Exploiting Commutativity For Practical Fast Replication

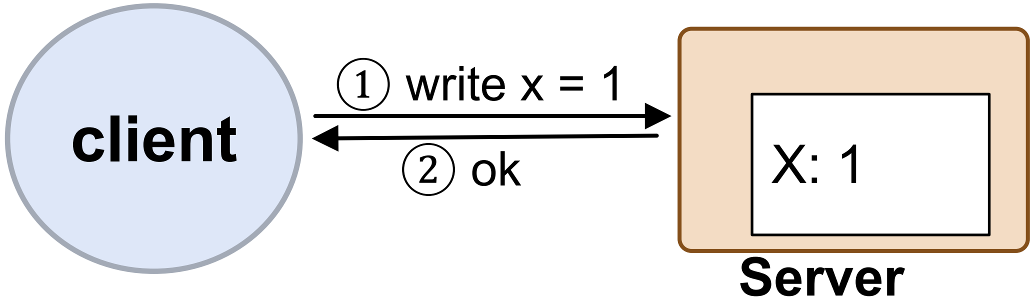
Seo Jin Park and John Ousterhout

PLATFORMLAB
Stanford University

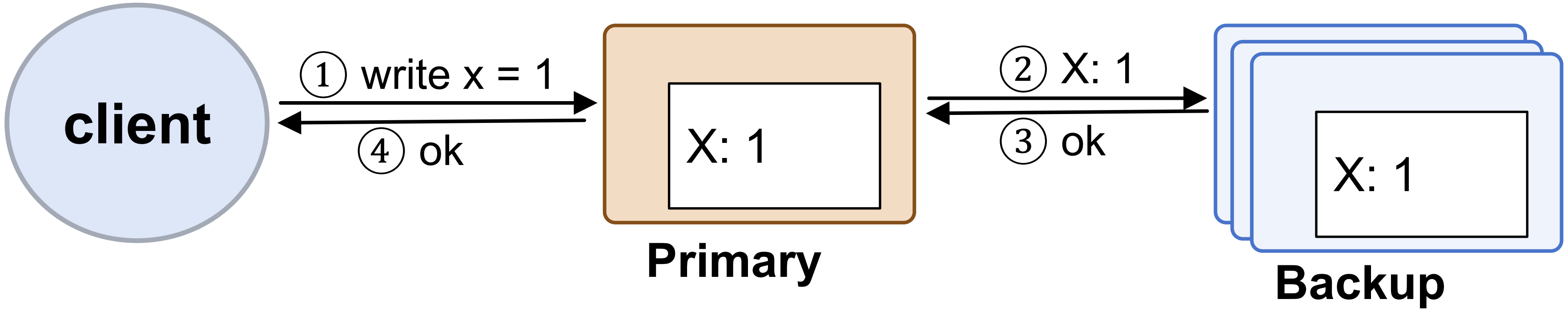
Overview

- **Problem:** consistent replication adds latency and throughput overheads
  - Why? Replication happens after ordering
- **Key idea:** exploit **commutativity** to enable fast replication before ordering
- **CURP (Consistent Unordered Replication Protocol)**
  - Clients replicate in **1 round-trip time** (RTT) if operations are **commutative**
  - Simple augmentation on existing primary-backup systems
- **Results**
  - RAMCloud's performance improvements
    - Latency: 14 μs → 7.1 μs (no replication: 6.1 μs)
    - Throughput: 184 kops/sec → 728 kops/s (~4x)
  - Redis cache is now fault-tolerant with small cost (12% latency ↑, 18% throughput ↓)

Consistent Replication Doubles Latencies

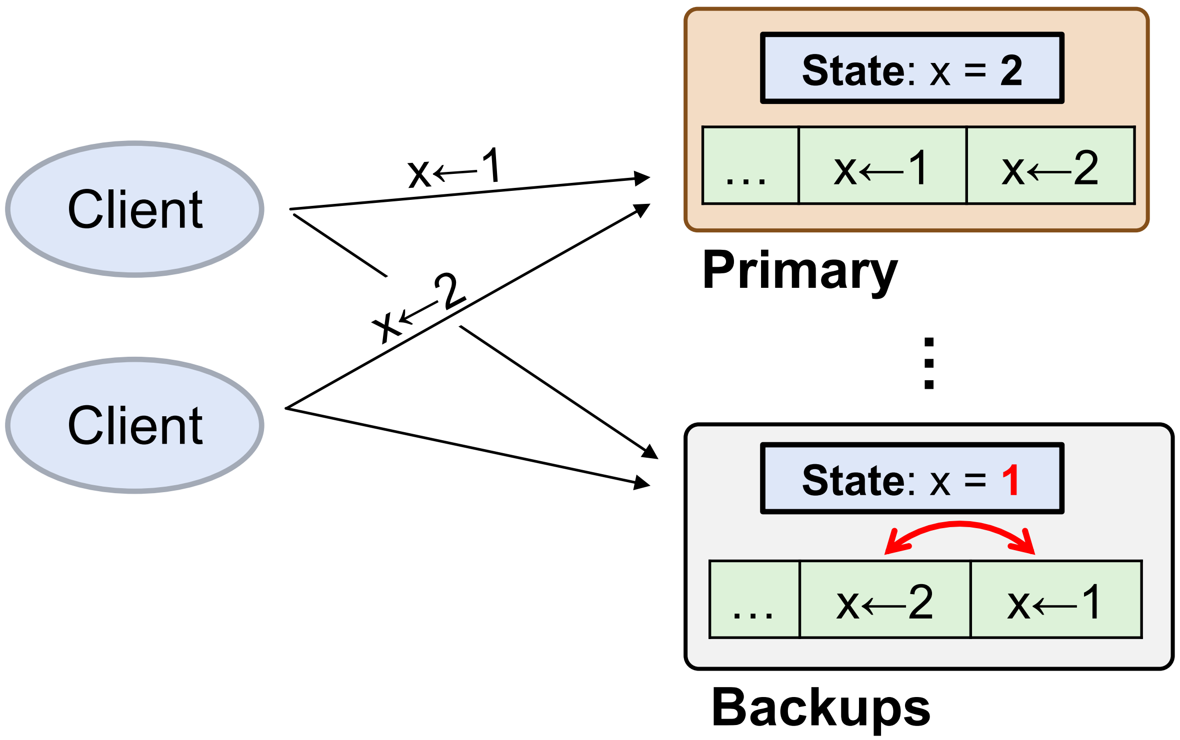


- **Unreplicated Systems: 1 RTT for operation**



- **Replicated Systems: 2 RTTs for operations**

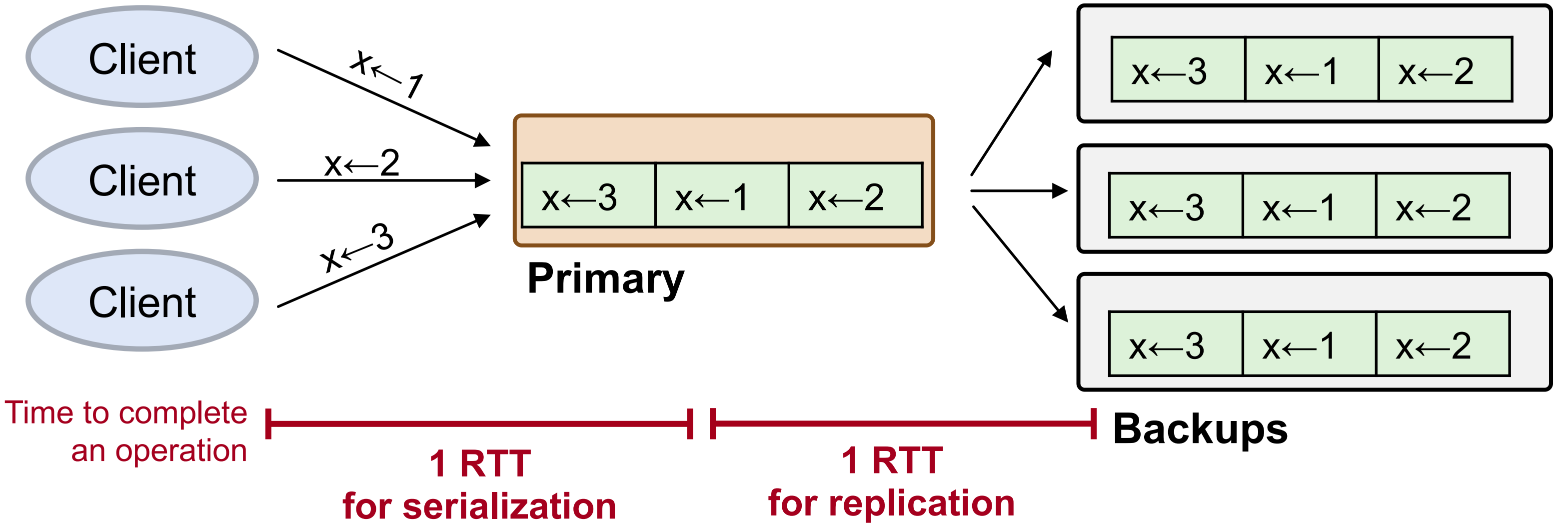
Strawman 1 RTT Replication



Strong consistency is broken!

What Makes Consistent Replication Expensive?

- **Consistent replication protocols must solve two problems:**
  - **Consistent Ordering**: all replicas should appear to execute operations in the same order
  - **Durability**: once completed, an operation must survive crashes.
- **Previous protocols combined the two requirements**

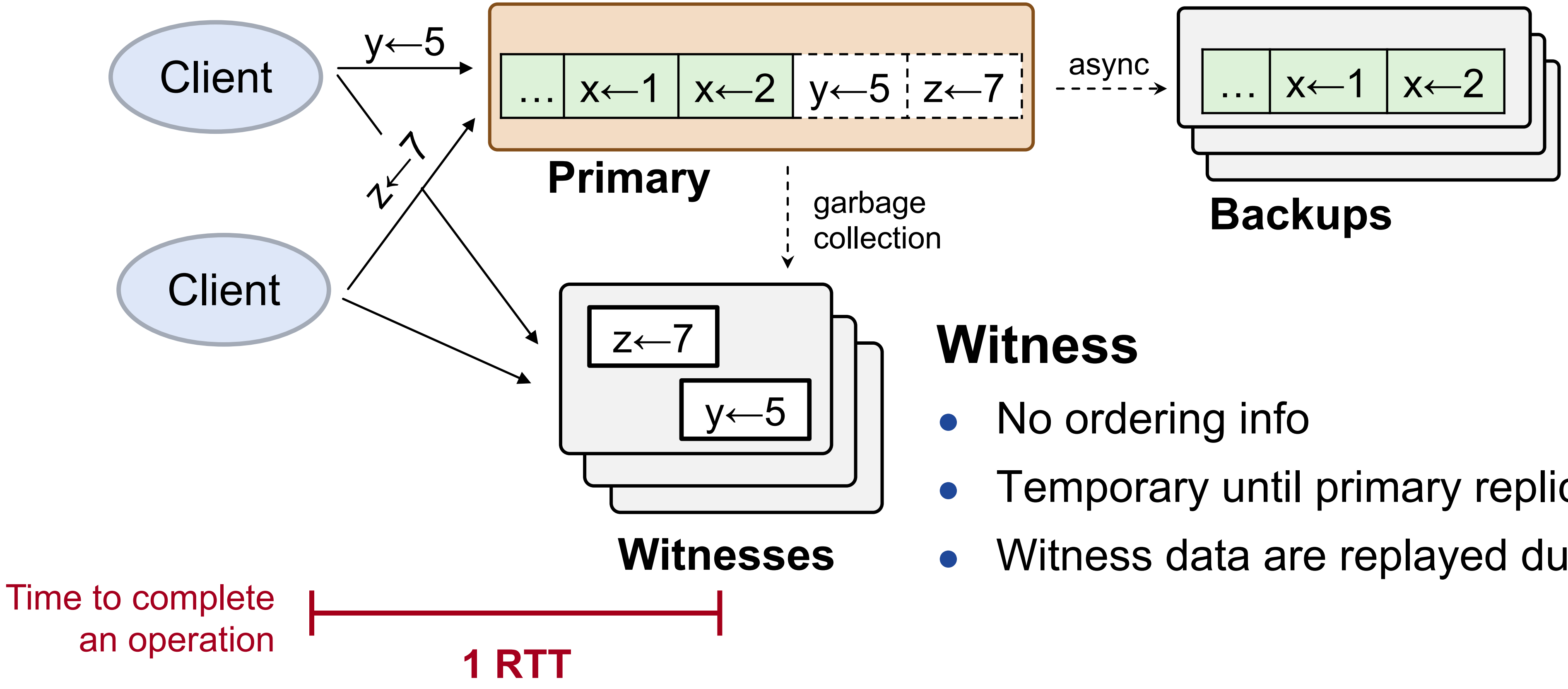


Exploiting Commutativity to Defer Ordering

- **For performance:** cannot do totally ordered replication in 2 RTTs
- Replicate just for **durability** & exploit **commutativity** to **defer ordering**
  - Safe to reorder if operations are *commutative* (e.g. updates on different keys)
- **Consistent Unordered Replication Protocol (CURP):**
  - When concurrent operations commute, replicate without ordering
  - When not, fall back to slow totally-ordered replication

Overview of CURP

- **Primary returns execution results immediately** (before syncing to backups)
- **Clients directly replicate to ensure durability**

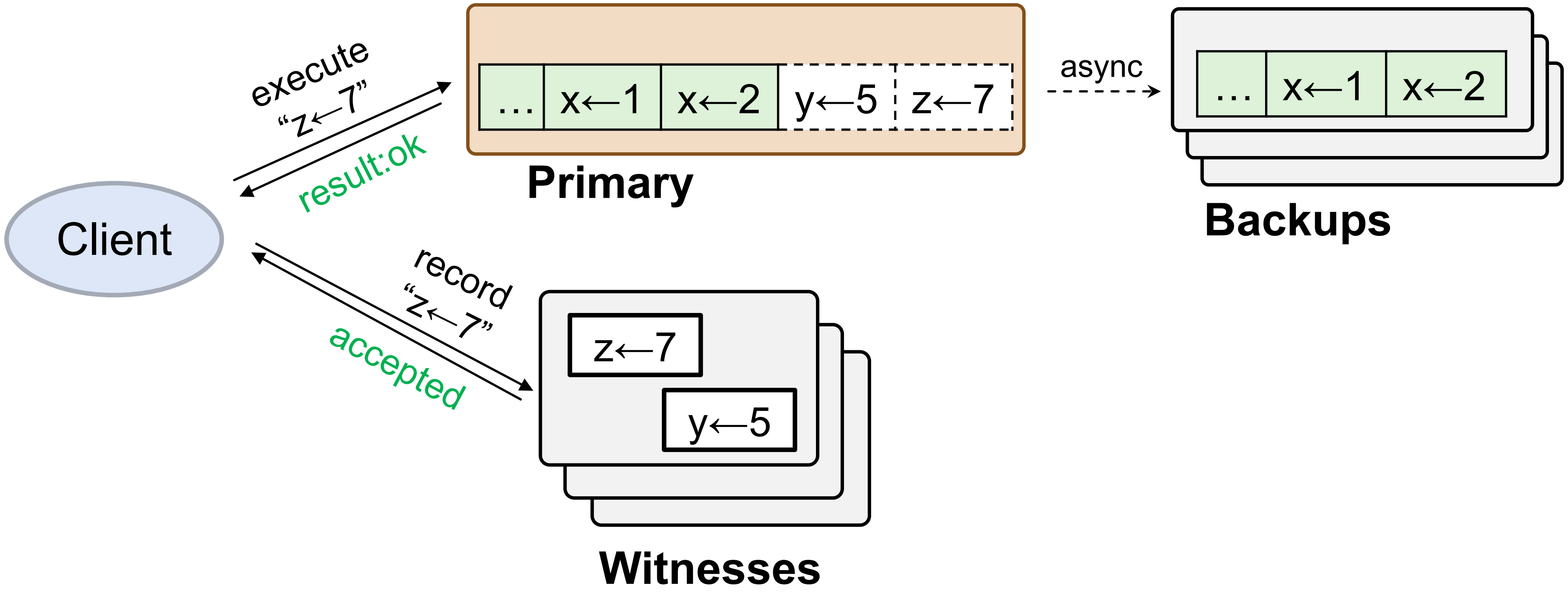


Witness

- No ordering info
- Temporary until primary replicates to backups
- Witness data are replayed during recovery

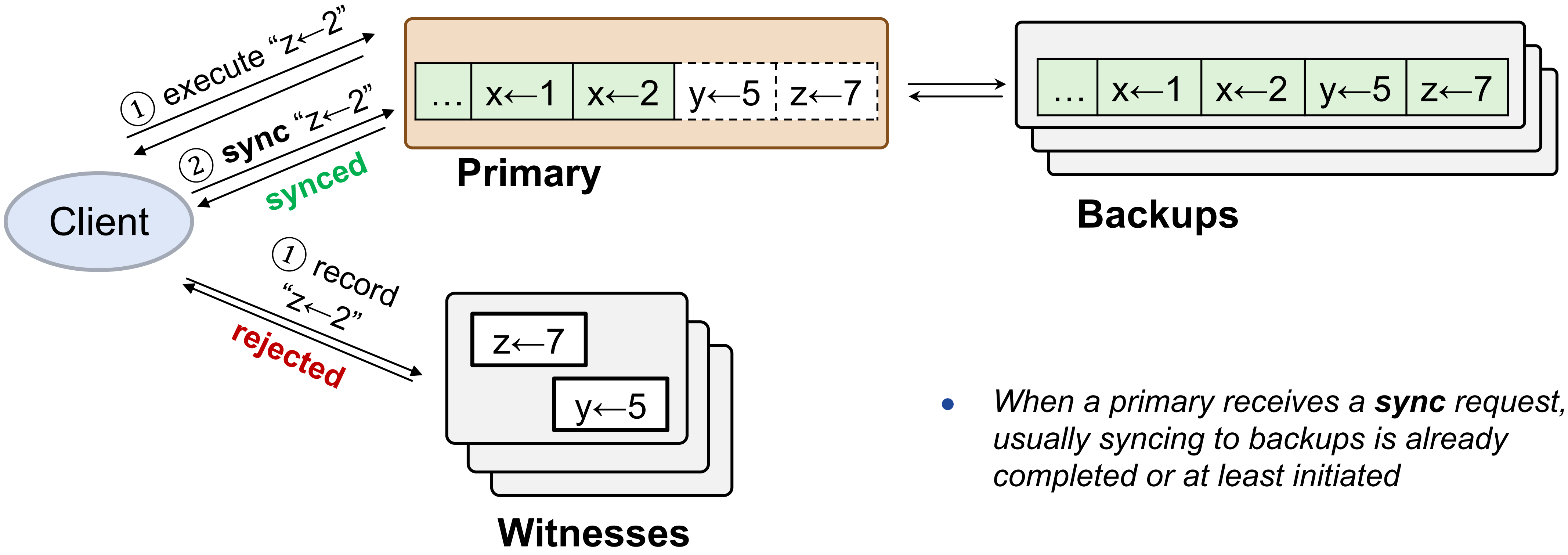
Normal Operation

- Clients send an RPC request to primary and witnesses **in parallel**
- If ***all*** witnesses ***accepted*** (saved) request, client can complete operation safely without sync to backups.



Normal Operation (continued.)

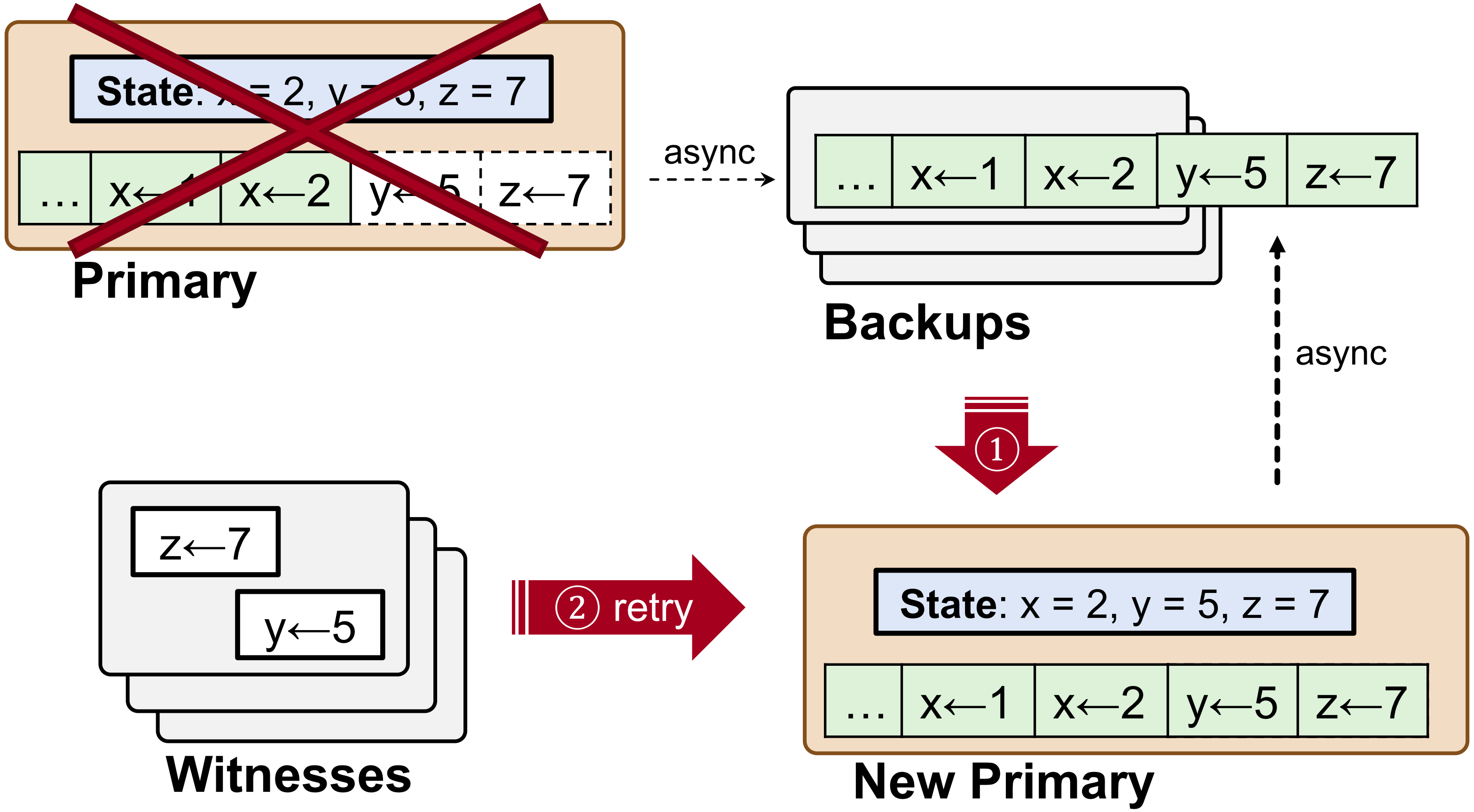
- If ***any*** witness ***rejected*** (not saved) request, client must wait for sync to backups.
  - Operation completes in 2 RTTs mostly (worst case 3 RTTs)



- *When a primary receives a **sync** request, usually syncing to backups is already completed or at least initiated*

Crash Recovery

- First load from **a backup** and then replay requests in **a witness**
- Example:

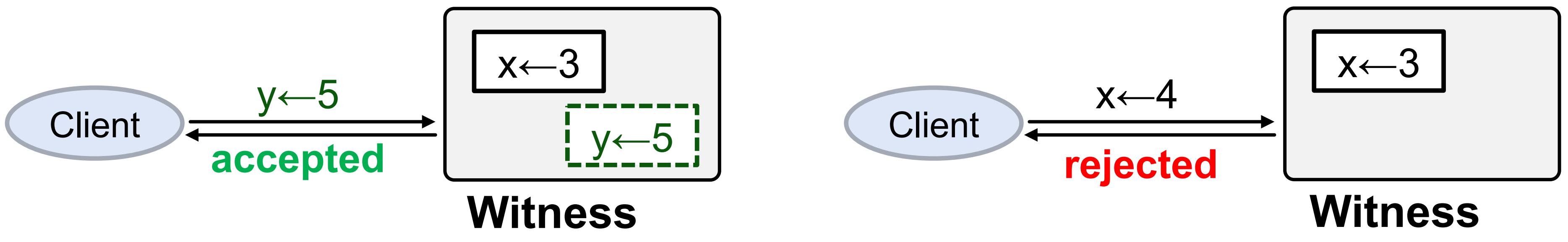


3 Potential Problems for Strong Consistency

1. **Replay from witness may be out of order**
   - Witnesses only keep commutative requests
2. **Primaries may reveal not-yet-durable data to other clients**
   - Primaries detect & block reads of unsynced data
3. **Requests replayed from witness may have been already recovered from backup**
   - Detect and avoid duplicate execution using RIFL

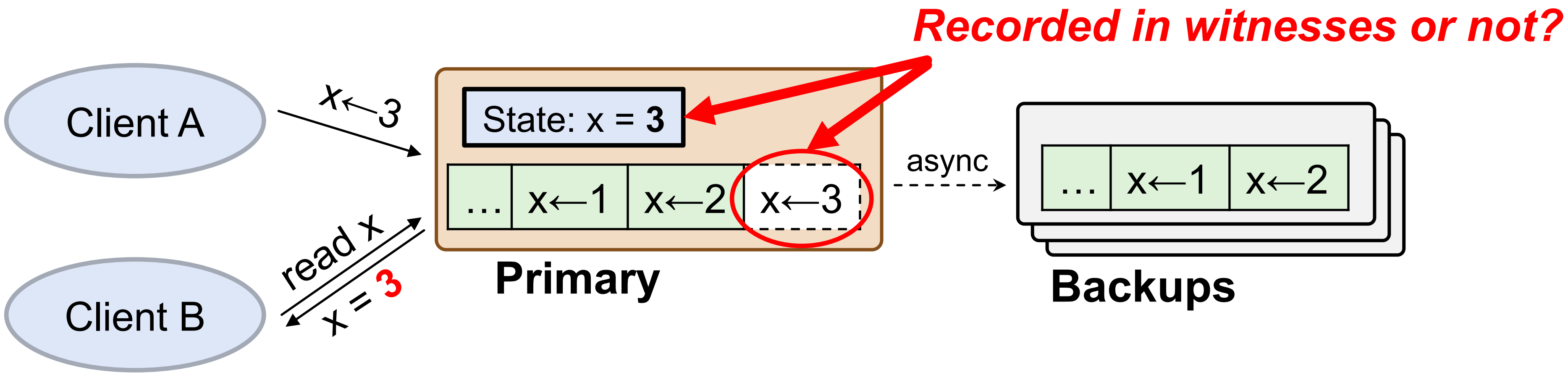
P1. Replay From Witness May Be Out Of Order

- **Witness has no way to know operation order determined by primary**
- **Witness detects non-commutative operations and rejects them**
  - Then, client needs to issue explicit sync request to primary
- **Okay to replay in any order**



P2. Primaries May Reveal Not-yet-durable Data

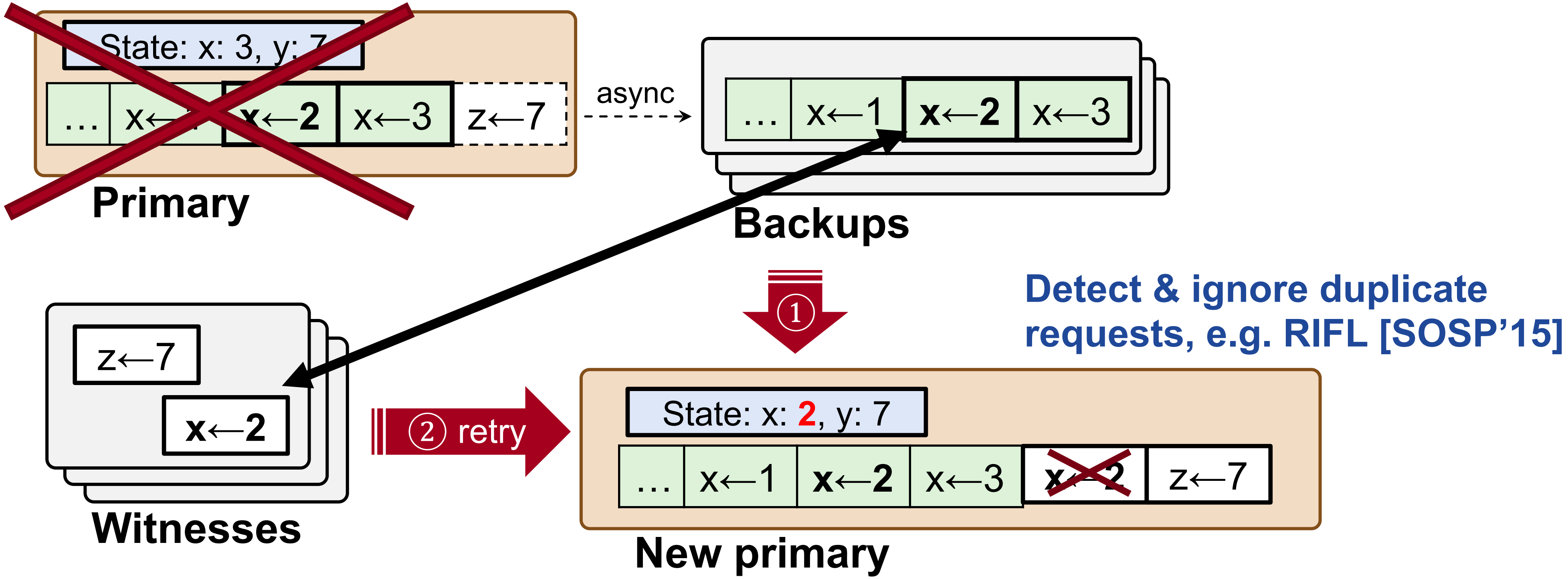
- **Primary doesn't know if an operation is made durable in witnesses**
- **Subsequent operations (e.g. reads) may externalize the new data**



- **Must wait for sync to backups if a client request depends on any unsynced updates**

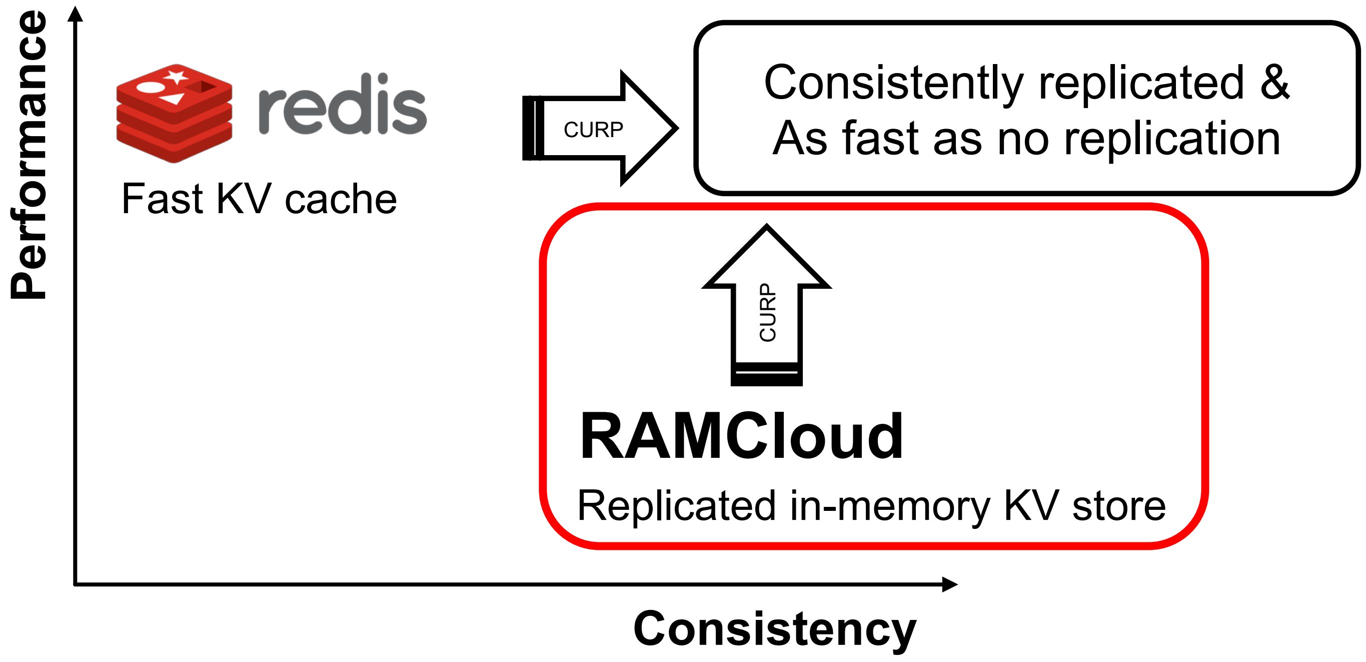
P3. Replay Can Cause Duplicate Executions

- A client request may exist both in backups and witnesses.
- Replaying operations recovered by backups endangers consistency



Performance Evaluation of CURP

- Implemented in Redis and RAMCloud

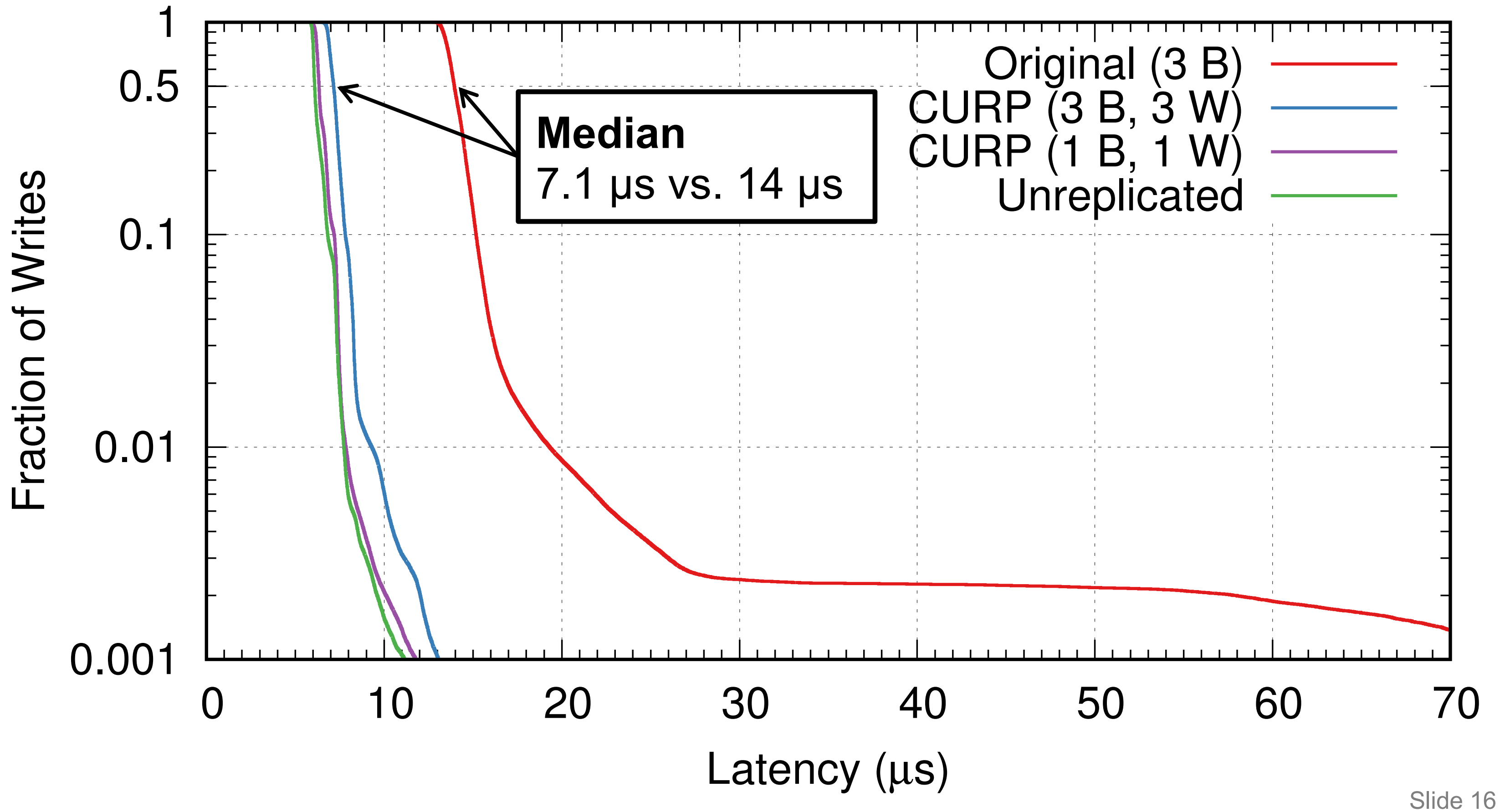


RAMCloud's Latency after CURP

- **Writes are issued sequentially by a client to a master**
  - 40 B key, 100 B value
  - Keys are randomly (uniform dist.) selected from 2 M unique keys

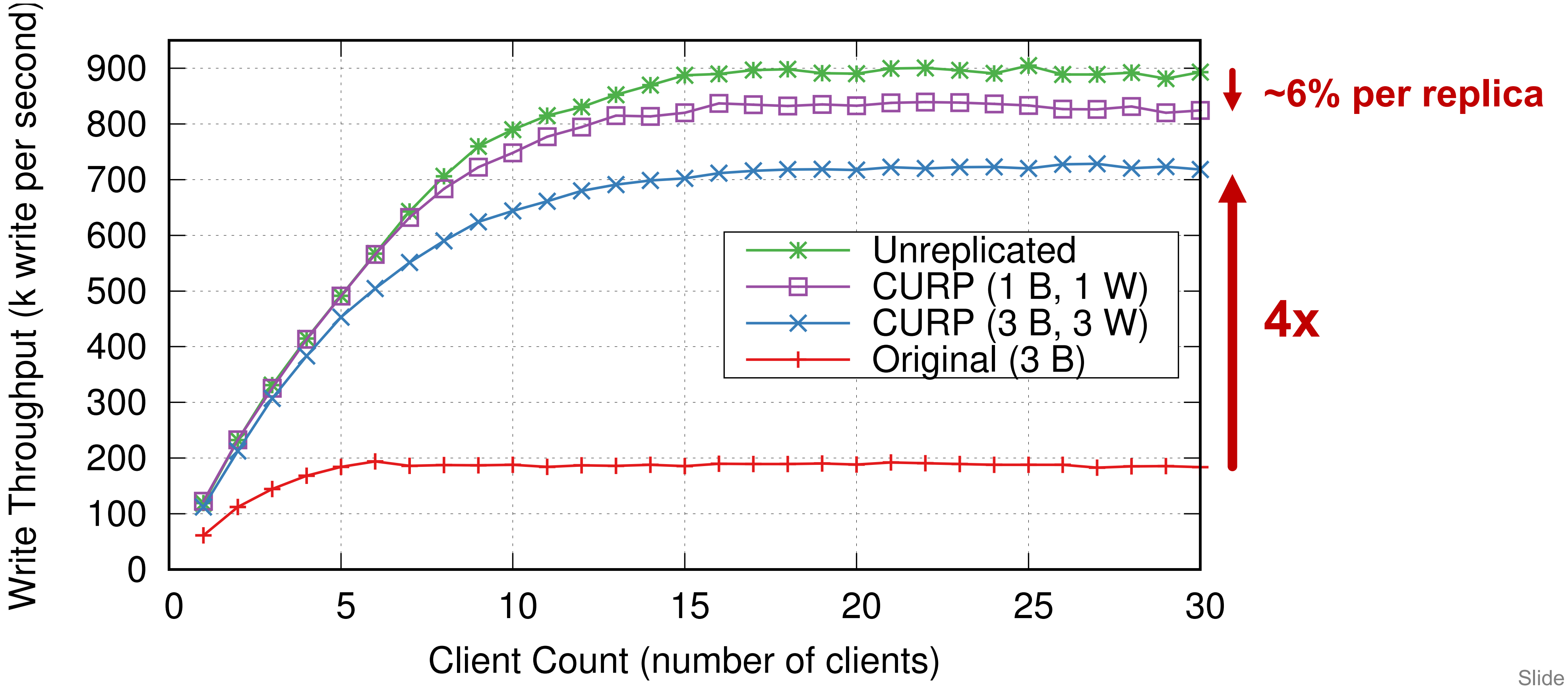
Configuration

- Xeon 4 cores (8 T) @ 3 GHz
- Mellanox Connect-X 2 InfiniBand (24 Gbps)
- Kernel-bypassing transport



RAMCloud's Throughput after CURP

- **Thanks to CURP, can batch replication requests without impacting latency: improves throughput**
- **Each client issues writes (40B key, 100B value) sequentially**



See Paper for...

- **Design**
  - Garbage collection
  - Reconfiguration handling (data migration, backup crash, witness crash)
  - Read operation
  - How to extend CURP to quorum-based consensus protocols
- **Performance**
  - Measurement with skewed workloads (many non-commutative ops)
  - Resource consumption by witness servers
  - CURP’s impact on Redis’ performance

Related work

- **Rely on commutativity for fast replication**
  - Generalized Paxos (2005): 1.5 RTT
  - EPaxos (SOSP'13) : 2 RTTs in LAN, expensive read
- **Rely on the network's in-order deliveries of broadcasts**
  - **Special networking hardware**: NOPaxos (OSDI'16), Speculative Paxos (NSDI'15)
  - **Presume & rollback**: PLATO (SRDS'06), Optimistic Active Replication (ICDCS'01)
  - **Combine with transaction layer for rollback**: TAPIR (SOSP'15), Janus (OSDI'16)
- **CURP is**
  - Faster than other protocols using commutativity
  - Doesn't require rollback or special networking hardware
  - Easy to integrate with existing primary-backup systems

Conclusion

- **Total order is not necessary for consistent replication**
- **CURP clients replicate without ordering in parallel with sending requests to execution servers ➔ 1 RTT**
- **Exploit commutativity for consistency**
- **Improves both latency (2 RTTs -> 1 RTT) and throughput (4x)**
  - RAMCloud's latency: 14 μs → 7.1 μs (no replication: 6.1 μs)
  - Throughput: 184 kops/sec → 728 kops/s (~4x)