Implementing Linearizability at Large Scale and Low Latency

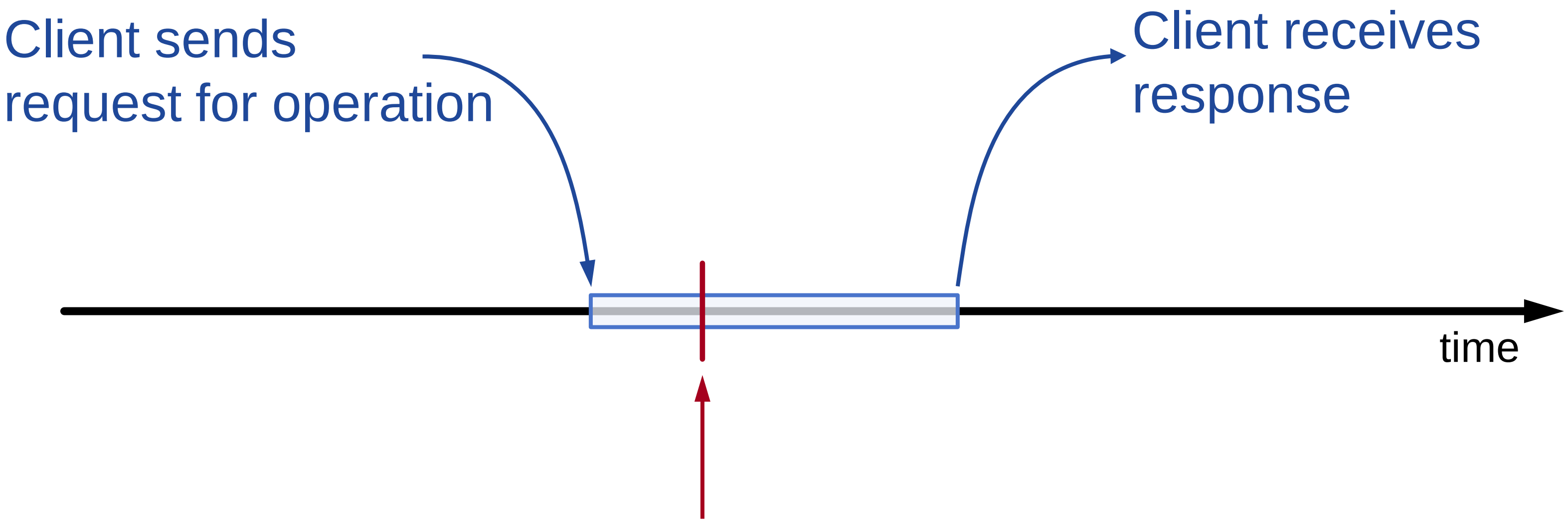
*Collin Lee, **Seo Jin Park**,*
Ankita Kejriwal, †Satoshi Matsushita,
John Ousterhout

Platform Lab
Stanford University **†NEC**

Overview

- **Goal:** take back consistency in large-scale systems
- **Approach:** distinct layer for linearizability
- **Reusable Infrastructure for Linearizability (RIFL)**
  - At-least-once RPC ➔ exactly-once RPC
  - Records RPC results durably
  - To handle reconfiguration, associates metadata with object
- **Implemented on distributed KV store, RAMCloud**
  - Low latency: **< 5% (500ns)** latency overhead
  - Scalable: supports states for 1M clients
- **RIFL simplified implementing transactions**
  - Simple distributed transaction commits in ~**22 μs**
  - Outperforms H-Store

What is Linearizability?

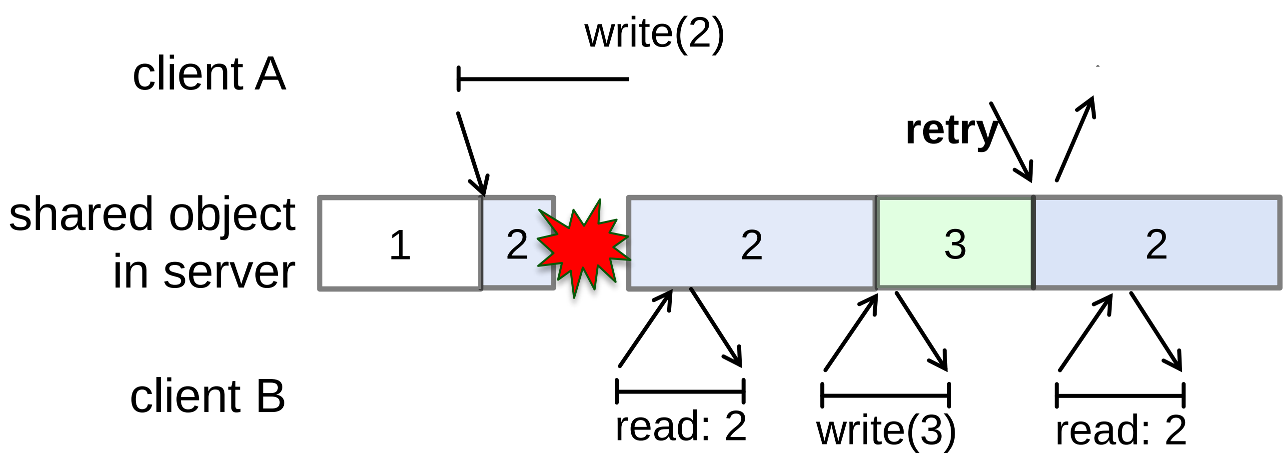


Behaves as if executing exactly once and instantaneously

Strongest form of consistency for concurrent systems

What is Missing from Existing Systems?

- **Most systems: at-least-once semantics, not exactly-once**
  - Retry (possibly) failed operations after crashes
  - Idempotent semantics: repeated executions *O.K.?*



- **At-least-once + idempotency ≠ linearizability**
- **Need exactly-once semantics!**

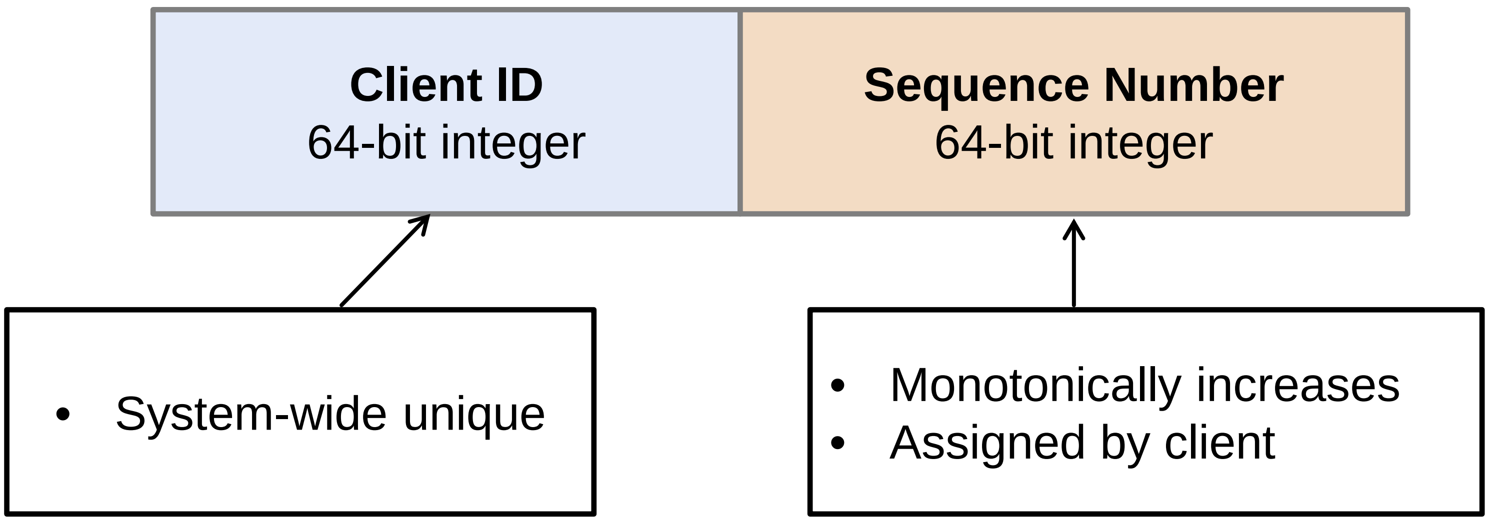
Architecture of RIFL

- **RIFL saves results of RPCs**
- **If client retries,**
  - Don't re-execute
  - Return saved result

➢ **Key problems**
  - Unique identification for each RPC
  - Durable completion record
  - Retry rendezvous
  - Garbage collection

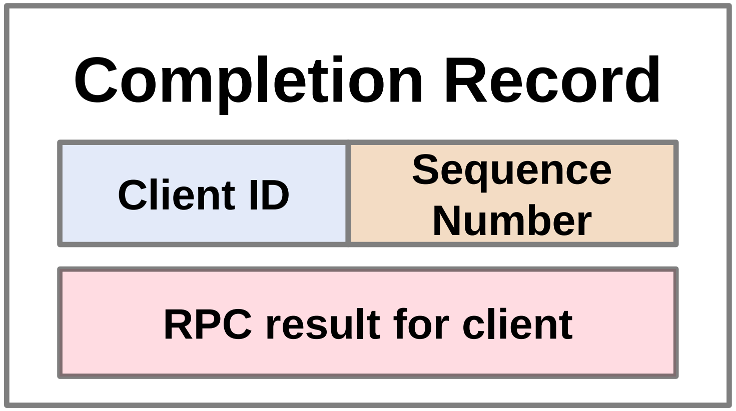
1) RPC Identification

- **Each RPC must have a unique ID**
- **Retries use the same ID**
- **Client assigns RPC IDs**



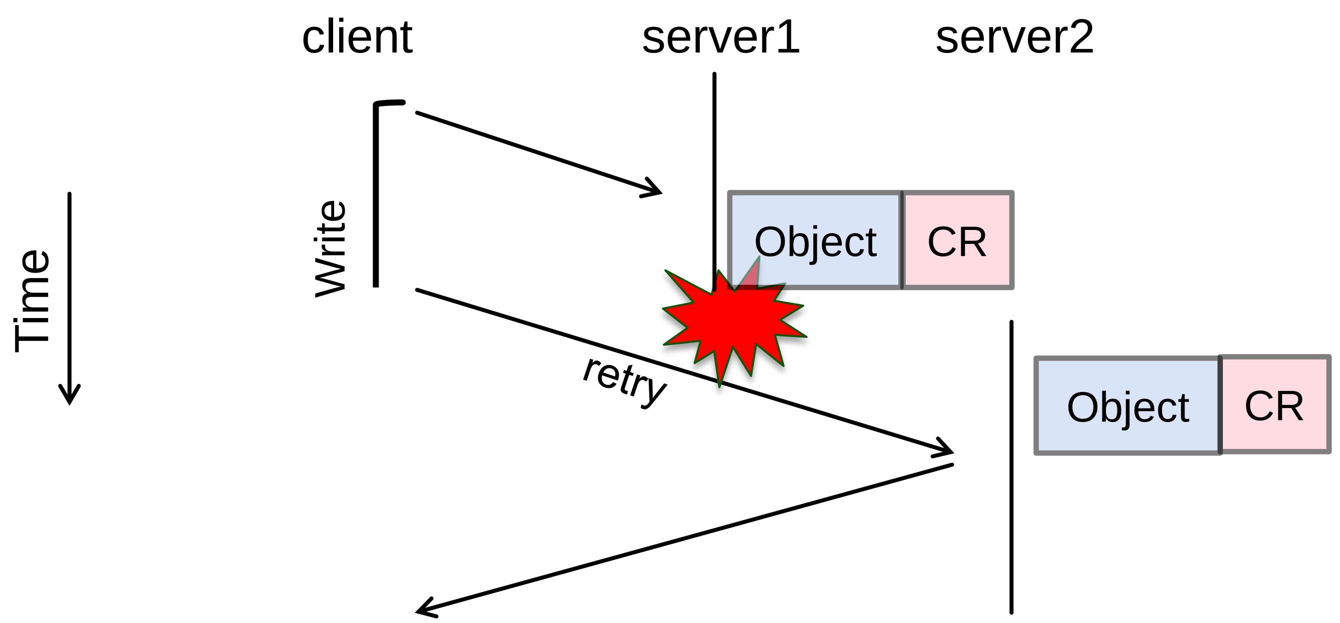
2) Durable Completion Record

- Written when an operation completes
- Same durability as object(s) being mutated
- Atomically created with object mutation



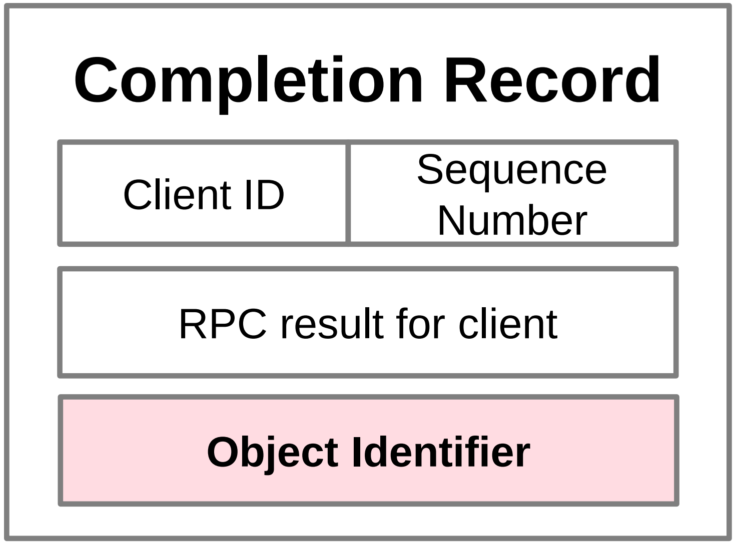
3) Retry Rendezvous

- Data migration is popular in large systems (eg. crash recovery)
- Retries must find completion record



3) Retry Rendezvous (cont.)

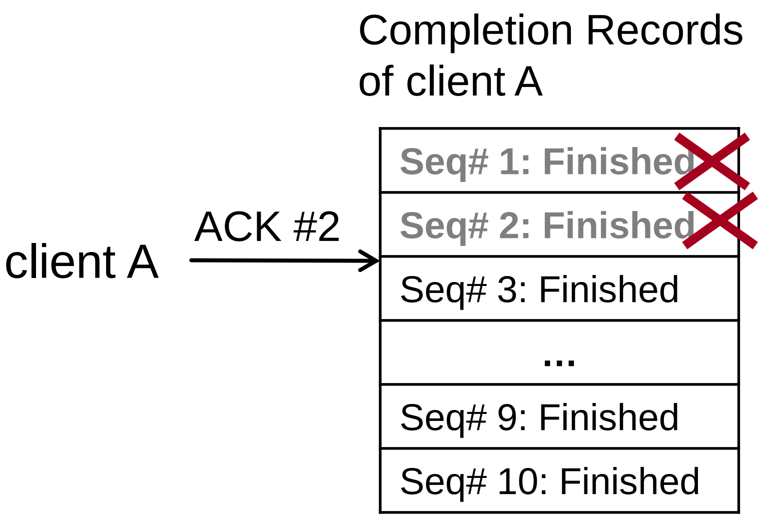
- **Associate each RPC with a specific object**
- **Completion record follows the object during migration**



4) Garbage Collection

- **Lifetime of completion record ≠ lifetime of object value**
- **Can't GC if client may retry later**
- **Server knows a client will never retry if**

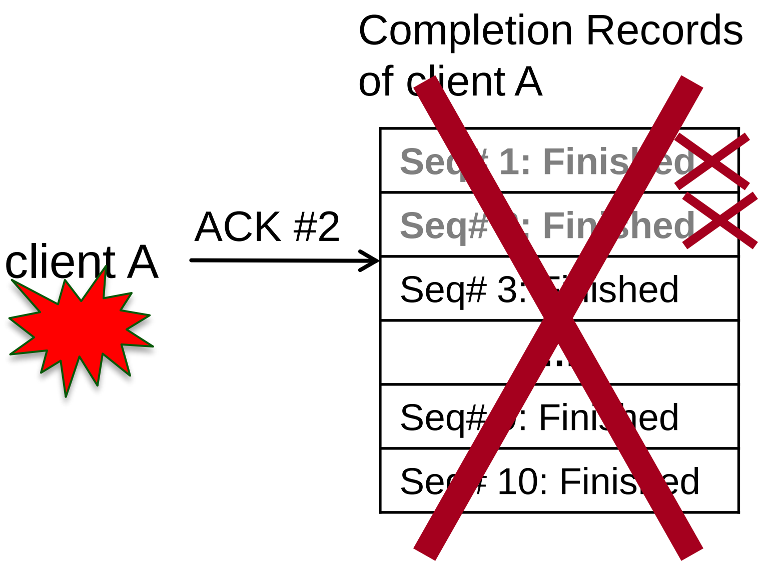
1. Client acknowledges receipt of RPC result



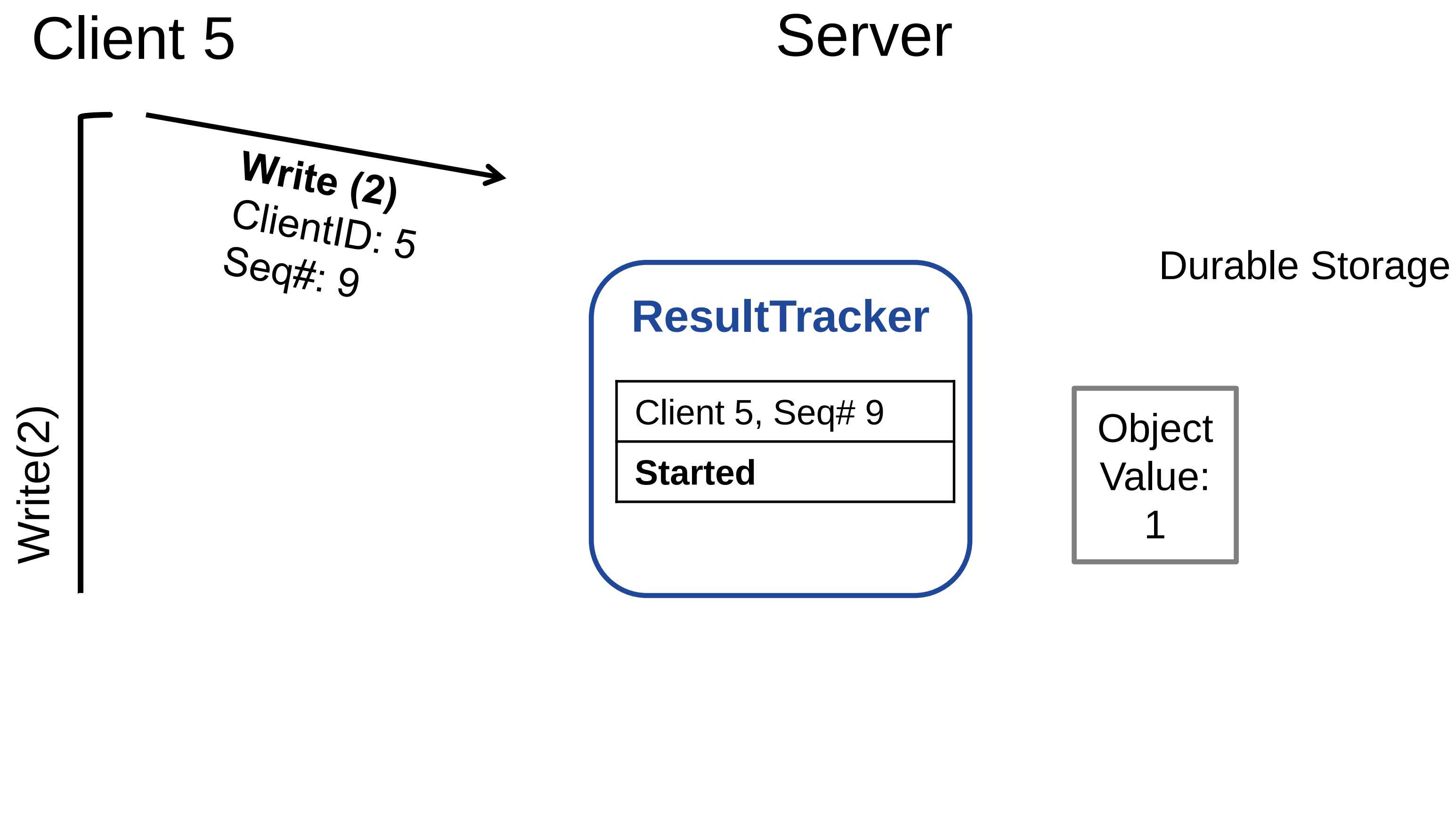
4) Garbage Collection

- **Lifetime of completion record ≠ lifetime of object value**
- **Can't GC if client may retry later**
- **Server knows a client will never retry if**

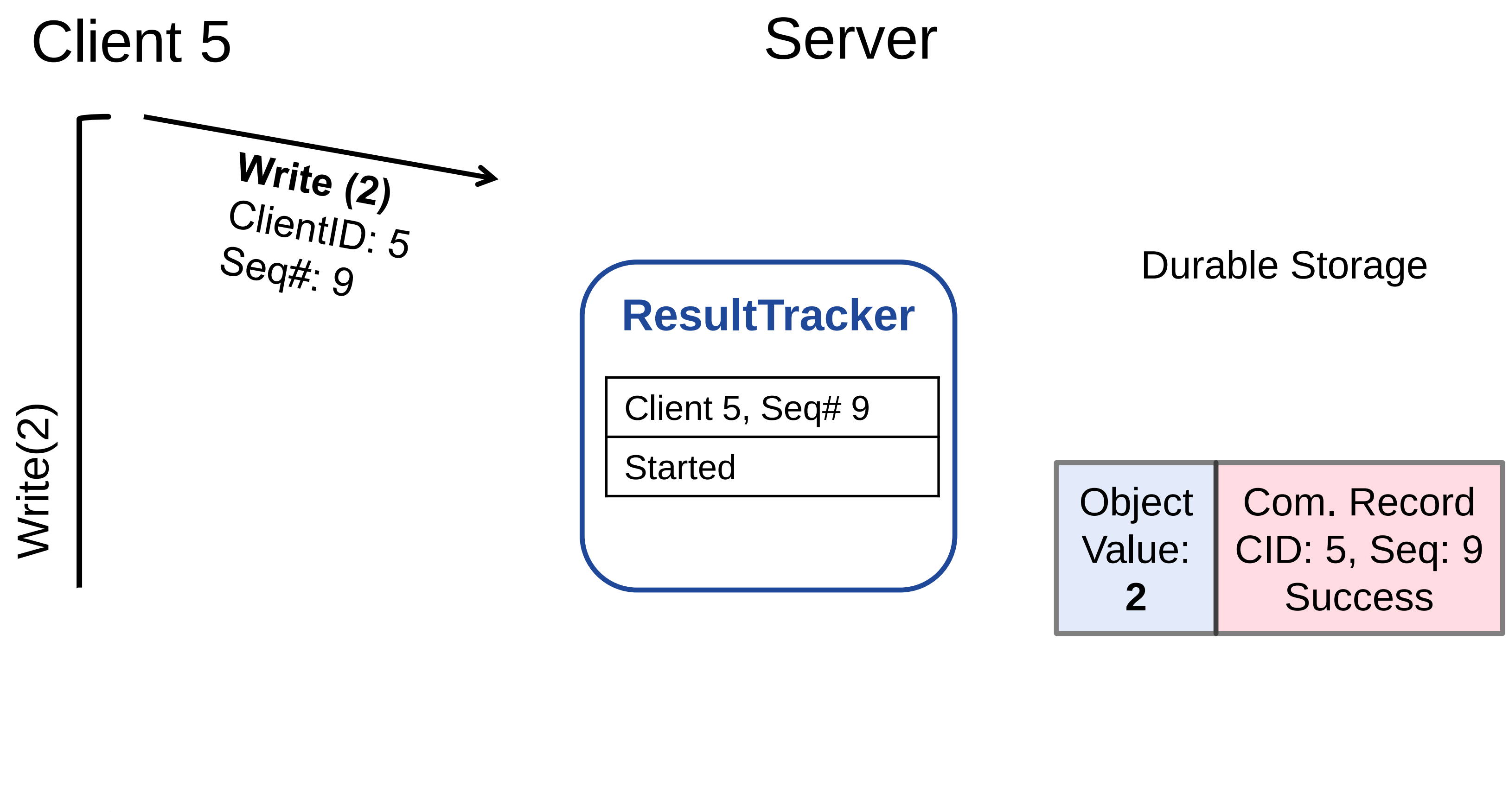
1. Client acknowledges receipt of RPC result
2. Detect client crashes with lease.



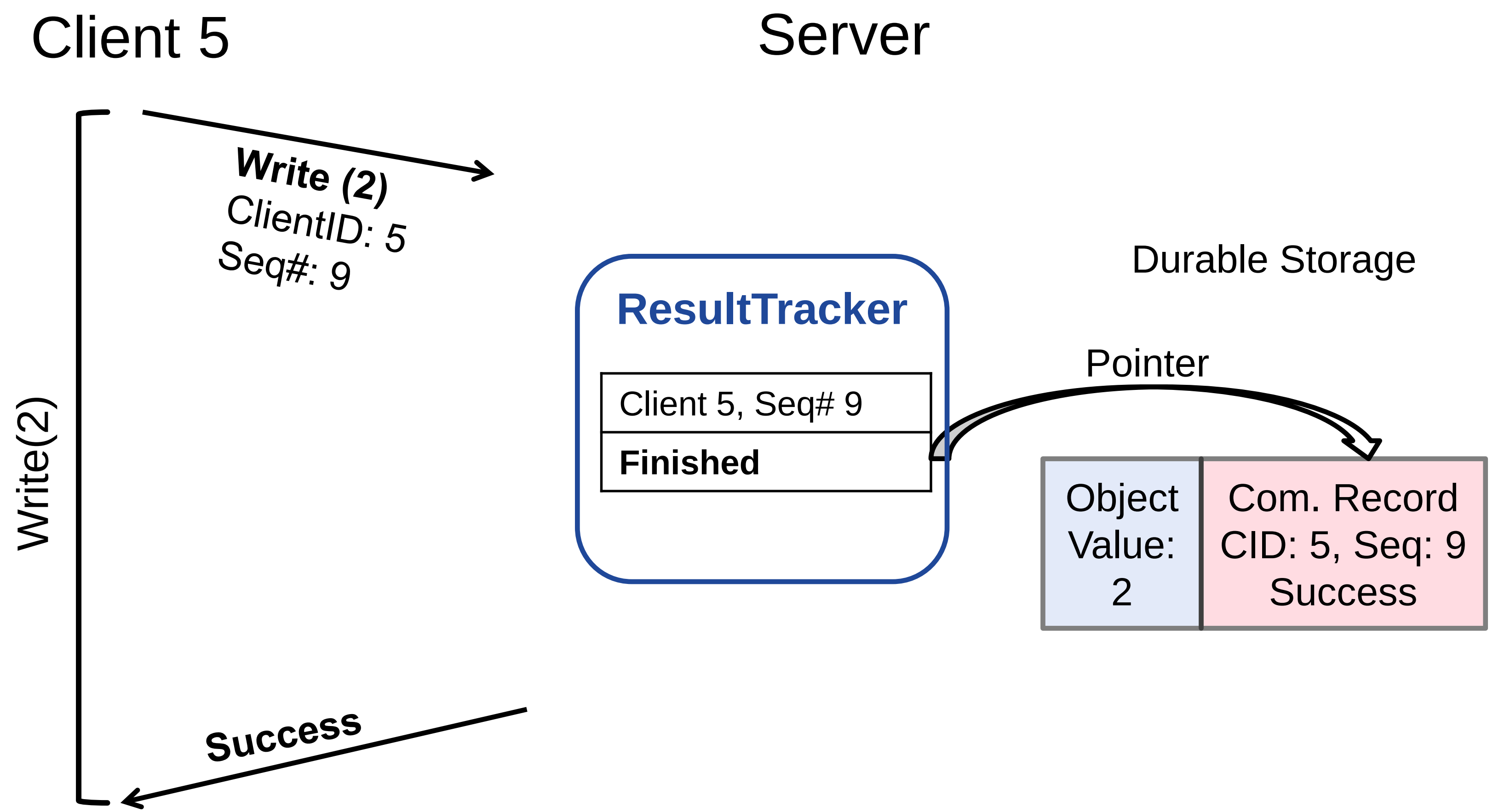
Linearizable RPC in Action



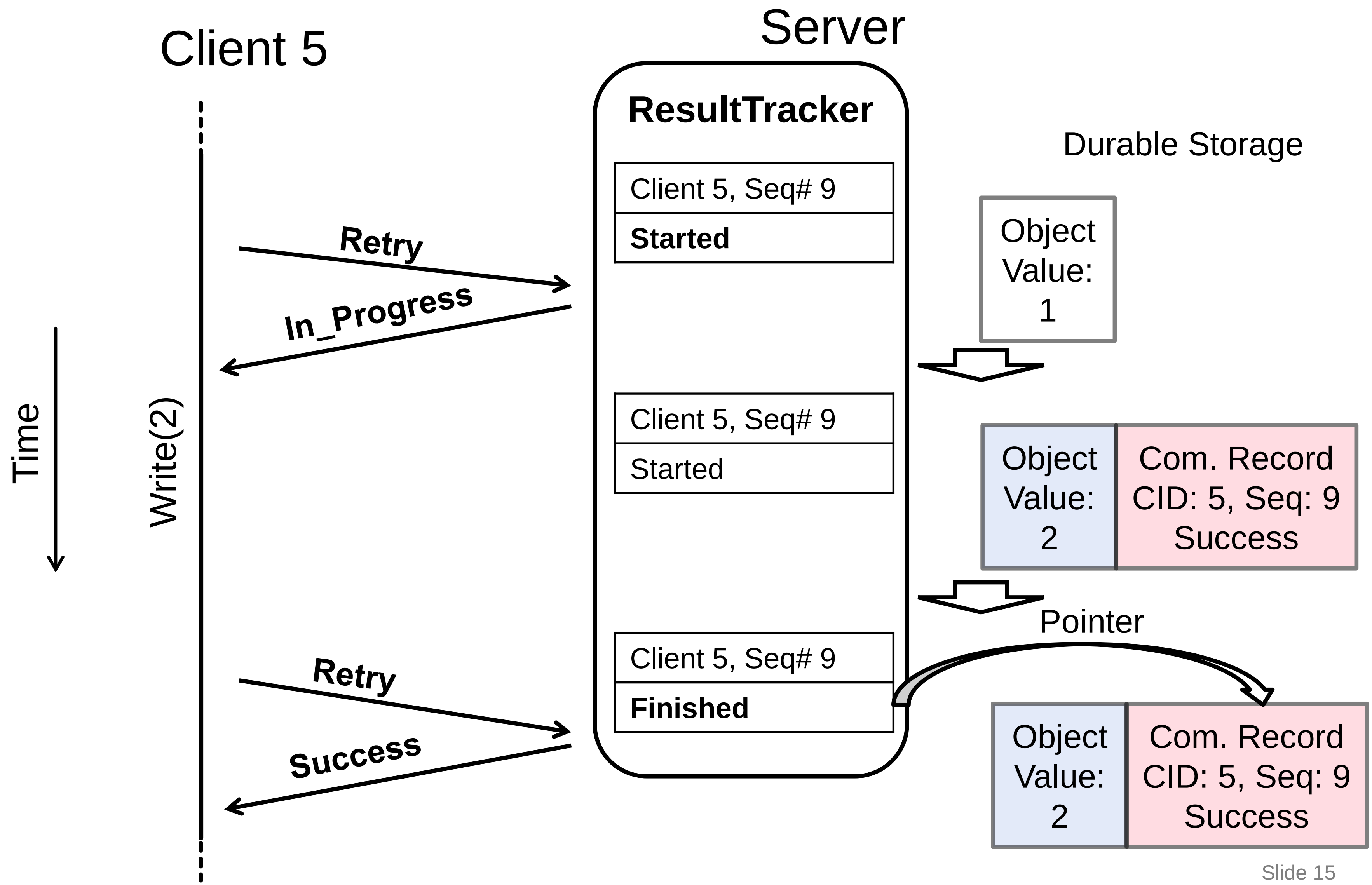
Linearizable RPC in Action



Linearizable RPC in Action



Handling Retries



Performance of RIFL in RAMCloud

- **Achieved linearizability without hurting performance of RAMCloud**
  - Minimal overhead on latency(< 5%) and throughput (~0%)
  - Supports state for 1M clients

Why RAMCloud?

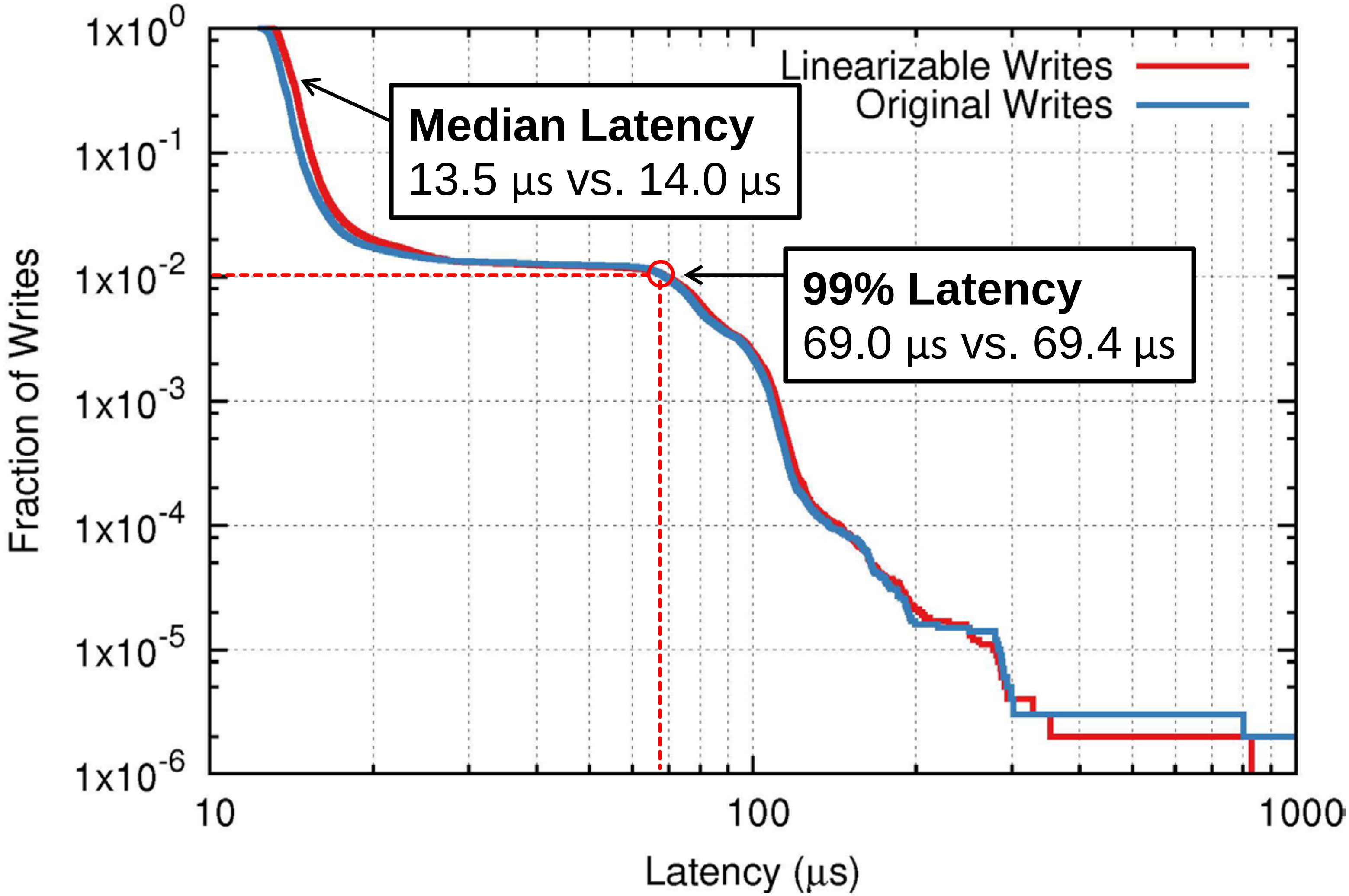
- **General-purpose distributed in-memory key-value storage system**
- **Durability:** 3-way replication
- **Fast recovery:** 1~2 sec for server crash
- **Large scale:** 1000+ servers, 100+ TB
- **Low latency:** 4.7 μs read, 13.5 μs write (100B object)

- **RIFL is implemented on top of RAMCloud**
  - Core: 1200 lines of C++ for infrastructure
  - Per operation: 17 lines of C++ to make an operation linearizable

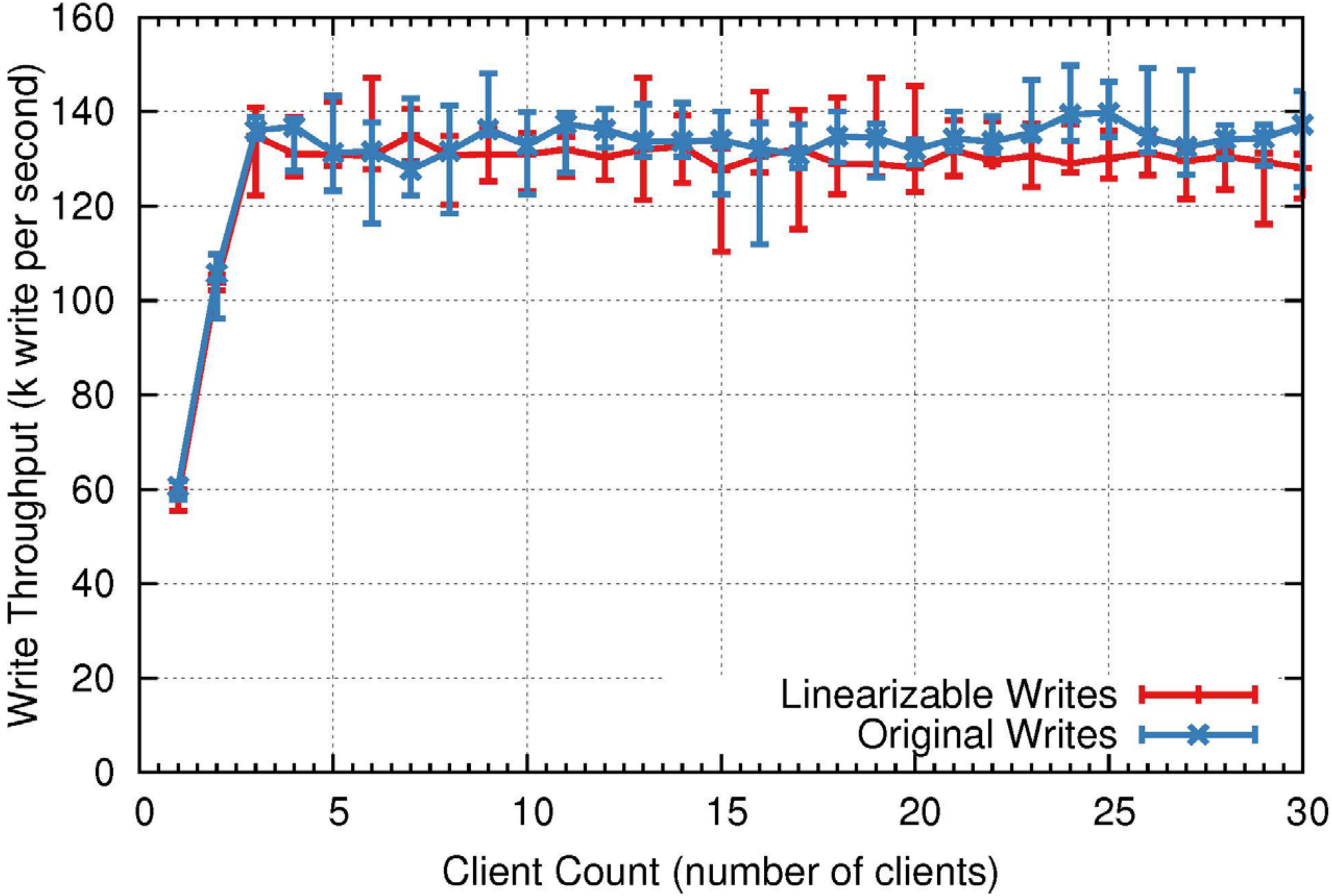
Experimental Setup

- **Server: Xeon 4 cores at 3 GHz**
- **Fast Network**
  - Infiniband (24 Gbps)
  - Kernel-bypassing transport (RAMCloud default)

Impact on Latency

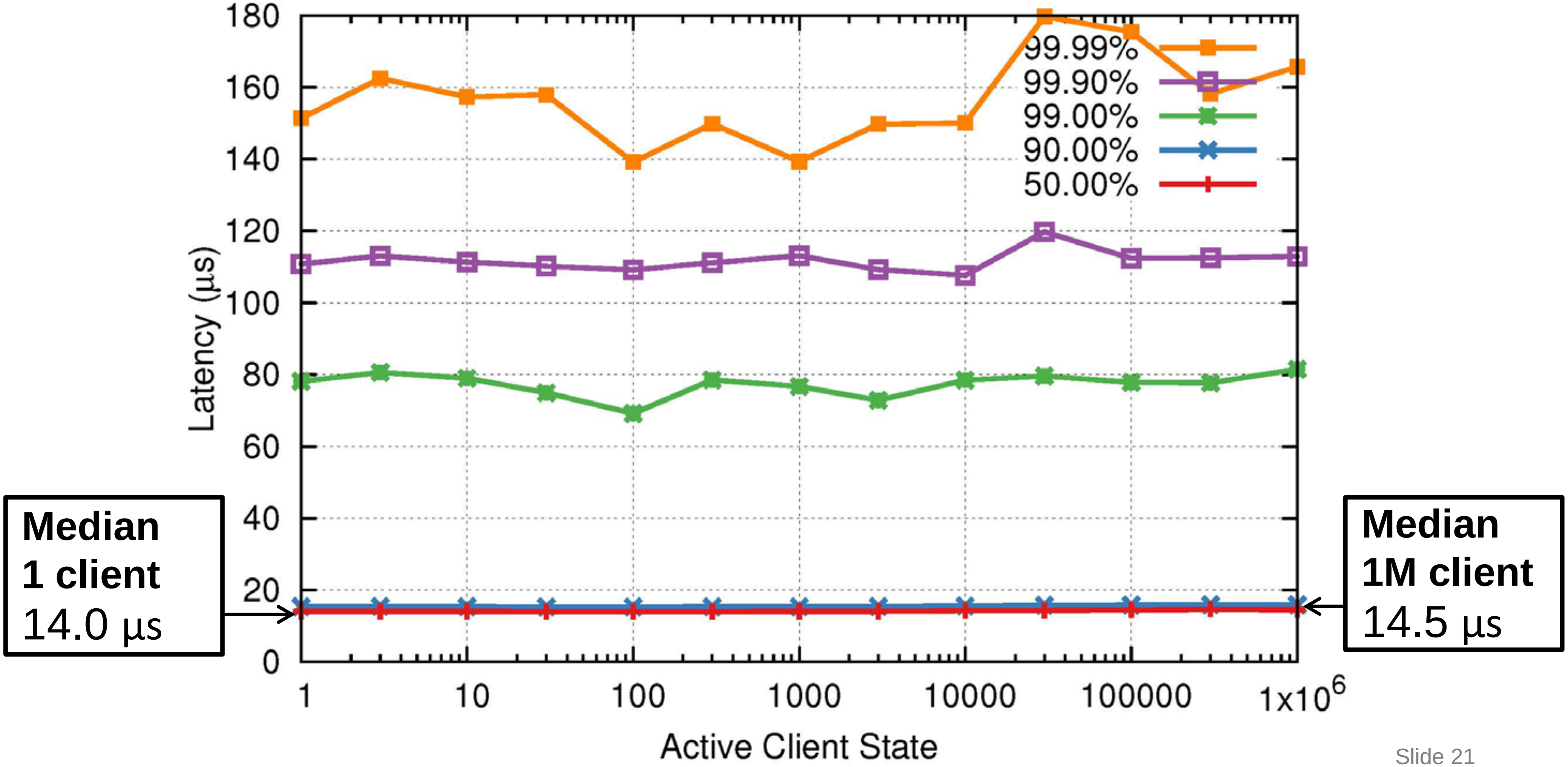


Impact on Throughput



Impact of Many Client States

- **Storage impact:** 116B per client ➔ 116MB for 1M clients
- **Latency impact** (linearizable write, unloaded):



Case study: Distributed Transactions with RIFL

- **Extended use of RIFL for more complex operations**
- **Two-phase commit protocol based on Sinfonia**
- **RIFL reduced mechanisms of Sinfonia significantly**
- **Lowest possible round-trip for distributed transactions**

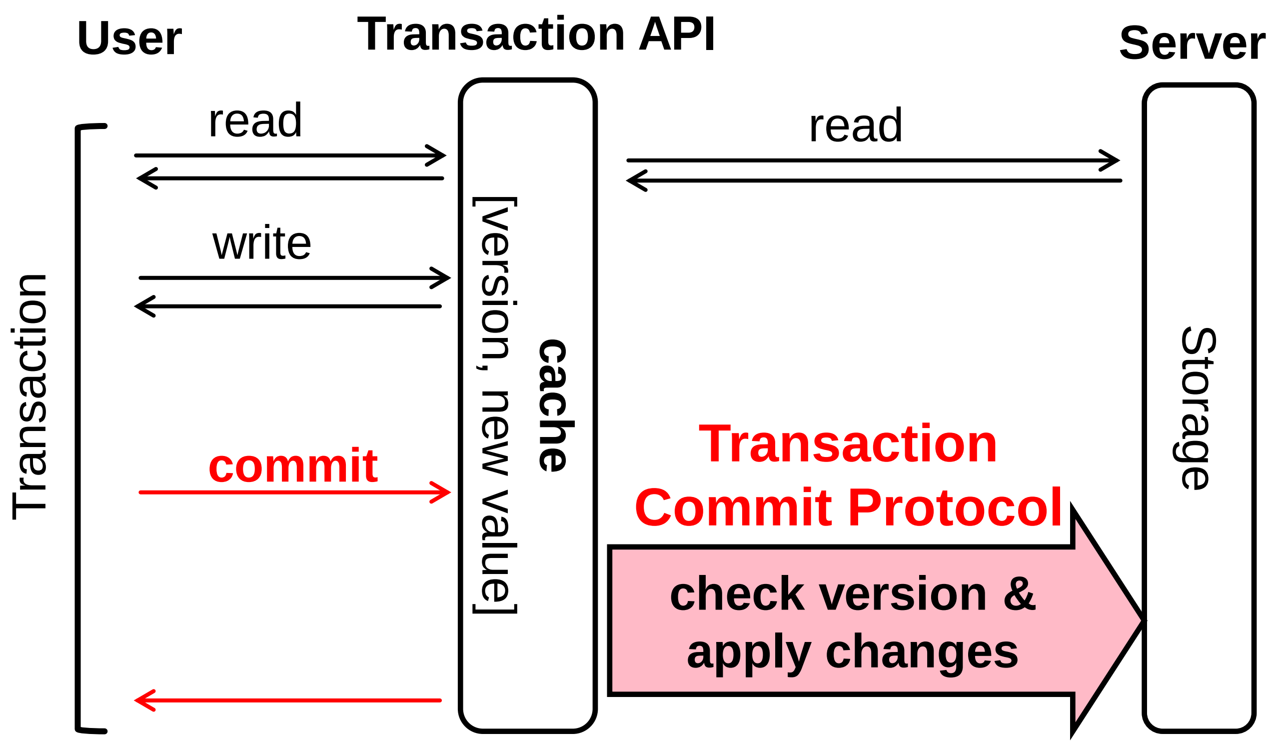
Transaction Client API

```
class Transaction {
        read(tableId, key) => blob
        write(tableId, key, blob)
        delete(tableId, key)
        commit() => COMMIT or ABORT
}
```

- **Optimistic concurrency control**
- **Mutations are “cached” in client until commit**

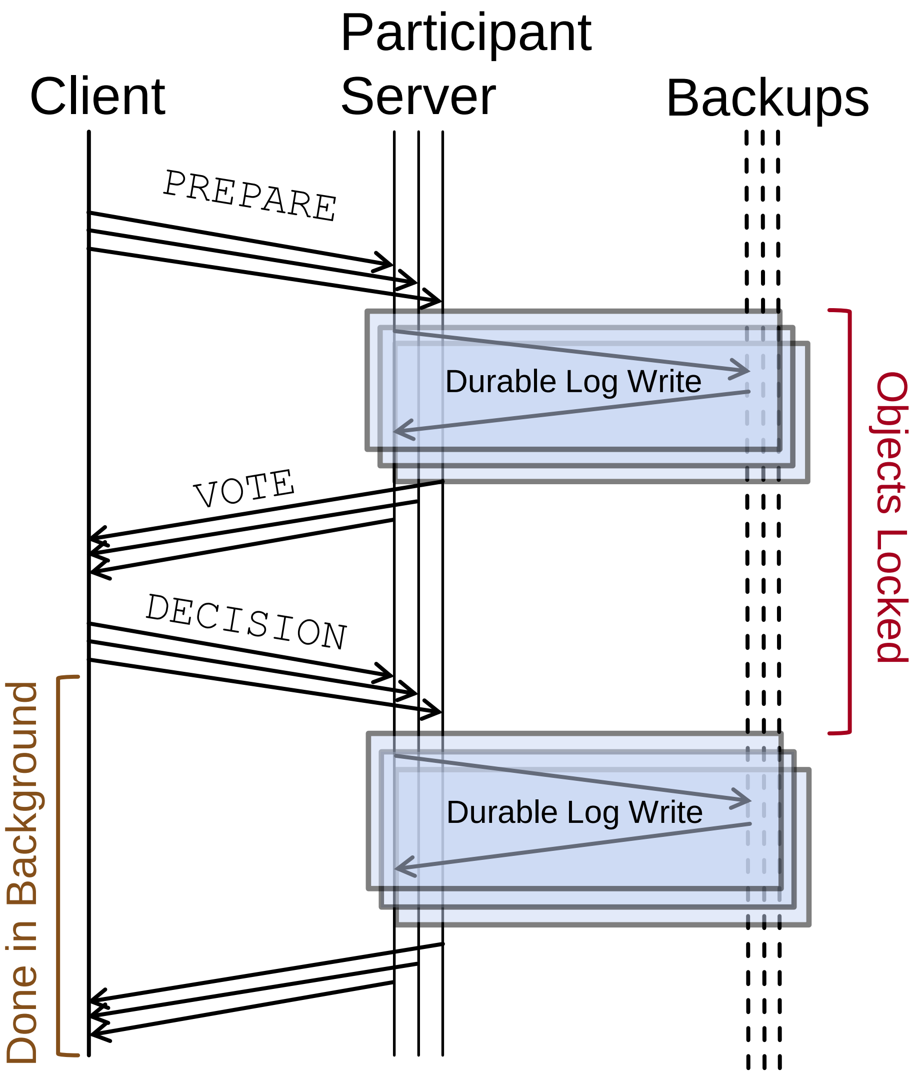
Transaction Commit Semantics

- **commit(): atomic multi-object operation**
  - Operations in client's cache are transmitted to servers
  - Conditioned on a version (same version ≡ object didn't change)



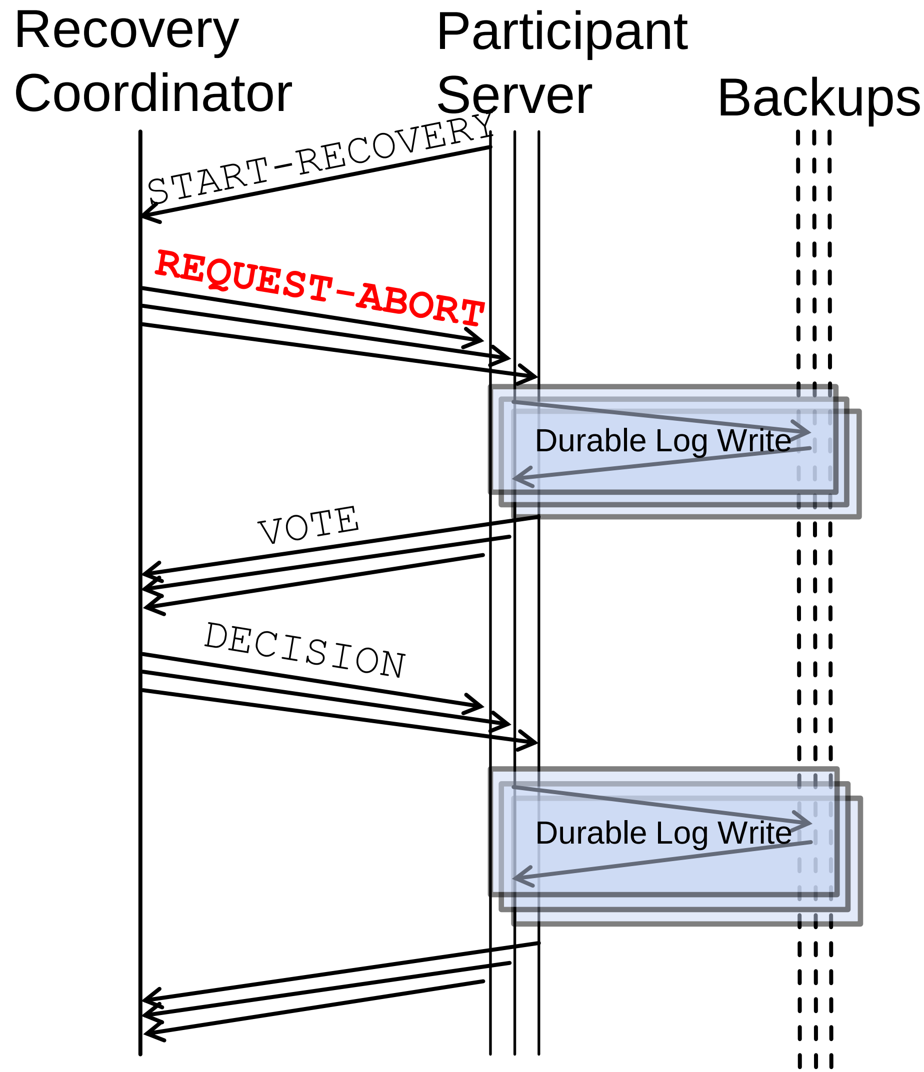
Transaction Commit Protocol

- **Client-driven 2PC**
- **RPCs:**
  - `PREPARE() => VOTE`
  - `DECISION()`
- **Fate of TX is determined after 1st phase**
- **Client blocked for 1 RTT + 1 log write**
- **Decisions processed in background**



Transaction Recovery on Client-crash

- **Server-driven 2PC**
- **Initiated by "worried" server**
- **RPCs:**
  - `START-RECOVERY()`
  - **`REQUEST-ABORT() => VOTE`**
  - `DECISION()`



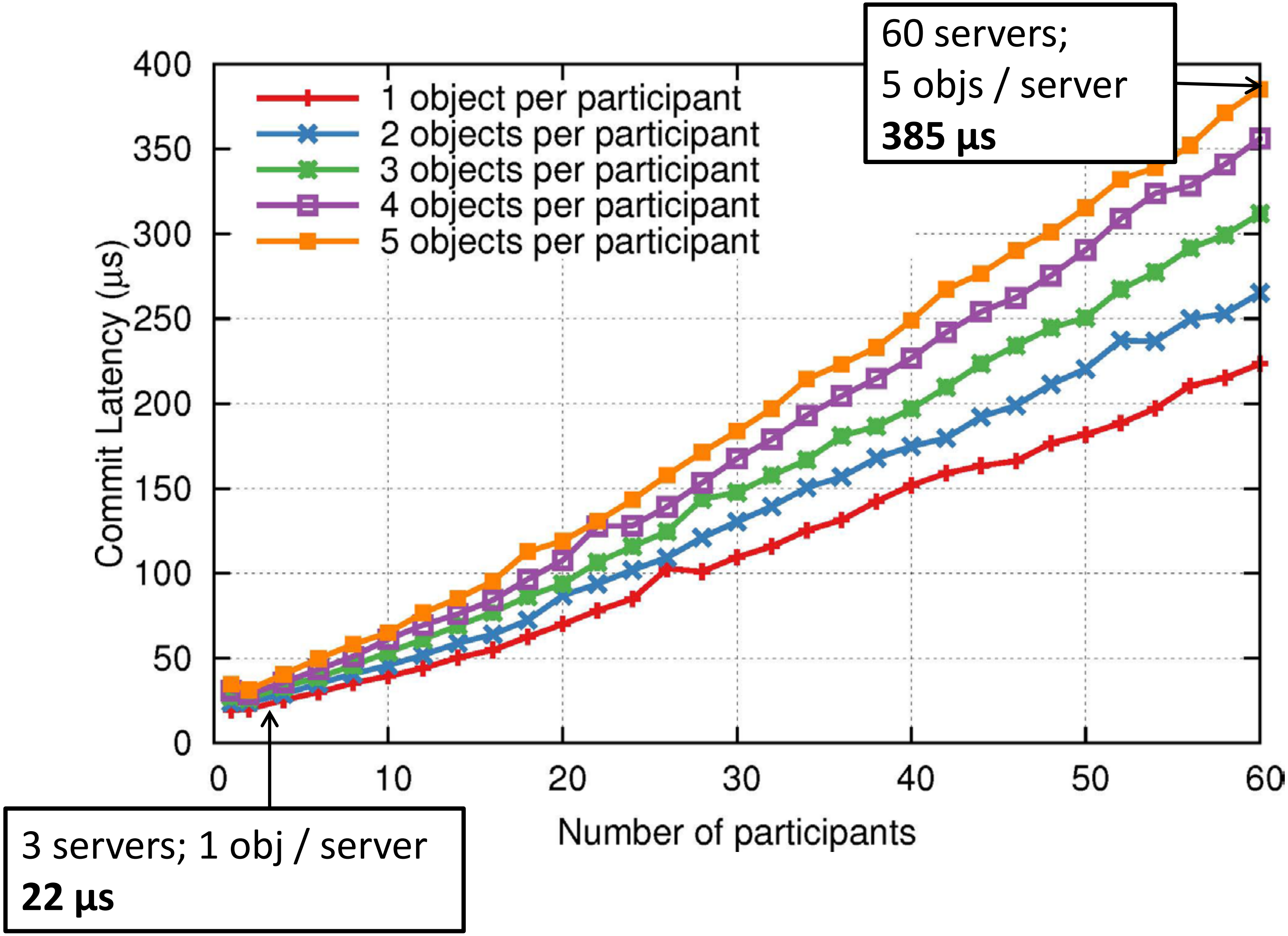
RIFL Simplifies TX Recovery

- **`PREPARE()` => `VOTE` is linearizable RPC**
- **Server crash: client retries PREPARE**
- **Client crash: recovery coordinator sends fake PREPARE (REQUEST-ABORT)**
  - Query ResultTracker with **same RPC ID** from client
  - Writes completion record of PREPARE / REQUEST-ABORT
- **Race between client's 2PC and recovery coordinator's 2PC is safe**

Performance of Transactions in RAMCloud

- Simple distributed transactions commits in 22 µs
- TPC-C benchmark shows RIFL-based transactions outperform H-Store

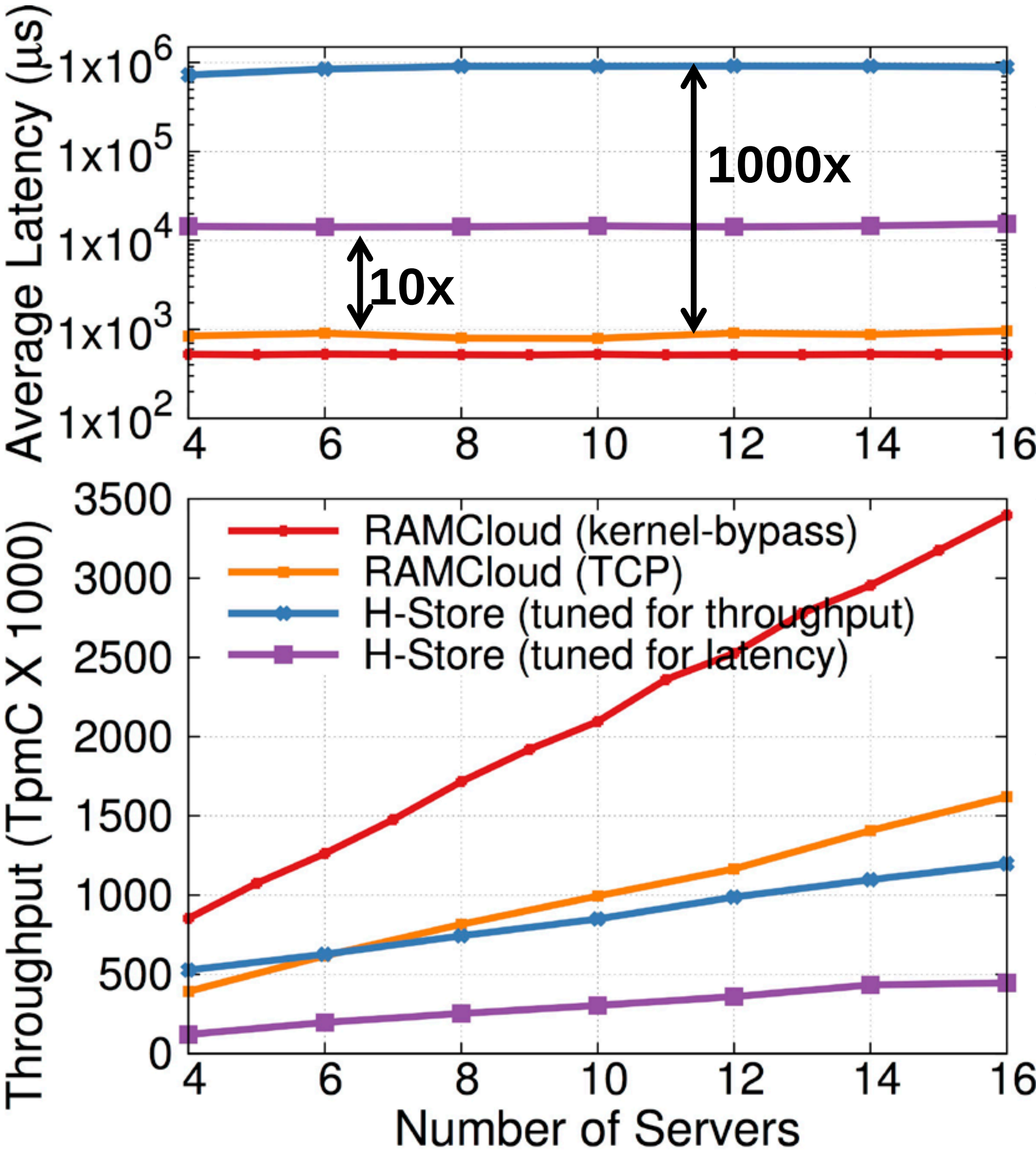
Latency of Transactions



TPC-C Benchmark

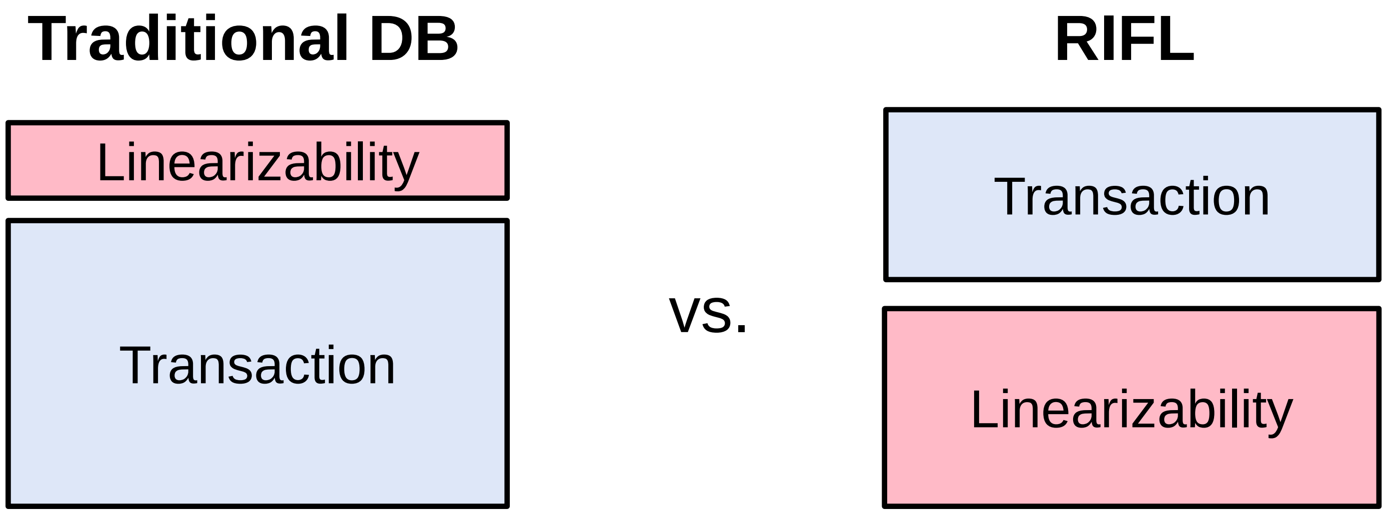
- **TPC-C simulates order fulfillment systems**
  - New-Order transaction: 23 reads, 23 writes
  - Used full-mix of TPC-C (~10% distributed)
  - Latency is measured from end to end
  - Modified TPC-C for benchmark to increase server load
- **Compared with H-Store**
  - Main-memory DBMS for OLTP
- **Two RAMCloud configurations**
  - Kernel-bypass for maximum performance
  - Kernel TCP for fair comparison

TPC-C Performance



- **TpmC:** NewOrder committed per minute
- **210,000 TpmC / server**
- **Latency: ~500µs**
- **RAMCloud faster than H-Store** (even over TCP)
- **Limited by serial log-replication** (need batching)

Should linearizability be a foundation?



- **Faster simple operations (eg. atomic increment)**
  - lower latency
  - higher throughput
- **Can be added on top of existing systems**
- **Better modular decomposition**
  - easier to implement transaction

Conclusion

- **Distinct layer for linearizability →** take back consistency in large-scale systems
- **RIFL saves results** of RPCs; If client **retries,** returns saved result **without re-executing**
  - 0.5us (< 5%) latency overhead, almost no throughput overhead.
- **RIFL** makes **transactions** easier
  - RIFL-based RAMCloud transaction: ~**20 μs** for commit
  - Outperform H-Store for TPC-C benchmark

Questions