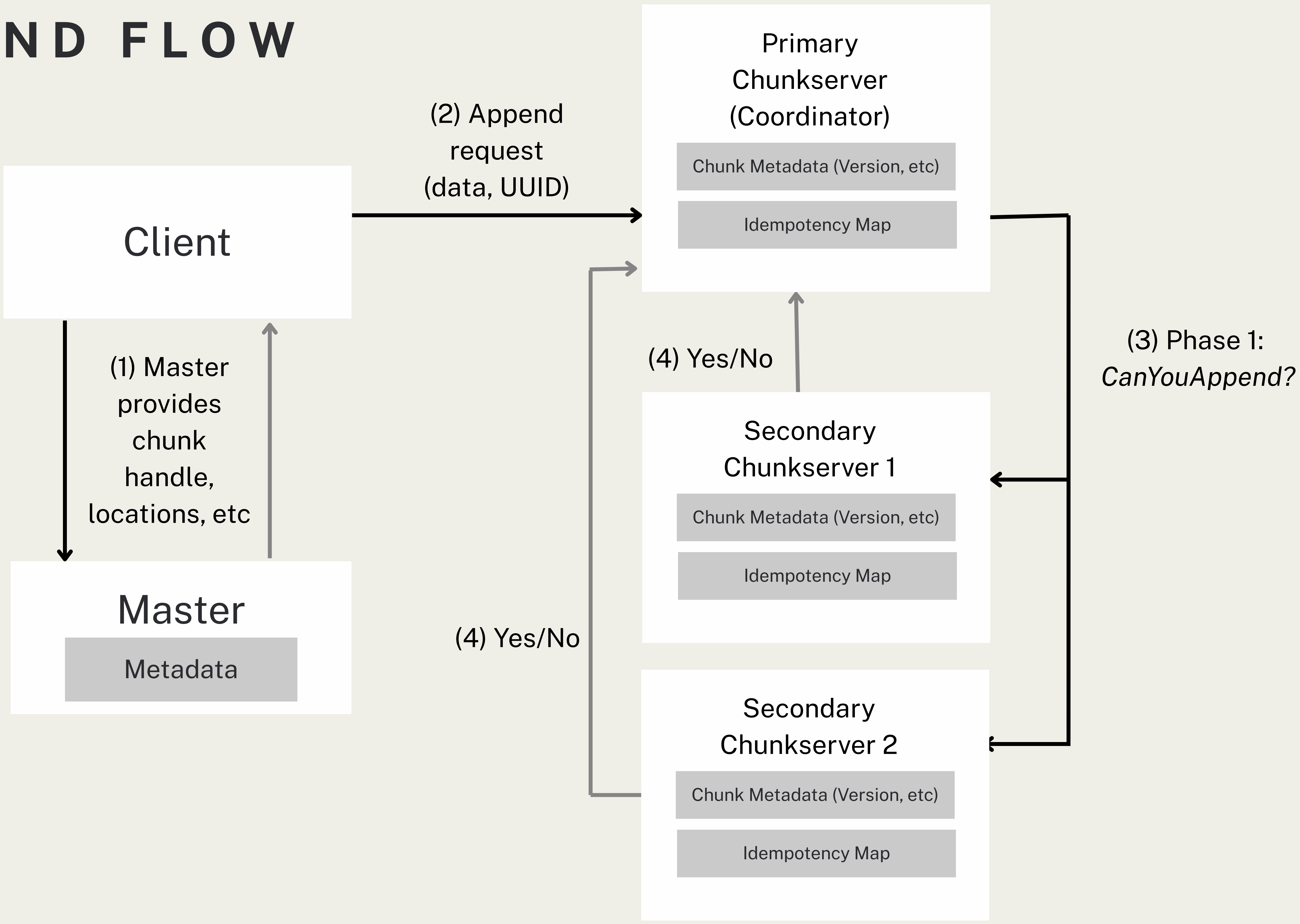
Under/Over Replication
Auto-balancing replica counts

Operation Log
Persists log for recovery

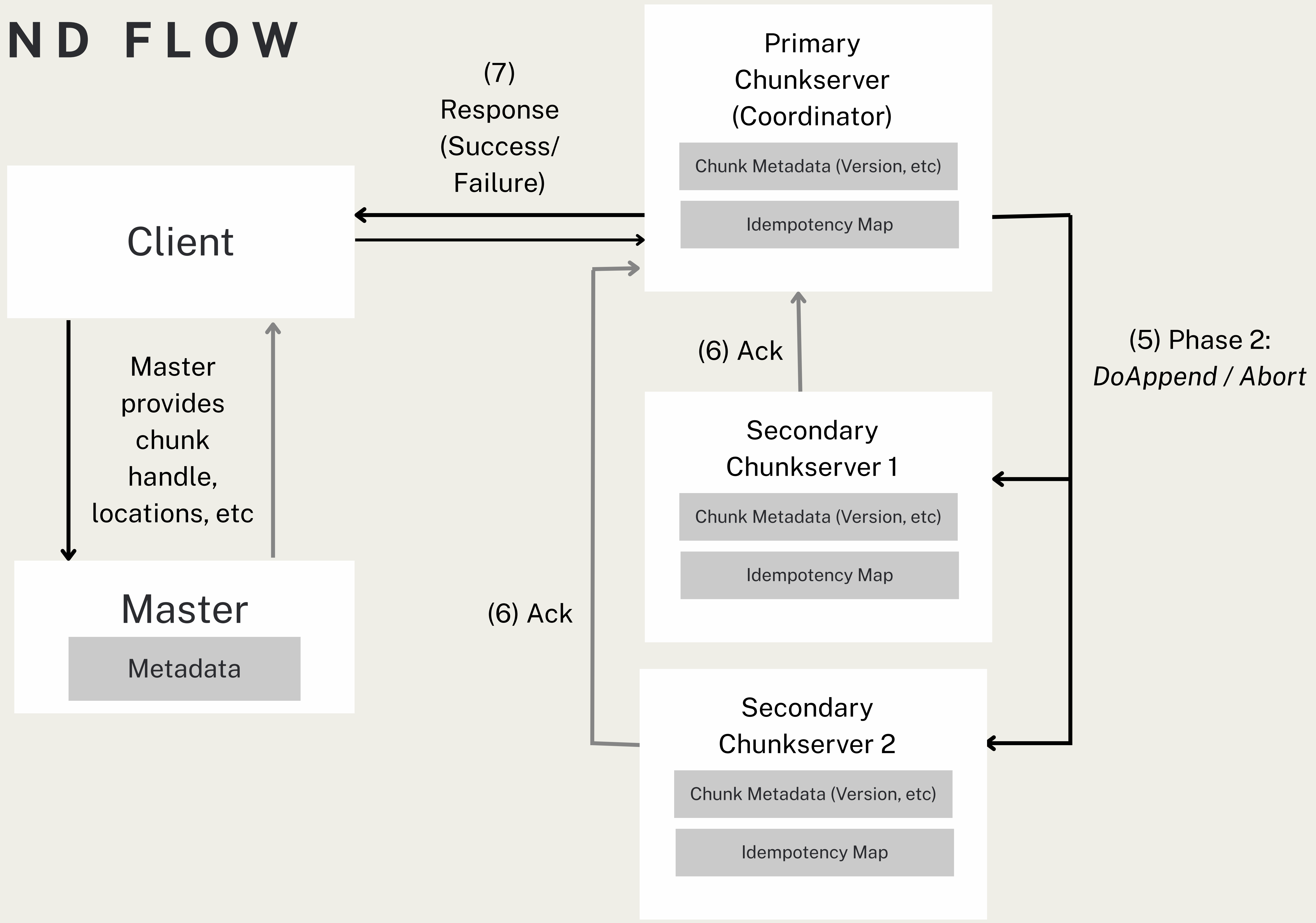
Testing/Benchmarking
Latency and Throughput metrics, Integration Tests

Exactly Once Append
UUIDs + 2-Phase Commit

APPEND FLOW


Client
Master
Metadata
(1) Master provides chunk handle, locations, etc
(2) Append request (data, UUID)
Primary Chunkserver (Coordinator)
Chunk Metadata (Version, etc)
Idempotency Map
(4) Yes/No
Secondary Chunkserver 1
Chunk Metadata (Version, etc)
Idempotency Map
(4) Yes/No
Secondary Chunkserver 2
Chunk Metadata (Version, etc)
Idempotency Map
(3) Phase 1: *CanYouAppend?*

APPEND FLOW



(7)
Response
(Success/
Failure)
Primary
Chunkserver
(Coordinator)
Chunk Metadata (Version, etc)
Idempotency Map
Client
Master
provides
chunk
handle,
locations, etc
(6) Ack
(5) Phase 2:
DoAppend / Abort
Secondary
Chunkserver 1
Chunk Metadata (Version, etc)
Idempotency Map
Master
Metadata
(6) Ack
Secondary
Chunkserver 2
Chunk Metadata (Version, etc)
Idempotency Map

SCENARIO: HANDLING SPLIT BRAIN

The Problem

- A Primary becomes isolated from the Master but can still communicate with clients.
- Isolated Primary attempts to continue serving writes indefinitely.

The Solution

- Leases will expire
- Master grants a **new lease** to a reachable replica, incrementing the chunk version number.
- **Stale Detection:** When the old Primary rejoins, the Master detects its lower **version** number and marks it as stale, preventing corruption.

SCENARIO: LOST ACKNOWLEDGEMENT

Write succeeds, but success response drops.

Client Behavior

- Client times out waiting for a response.
- Retries the append operation using the same Idempotency ID.

Primary Behavior

- Checks internal "append state map" for the incoming ID.
- Finds an existing entry marked status=Committed.
- Does not re-write data.
- Returns the previously assigned offset and version.

Result: Client receives success; data exists only once in the file.

SCENARIO: 2-PHASE COMMIT FAILURE

Scenario: A secondary fails before ACKing PREPARE.

Failure Detection

- Primary receives an error from a secondary or times out waiting for response.

Rollback Procedure (Abort)

1. Primary sends an globally ABORT command to all replicas.
2. Replica Cleanup: All replicas truncate any space reserved for this operation and delete the associated append state entry.

Outcome

- The operation fails atomically.
- Client is responsible for initiating a retry (potentially leading to a new replica set selection).

SCENARIO: 2-PHASE COMMIT FAILURE

Scenario: Secondary crashes after ACKing PREPARE, but before receiving COMMIT.

Immediate Outcome

- The Primary and remaining healthy Secondaries successfully commit the data.
- The crashed secondary is left with a "hole" — space reserved but no data written.

Recovery & Reconciliation

- Restart: When the failed secondary restarts, it will possess an older (stale) chunk version compared to the healthy replicas.
- Master Detection: Master notices the version mismatch during standard heartbeats.
- Resolution: Master marks the replica as stale, eventually triggering garbage collection or re-replication from a healthy source.

PERFORMANCE

(5 chunkservers; 3 replicas; 4 clients)

Exactly Once Append

15–23 ms per operation

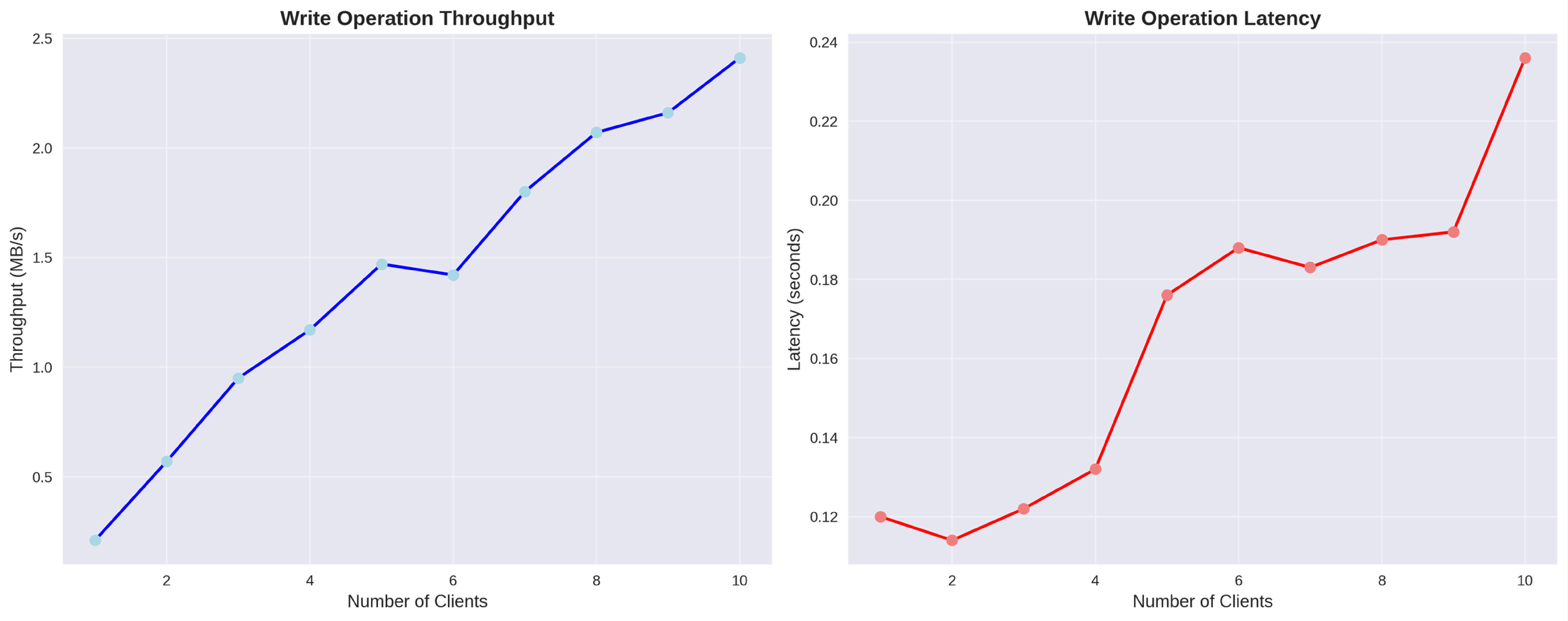
Atleast Once Append

5–10 ms per operation

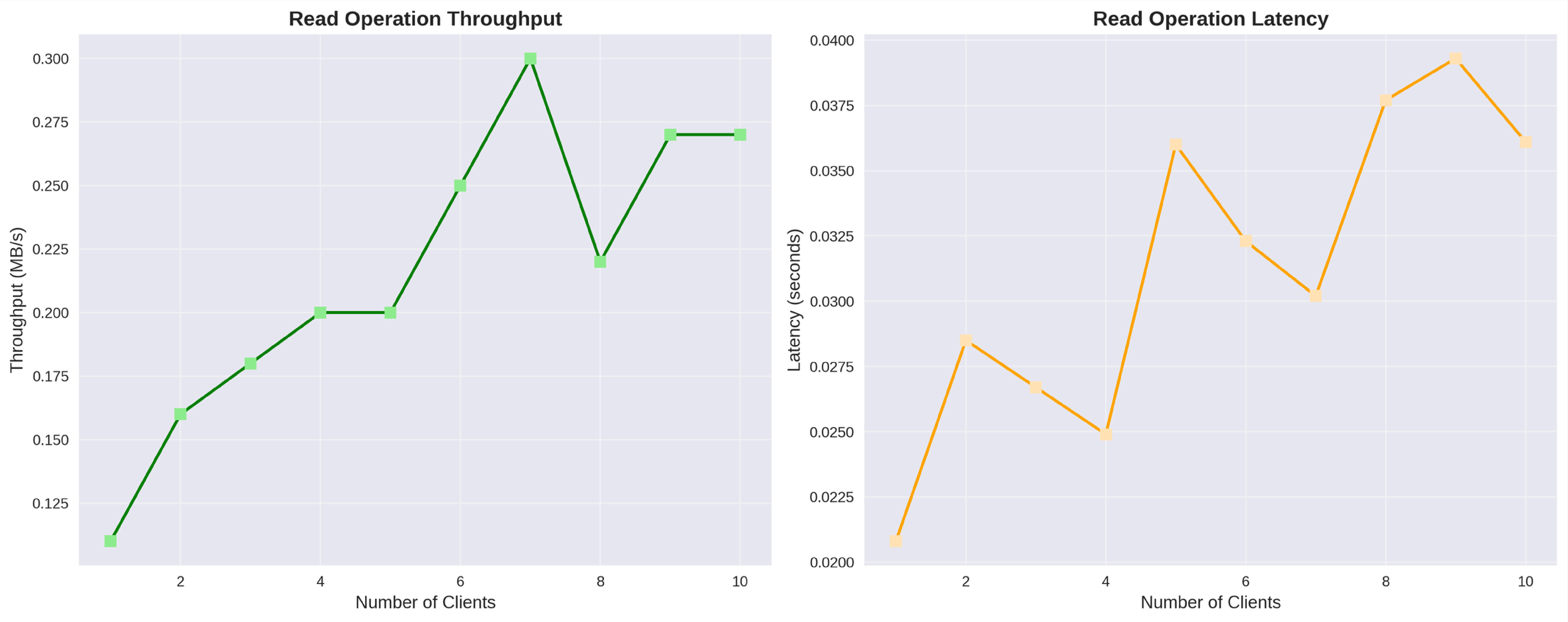
Overheads:

- Execution of the 2PC protocol to ensure atomicity.
- Synchronization across replicas to ensure consistency.

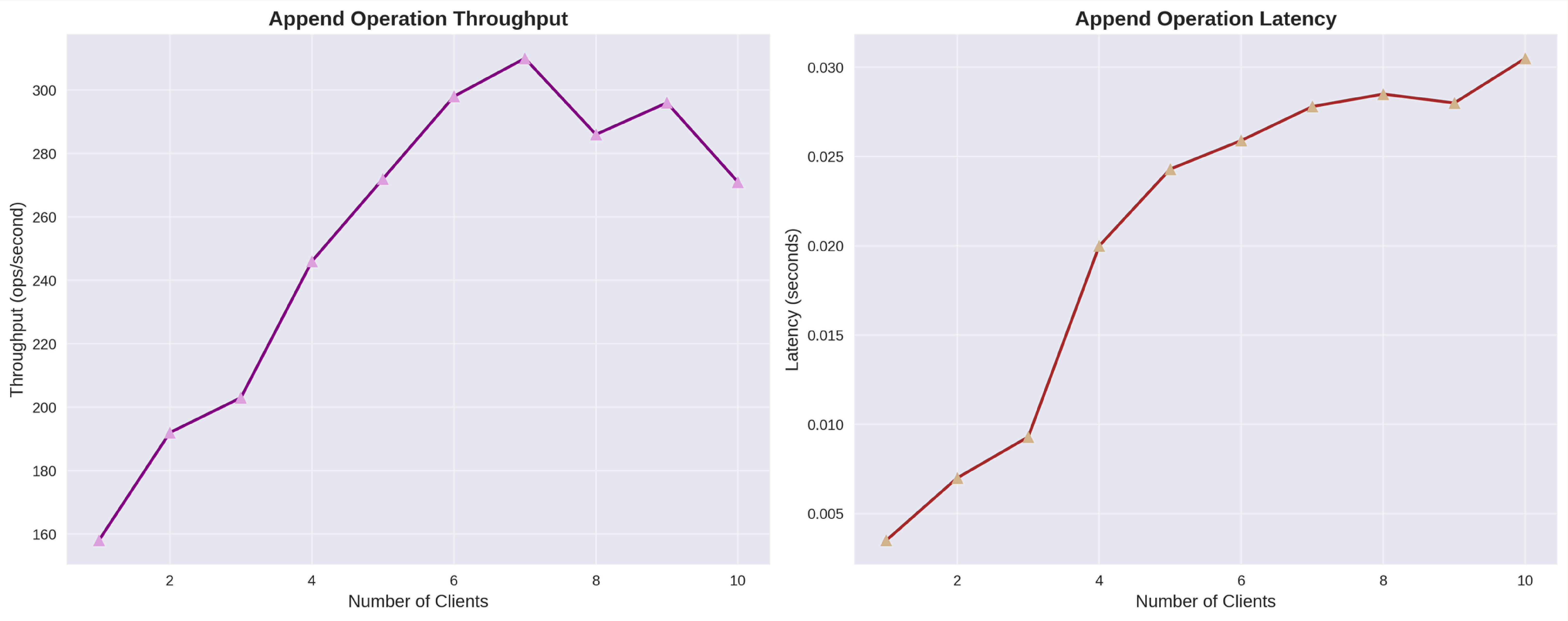
BENCHMARKS : WRITE



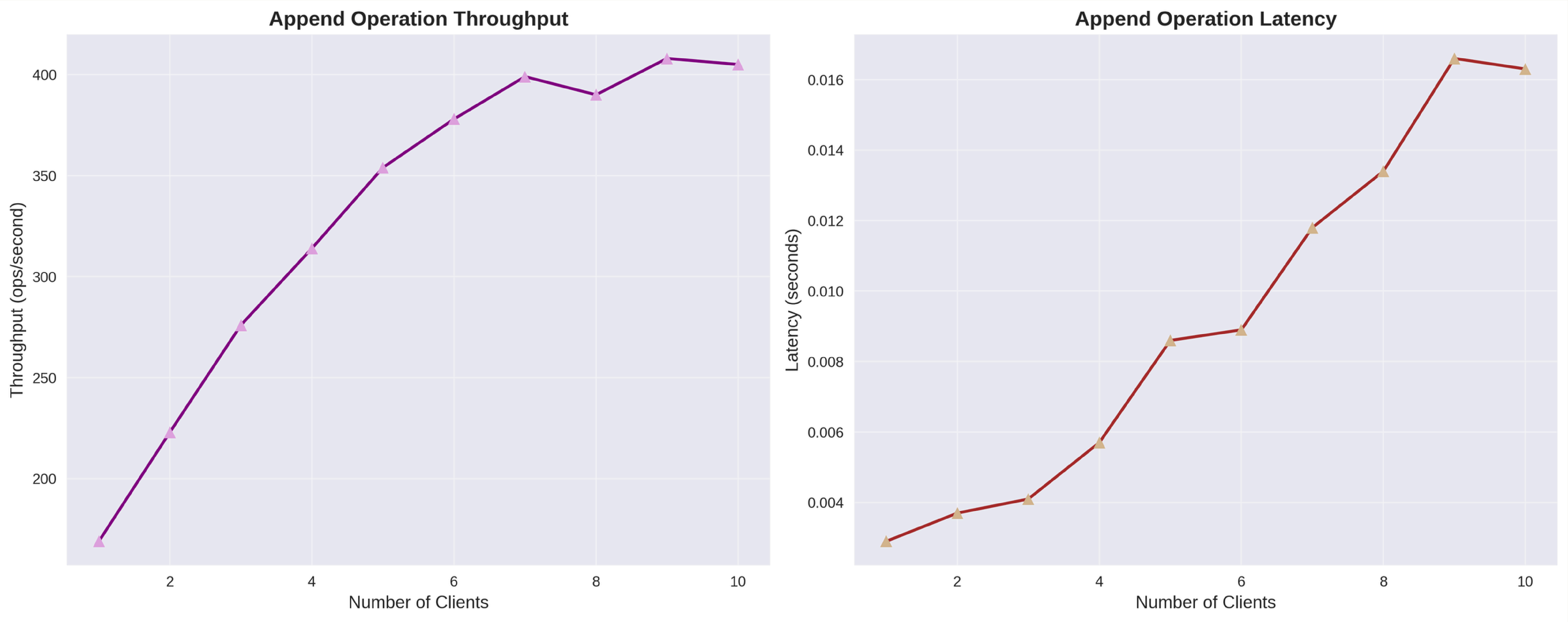
BENCHMARKS : READ



BENCHMARKS : EXACTLY ONCE APPEND



BENCHMARKS : AT LEAST ONCE APPEND



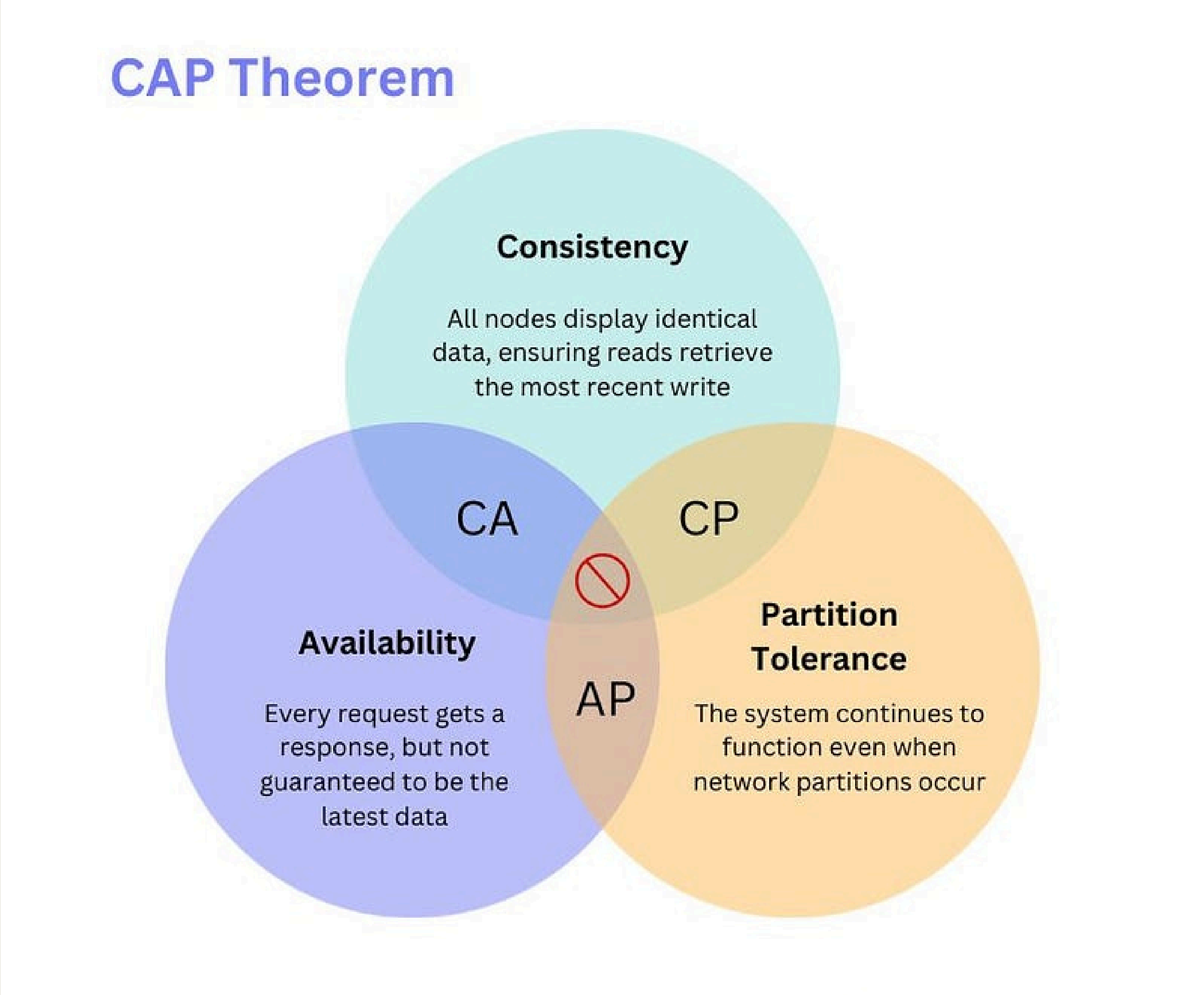
OBSERVATION

The Cost of Correctness

Implementing exactly-once semantics introduces a fundamental trade-off between operation latency and data consistency.

While standard "at-least-once" systems prioritize raw throughput by allowing duplicates during network partitions, our approach prioritizes strict data integrity.

The introduction of the Two-Phase Commit (2PC) protocol and UUID verification adds measurable network overhead, sacrificing the speed of a pipelined write for the guarantee that the file state remains deterministic across all replicas.



FUTURE WORK & LIMITATIONS

Geographical Separation

Adapt for non-localized, geographically distributed environments.
Address higher latency and consistency trade-offs when replicas span across different regions/datacenters.

Multiple Masters

The single Master node becomes a bottleneck for metadata operations.
Implement Master Sharding where different masters manage different namespace sub-trees

Load Balancing

Ooptimizing resource utilization across Chunkservers. Prevent hotspots by distributing file chunks more evenly

Distributed Consensus

Implement Raft or Multi-Paxos for leader election and log replication. This allows the system to make progress as long as a quorum of replicas is alive, increasing avalibility.

Thank you!