usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Exploiting Commutativity For Practical Fast Replication

Seo Jin Park and John Ousterhout, *Stanford University*

https://www.usenix.org/conference/nsdi19/presentation/park

This paper is included in the Proceedings of the
16th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the
16th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '19)
is sponsored by

NetApp®

Exploiting Commutativity For Practical Fast Replication

Seo Jin Park
Stanford University

John Ousterhout
Stanford University

Abstract

Traditional approaches to replication require client requests to be ordered before making them durable by copying them to replicas. As a result, clients must wait for two round-trip times (RTTs) before updates complete. In this paper, we show that this entanglement of ordering and durability is unnecessary for strong consistency. The Consistent Unordered Replication Protocol (CURP) allows clients to replicate requests that have not yet been ordered, as long as they are commutative. This strategy allows most operations to complete in 1 RTT (the same as an unreplicated system). We implemented CURP in the Redis and RAMCloud storage systems. In RAMCloud, CURP improved write latency by ∼2x (14 µs → 7.1 µs) and write throughput by 4x. Compared to unreplicated RAMCloud, CURP's latency overhead for 3-way replication is just 1 µs (6.1 µs vs 7.1 µs). CURP transformed a non-durable Redis cache into a consistent and durable storage system with only a small performance overhead.

1 Introduction

Fault-tolerant systems rely on replication to mask individual failures. To ensure that an operation is durable, it cannot be considered complete until it has been properly replicated. Replication introduces a significant overhead because it requires round-trip communication to one or more additional servers. Within a datacenter, replication can easily double the latency for operations in comparison to an unreplicated system; in geo-replicated environments the cost of replication can be even greater.

In principle, the cost of replication could be reduced or eliminated if replication could be overlapped with the execution of the operation. In practice, however, this is difficult to do. Executing an operation typically establishes an ordering between that operation and other concurrent operations, and the order must survive crashes if the system is to provide consistent behavior. If replication happens in parallel with execution, different replicas may record different orders for the operations, which can result in inconsistent behavior after crashes. As a result, most systems perform ordering before replication: a client first sends an operation to a server that orders the operation (and usually executes it as well); then that server issues replication requests to other servers, ensuring a consistent ordering among replicas. As a result, the minimum latency for an operation is two round-trip times (RTTs). This problem affects all systems that provide consistency and replication, including both primary-backup approaches and consensus approaches.

Consistent Unordered Replication Protocol (CURP) reduces the overhead for replication by taking advantage of the fact that most operations are commutative, so their order of execution doesn't matter. CURP supplements a system's existing replication mechanism with a lightweight form of replication without ordering based on *witnesses*. A client replicates each operation to one or more witnesses in parallel with sending the request to the primary server; the primary can then execute the operation and return to the client without waiting for normal replication, which happens asynchronously. This allows operations to complete in 1 RTT, as long as all witnessed-but-not-yet-replicated operations are commutative. Non-commutative operations still require 2 RTTs. If the primary crashes, information from witnesses is combined with that from the normal replicas to re-create a consistent server state.

CURP can be easily applied to most existing systems using primary-backup replication. Changes required by CURP are not intrusive, and it works with any kind of backup mechanism (e.g. state machine replication [31], file writes to network replicated drives [1], or scattered replication [26]). This is important since most high-performance systems optimize their backup mechanisms, and we don't want to lose those optimizations (e.g. CURP can be used with RAMCloud without sacrificing its fast crash recovery [26]).

To show its performance benefits and applicability, we implemented CURP in two NoSQL storage systems: Redis [30] and RAMCloud [27]. Redis is generally used as a non-durable cache due to its very expensive durability mechanism. By applying CURP to Redis, we were able to provide durability and consistency with similar performance to the non-durable Redis. For RAMCloud, CURP reduced write latency by half (only a 1 µs penalty relative to RAMCloud without replication) and increased throughput by 3.8x without compromising consistency.

Overall, CURP is the first replication protocol that completes linearizable deterministic update operations within 1 RTT without special networking. Instead of relying on special network devices or properties for fast replication [21, 28, 22, 12, 3], CURP exploits commutativity, and it can be used for any system where commutativity of client requests can be checked just from operation parameters (CURP cannot use state-dependent commutativity). Even when compared to Speculative Paxos or NOPaxos (which require a special network topology and special network switches), CURP is faster since client request packets do not need to detour to get ordered by a networking device (NOPaxos has an overhead of 16 µs, but CURP only increased latency by 1 µs).

2 Separating Durability from Ordering

Replication protocols supporting concurrent clients have combined the job of ordering client requests consistently among replicas and the job of ensuring the durability of operations. This entanglement causes update operations to take 2 RTTs.

Replication protocols must typically guarantee the following two properties:

- **Consistent Ordering**: if a replica completes operation *a* before *b*, no client in the system should see the effects of *b* without the effects of *a*.
- **Durability**: once its completion has been externalized to an application, an executed operation must survive crashes.

To achieve both consistent ordering and durability, current replication protocols need 2 RTTs. For example, in master-backup (a.k.a. primary-backup) replication, client requests are always routed to a master replica, which serializes requests from different clients. As part of executing an operation, the master replicates either the client request itself or the result of the execution to backup replicas; then the master responds back to clients. This entire process takes 2 RTTs total: 1 from clients to masters and another RTT for masters to replicate data to backups in parallel.

Consensus protocols with strong leaders (e.g. Multi-Paxos [17] or Raft [25]) also require 2 RTTs for update operations. Clients route their requests to the current leader replica, which serializes the requests into its operation log. To ensure durability and consistent ordering of the client requests, the leader replicates its operation log to a majority of replicas, and then it executes the operation and replies back to clients with the results. In consequence, consensus protocols with strong leaders also require 2 RTTs for updates: 1 RTT from clients to leaders and another RTT for leaders to replicate the operation log to other replicas.

Fast Paxos [19] and Generalized Paxos [18] reduced the latency of replicated updates from 2 RTTs to 1.5 RTT by allowing clients to optimistically replicate requests with presumed ordering. Although their leaders don't serialize client requests by themselves, leaders must still wait for a majority of replicas to durably agree on the ordering of the requests before executing them. This extra waiting adds 0.5 RTT overhead. (See §B.3 for a detailed explanation on why they cannot achieve 1 RTT.)

Network-Ordered Paxos [21] and Speculative Paxos [28] achieve near 1 RTT latency for updates by using special networking to ensure that all replicas receive requests in the same order. However, since they require special networking hardware, it is difficult to deploy them in practice. Also, they can't achieve the minimum possible latency since client requests detour to a common root-layer switch (or a middlebox).

The key idea of CURP is to separate durability and consistent ordering, so update operations can be done in 1 RTT in the normal case. Instead of replicating totally ordered operations in 2 RTTs, CURP achieves durability without ordering and uses the commutativity of operations to defer agreement on operation order.

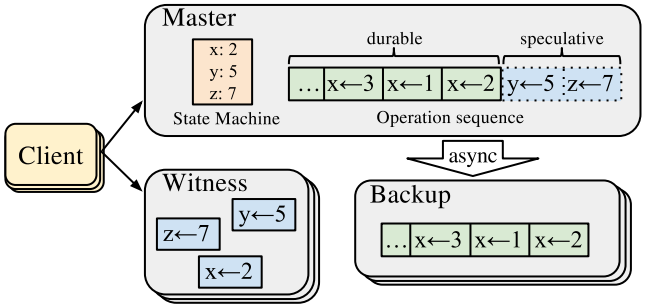

Figure 1: CURP clients directly replicate to witnesses. Witnesses only guarantee durability without ordering. Backups hold data that includes ordering information. Witnesses are temporary storage to ensure durability until operations are replicated to backups.

To achieve durability in 1 RTT, CURP clients directly record their requests in temporary storage, called a *witness*, without serializing them through masters. As shown in Figure 1, witnesses do not carry ordering information, so clients can directly record operations into witnesses in parallel with sending operations to masters so that all requests will finish in 1 RTT. In addition to the unordered replication to witnesses, masters still replicate ordered data to backups, but do so asynchronously after sending the execution results back to the clients. Since clients directly make their operations durable through witnesses, masters can reply to clients as soon as they execute the operations without waiting for permanent replication to backups. If a master crashes, the client requests recorded in witnesses are replayed to recover any operations that were not replicated to backups. A client can then complete an update operation and reveal the result returned from the master if it successfully recorded the request in witnesses (optimistic fast path: 1 RTT), or after waiting for the master to replicate to backups (slow path: 2 RTT).

CURP's approach introduces two threats to consistency: ordering and duplication. The first problem is that the order in which requests are replayed after a server crash may not match the order in which the master processed those requests. CURP uses commutativity to solve this problem: all of the *unsynced* requests (those that a client considers complete, but which have not been replicated to backups) must be commutative. Given this restriction, the order of replay will have no visible impact on system behavior. Specifically, a witness only accepts and saves an operation if it is commutative with every other operation currently stored by that witness (e.g., writes to different objects). In addition, a master will only execute client operations speculatively (by responding before replication is complete), if that operation is commutative with every other unsynced operation. If either a witness or master finds that a new operation is not commutative, the client must ask the master to sync with backups. This adds an extra RTT of latency, but it flushes all of the speculative operations.

The second problem introduced by CURP is duplication.

When a master crashes, it may have completed the replication of one or more operations that are recorded by witnesses. Any completed operations will be re-executed during replay from witnesses. Thus there must be a mechanism to detect and filter out these re-executions. The problem of re-executions is not unique to CURP, and it can happen in distributed systems for a variety of other reasons. There exist mechanisms to filter out duplicate executions, such as RIFL [20], and they can be applied to CURP as well.

We can apply the idea of separating ordering and durability to both consensus-based replicated state machines (RSM) and primary-backup, but this paper focuses on primary-backup since it is more critical for application performance. Fault-tolerant large-scale high-performance systems are mostly configured with a single cluster coordinator replicated by consensus and many data servers using primary-backup (e.g. Chubby [6], ZooKeeper [15], Raft [25] are used for cluster coordinators in GFS [13], HDFS [32], and RAMCloud [27]). The cluster coordinators are used to prevent split-brains for data servers, and operations to the cluster coordinators (e.g. change of master node during recovery) are infrequent and less latency sensitive. On the other hand, operations to data servers (e.g. insert, replace, etc) directly impact application performance, so the rest of this paper will focus on the CURP protocol for primary-backup, which is the main replication technique for data servers. In §B.2, we sketch how the same technique can be applied for consensus.

3 CURP Protocol

CURP is a new replication protocol that allows clients to complete linearizable updates within 1 RTT. Masters in CURP speculatively execute and respond to clients before the replication to backups has completed. To ensure the durability of the speculatively completed updates, clients multicast update operations to witnesses. To preserve linearizability, witnesses and masters enforce commutativity among operations that are not fully replicated to backups.

3.1 Architecture and Model

CURP provides the same guarantee as current primary-backup protocols: it provides linearizability to client requests in spite of failures. CURP assumes a fail-stop model and does not handle byzantine faults. As in typical primary-backup replications, it uses a total of $f+1$ replicas composed of 1 master and f backups, where f is the number of replicas that can fail without loss of availability. In addition to that, it uses f witnesses to ensure durability of updates even before replications to backups are completed. As shown in Figure 2, witnesses may fail independently and may be co-hosted with backups. CURP remains available (i.e. immediately recoverable) despite up to f failures, but will still be strongly consistent even if all replicas fail.

Throughout the paper, we assume that witnesses are separate from backups. This allows CURP to be applied to a wide range of existing replicated systems without modi-

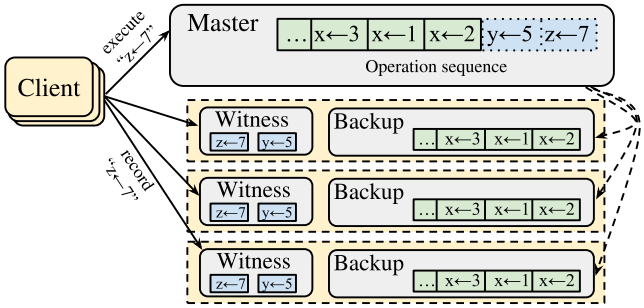

Figure 2: CURP architecture for $f=3$ fault tolerance.

fying their specialized backup mechanisms. For example, CURP can be applied to a system which uses file writes to network replicated drives as a backup mechanism, where the use of witnesses will improve latency while retaining its special backup mechanism. However, when designing new systems, witnesses may be combined with backups for extra performance benefits. (See §B.1 for details.)

CURP makes no assumptions about the network. It operates correctly even with networks that are asynchronous (no bound on message delay) and unreliable (messages can be dropped). Thus, it can achieve 1 RTT updates on replicated systems in any environment, unlike other alternative solutions. (For example, Speculative Paxos [28] and Network-Ordered Paxos [21] require special networking hardware and cannot be used for geo-replication.)

3.2 Normal Operation

3.2.1 Client

Client interaction with masters is generally the same as it would be without CURP. Clients send update RPC requests to masters. If a client cannot receive a response, it retries the update RPC. If the master crashes, the client may retry the RPC with a different server.

For 1 RTT updates, masters return to clients before replication to backups. To ensure durability, clients directly record their requests to *witnesses* concurrently while waiting for responses from masters. Once all f witnesses have accepted the requests, clients are assured that the requests will survive master crashes, so clients complete the operations with the results returned from masters.

If a client cannot record in all f witnesses (due to failures or rejections by witnesses), the client cannot complete an update operation in 1 RTT. To ensure the durability of the operation, the client must wait for replication to backups by sending a **sync** RPC to the master. Upon receiving **sync** RPCs, the master ensures the operation is replicated to backups before returning to the client. This waiting for sync increases the operation latency to 2 RTTs in most cases and up to 3 RTT in the worst case where the master hasn't started syncing until it receives a **sync** RPC from a client. If there is no response to the **sync** RPC (indicating the master might have crashed), the client restarts the entire process; it resends the update RPC to a new master and tries to record the RPC request in witnesses of the new master.

3.2.2 Witness

Witnesses support 3 basic operations: they record operations in response to client requests, hold the operations until explicitly told to drop by masters, and provide the saved operations during recovery.

Once a witness accepts a **record** RPC for an operation, it guarantees the durability of the operation until told that the operation is safe to drop. To be safe from power failures, witnesses store their data in non-volatile memory (such as flash-backed DRAM). This is feasible since a witness needs only a small amount of space to temporarily hold recent client requests. Similar techniques are used in strongly-consistent low-latency storage systems, such as RAMCloud [27].

A witness accepts a new **record** RPC from a client only if the new operation is commutative with all operations that are currently saved in the witness. If the new request doesn't commute with one of the existing requests, the witness must reject the record RPC since the witness has no way to order the two noncommutative operations consistent with the execution order in masters. For example, if a witness already accepted "$x \leftarrow 1$", it cannot accept "$x \leftarrow 5$".

Witnesses must be able to determine whether operations are commutative or not just from the operation parameters. For example, in key-value stores, witnesses can exploit the fact that operations on different keys are commutative. In some cases, it is difficult to determine whether two operations commute each other. SQL UPDATE is an example; it is impossible to determine the commutativity of "`UPDATE T SET rate = 40 WHERE level = 3`" and "`UPDATE T SET rate = rate + 10 WHERE dept = SDE`" just from the requests themselves. To determine the commutativity of the two updates, we must run them with real data. Thus, witnesses cannot be used for operations whose commutativity depends on the system state. In addition to the case explained, determining commutativity can be more subtle for complex systems, such as DBMS with triggers and views.

Each of f witnesses operates independently; witnesses need not agree on either ordering or durability of operations. In an asynchronous network, record RPCs may arrive at witnesses in different order, which can cause witnesses to accept and reject different sets of operations. However, this does not endanger consistency. First, as mentioned in §3.2.1, a client can proceed without waiting for sync to backups only if all f witnesses accepted its record RPCs. Second, requests in each witness are required to be commutative independently, and only one witness is selected and used during recovery (described in §3.3).

3.2.3 Master

The role of masters in CURP is similar to their role in traditional primary-backup replications. Masters in CURP receive, serialize, and execute all update RPC requests from clients. If an executed operation updates the system state, the master synchronizes (*syncs*) its current state with backups by replicating the updated value or the log of ordered operations.

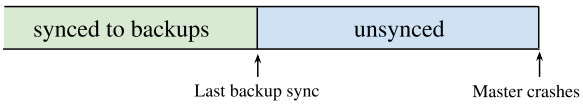

Figure 3: Sequence of executed operations in the crashed master.

Unlike traditional primary-backup replication, masters in CURP generally respond back to clients *before* syncing to backups, so that clients can receive the results of update RPCs within 1 RTT. We call this *speculative* execution since the execution may be lost if masters crash. Also, we call the operations that were speculatively executed but not yet replicated to backups *unsynced* operations. As shown in Figure 3, all unsynced operations are contiguous at the tail of the masters' execution history.

To prevent inconsistency, a master must sync before responding if the operation is not commutative with any existing unsynced operations. If a master responds for a non-commutative operation before syncing, the result returned to the client may become inconsistent if the master crashes. This is because the later operation might complete and its result could be externalized (because it was recorded to witnesses) while the earlier operation might not survive the crash (because, for example, its client crashed before recording it to witnesses). For example, if a master speculatively executes "$x \leftarrow 2$" and "read x", the returned read value, 2, will not be valid if the master crashes and loses "$x \leftarrow 2$". To prevent such unsafe dependencies, masters enforce commutativity among unsynced operations; this ensures that all results returned to clients will be valid as long as they are recorded in witnesses.

If an operation is synced because of a conflict, the master tags its result as "synced" in the response; so, even if the witnesses rejected the operation, the client doesn't need to send a **sync** RPC and can complete the operation in 2 RTTs.

3.3 Recovery

CURP recovers from a master's crash in two phases: (1) restoration from backups and (2) replay from witnesses. First, the new master restores data from one of the backups, using the same mechanism it would have used in the absence of CURP.

Once all data from backups have been restored, the new master replays the requests recorded in witnesses. The new master picks any available witness. If none of the f witnesses are reachable, the new master must wait. After picking the witness to recover from, the new master first asks it to stop accepting more operations; this prevents clients from erroneously completing update operations after recording them in a stale witness whose requests will not be retried anymore. After making the selected witness immutable, the new master retrieves the requests recorded in the witness. Since all requests in a single witness are guaranteed to be commutative, the new master can execute them in any order. After replaying all requests recorded in the selected witness, the new master finalizes the recovery by syncing to backups and resetting witnesses for the new master (or assigning a new set of witnesses). Then the new master can start accepting

client requests again.

Some of the requests in the selected witness may have been executed and replicated to backups before the master crashed, so the replay of such requests will result in re-execution of already executed operations. Duplicate executions of the requests can violate linearizability [20].

To avoid duplicate executions of the requests that are already replicated to backups, CURP relies on exactly-once semantics provided by RIFL [20], which detects already executed client requests and avoids their re-execution. Such mechanisms for exactly-once semantics are already necessary to achieve linearizability for distributed systems [20], so CURP does not introduce a new requirement. In RIFL, clients assign a unique ID to each RPC; servers save the IDs and results of completed requests and use them to detect and answer duplicate requests. The IDs and results are durably preserved with updated objects in an atomic fashion. (If a system replicates client requests to backups instead of just updated values, providing atomic durability becomes trivial since each request already contains its ID and its result can be obtained from its replay during recovery.)

This recovery protocol together with the normal operation protocol described in §3.2 guarantee linearizability of client operations even with server failures. An informal proof of correctness can be found in appendix §A.

3.4 Garbage Collection

To limit memory usage in witnesses and reduce possible rejections due to commutativity violations, witnesses must discard requests as soon as possible. Witnesses can drop the recorded client requests after masters make their outcomes durable in backups. In CURP, masters send garbage collection RPCs for the synced updates to their witnesses. The garbage collection RPCs are batched: each RPC lists several operations that are now durable (using RPC IDs provided by RIFL [20]).

3.5 Reconfigurations

This section discusses three cases of reconfiguration: recovery of a crashed backup, recovery of a crashed witness, and data migration for load balancing. First, CURP doesn't change the way to handle backup failures, so a system can just recover a failed backup as it would without CURP.

Second, if a witness crashes or becomes non-responsive, the system configuration manager (the owner of all cluster configurations) decommissions the crashed witness and assigns a new witness for the master; then it notifies the master of the new witness list. When the master receives the notification, it syncs to backups to ensure f-fault tolerance and responds back to the configuration manager that it is now safe to recover from the new witness. After this point, clients can use f witnesses again to record operations. However, CURP does not push the new list of witnesses to clients. Since clients cache the list of witnesses, clients may still use the decommissioned witness (if it was temporarily disconnected, the witness will continue to accept record RPCs from clients). This endangers consistency since requests recorded in the old witnesses will not be replayed during recovery.

To prevent clients from completing an unsynced update operation with just recording to old witnesses, CURP maintains a monotonically increasing integer, *WitnessListVersion*, for each master. A master's *WitnessListVersion* is incremented every time the witness configuration for the master is updated, and the master is notified of the new version along with the new witness list. Clients obtain the *WitnessListVersion* when they fetch the witness list from the configuration manager. On all update requests, clients include the *WitnessListVersion*, so that masters can detect and return errors if the clients used wrong witnesses; if they receive errors, the clients fetch new witness lists and retry the updates. This ensures that clients' update operations can never complete without syncing to backups or recording to current witnesses.

Third, for load balancing, a master can split its data into two partitions and migrate a partition to a different master. Migrations usually happen in two steps: a prepare step of copying data while servicing requests and a final step which stops servicing (to ensure that all recent operations are copied) and changes configuration. To simplify the protocol changes from the base primary-backup protocol, CURP masters sync to backups and reset witnesses before the final step of migration, so witnesses are completely ruled out of migration protocols. After the migration is completed, some clients may send updates on the migrated partition to the old master and old witnesses; the old master will reject and tell the client to fetch the new master information (this is the same as without CURP); then the client will fetch the new master and its witness information and retry the update. Meanwhile, the requests on the migrated partition can be accidentally recorded in the old witness, but this does not cause safety issues; masters will ignore such requests during the replay phase of recovery by the filtering mechanism used to reject requests on not owned partitions during normal operations.

3.6 Read Operations

CURP handles read operations in a fashion similar to that of primary-backup replication. Since such operations don't modify system state, clients can directly read from masters, and neither clients nor masters replicate read-only operations to witnesses or backups.

However, even for read operations, a master must check whether a read operation commutes with all currently *unsynced* operations as discussed in §3.2.3. If the read operation conflicts with some unsynced update operations, the master must sync the unsynced updates to backups before responding for the read.

3.7 Consistent Reads from Backups

In primary-backup replication, clients normally issue all read operations to the master. However, some systems allow reading from backups because it reduces the load on masters and can provide better latency in a geo-replicated environment (clients can read from a backup in the same

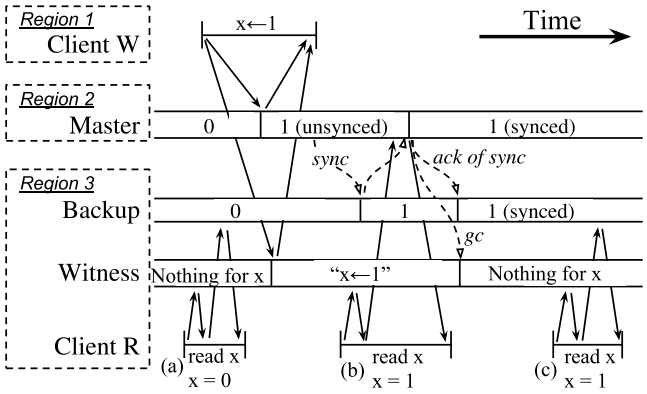

Figure 4: Three cases of reading the value of x from a backup replica while another client is changing the value of x from 0 to 1: (a) client R first confirms that a nearby witness has no request that is not commutative with "read x," so the client directly reads the value of x from a nearby backup. (b) Just after client W completes "x ← 1", client R starts another read. Client R finds that there is a non-commutative request saved in a nearby witness, so it must read from a remote master to guarantee consistency. (c) After syncing "x ← 1" to the backup, the master garbage collected the update request from witnesses and acknowledged the full sync to backups. Now, client R sees no non-commutative requests in the witness and can complete read operation by reading from the nearby backup.

region to avoid wide-area RTTs). However, naively reading from backups can violate linearizability since updates in CURP can complete before syncing to backups.

To avoid reading stale values, clients in CURP use a nearby witness (possibly colocated with a backup) to check whether the value read from a nearby backup is up to date. To perform a consistent read, a client must first ask a witness whether the read operation commutes with the operations currently saved in the witness (as shown in Figure 4). If it commutes, the client is assured that the value read from a backup will be up to date. If it doesn't commute (i.e. the witness retains a write request on the key being read), the value read from a backup might be stale. In this case, the client must read from the master.

In addition, we assume that the underlying primary-backup replication mechanism prevents backups from returning new values that are not yet fully synced to all backups. Such mechanism is neccessary even before applying CURP since returning a new value prematurely can cause inconsistency; even if a value is replicated to some of backups, the value may get lost if the master crashes and a new master recovers from a backup that didn't receive the new value. A simple solution for this problem is that backups don't allow reading values that are not yet fully replicated to all backups. For backups to track which values are fully replicated and ok to be read, a master can piggyback the acknowlegements for successful previous syncs when it sends **sync** requests to backups. When a client tries to read a value that is not known to be yet fully replicated, the backup can wait for full replication or ask the client to retry.

Thanks to the safety mechanisms discussed above, CURP still guarantees linearizability. With a concurrent update, reading from backups could violate linearizability in two ways: (1) a read sees the old value after the completion of the update operation and (2) a read sees the old value after another read returned the new value. The first issue is prevented by checking a witness before reading from a backup. Since clients can complete an update operation only if it is synced to *all* backups or recorded in *all* witnesses, a reader will either see a noncommutative update request in the witness being checked or find the new value from the backup; thus, it is impossible for a read after an update to return the old value. For the second issue, since both a master and backups delay reads of a new value until it is fully replicated to all backups, it is impossible to read an older value after another client reads the new value.

4 Implementation on NoSQL Storage

This section describes how to implement CURP on low-latency NoSQL storage systems that use primary-backup replications. With the emergence of large-scale Web services, NoSQL storage systems became very popular (e.g. Redis [30], RAMCloud [27], DynamoDB [33] and MongoDB [7]), and they range from simple key-value stores to more fully featured stores supporting secondary indexing and multi-object transactions; so, improving their performance using CURP is an important problem with a broad impact.

The most important piece missing from §3 to implement CURP is how to efficiently detect commutativity violations. Fortunately for NoSQL systems, CURP can use *primary keys* to efficiently check the commutativity of operations. NoSQL systems store data as a collection of objects, which are identified by *primary key*s. Most update operations in NoSQL specify the affected object with its primary key (or a list of primary keys), and update operations are commutative if they modify disjoint sets of objects. The rest of this section describes an implementation of CURP that exploits this efficient commutativity check.

4.1 Life of A Witness

Witnesses have two modes of operation: normal and recovery. In each mode, witnesses service a subset of operations listed in Figure 5. When it receives a **start** RPC, a witness starts its life for a master in normal mode, in which the witness is allowed to mutate its collection of saved requests. In normal mode, the witness services **record** RPCs for client requests targeted to the master for which the witness was configured by **start**; by accepting only requests for the correct master, CURP prevents clients from recording to incorrect witnesses. Also, witnesses drop their saved client requests as they receive **gc** RPCs from masters.

A witness irreversibly switches to a recovery mode once it receives a **getRecoveryData** RPC. In recovery mode, mutations on the saved requests are prohibited; witnesses reject all **record** RPCs and only service **getRecoveryData** or **end**. As a recovery is completed and the witness becomes useless, the cluster coordinator may send **end** to free up the resources, so that the witness server can start another life for a different master.

CLIENT TO WITNESS:
record(*masterID*, list of *keyHash*, *rpcId*, *request*) → {ACCEPTED or REJECTED}
Saves the client *request* (with *rpcId*) of an update on *keyHash*es. Returns whether the witness could accomodate and save the request.

MASTER TO WITNESS:
gc(list of {*keyHash, rpcId*}) → list of *request*
Drops the saved requests with the given *keyHash*es and *rpcId*s. Returns stale requests that haven't been garbage collected for a long time.
getRecoveryData() → list of *request*
Returns all requests saved for a particular crashed master.

CLUSTER COORDINATOR TO WITNESS:
start(*masterId*) → {SUCCESS or FAIL}
Start a witness instance for the given master, and return SUCCESS. If the server fails to create the instance, FAIL is returned.
end() → *NULL*
This witness is decommissioned. Destruct itself.

Figure 5: The APIs of Witnesses.

4.2 Data Structure of Witnesses

Witnesses are designed to minimize the CPU cycles spent for handling **record** RPCs. For client requests mutating a single object, recording to a witness is similar to inserting in a set-associative cache; a record operation finds a set of slots using a hash of the object's primary key and writes the given request to an available slot in the set. To enforce commutativity, the witness searches the occupied slots in the set and rejects if there is another request with the same primary key (for performance, we compare 64-bit hashes of primary keys instead of full keys). If there is no slot available in the set for the key, the record operation is rejected as well.

For client requests mutating multiple objects, witnesses perform the commutativity and space check for every affected object; to accept an update affecting n objects, a witness must ensure that (1) no existing client request mutates any of the n objects and (2) there is an available slot in each set for all n objects. If the update is commutative and space is available, the witness writes the update request n times as if recording n different requests on each object.

4.3 Commutativity Checks in Masters

Every NoSQL update operation changes the values of one or more objects. To enforce commutativity, a master can check if the objects touched (either updated or just read) by an operation are *unsynced* at the time of its execution. If an operation touches any *unsynced* value, it is not commutative and the master must sync all unsynced operations to backups before responding back to the client.

If the object values are stored in a log, masters can determine if an object value is synced or not by comparing its position in the log against the last synced position.

If the object values are not stored in a log, masters can use monotonically increasing timestamps. Whenever a master updates the value of an object, it tags the new value with a current timestamp. Also, the master keeps the timestamp of when last backup sync started. By comparing the timestamp of an object against the timestamp of the last backup sync, a master can tell whether the value of the object has been synced to backups.

4.4 Improving Throughput of Masters

Masters in primary-backup replication are usually the bottlenecks of systems since they drive replication to backups. Since masters in CURP can respond to clients before syncing to backups, they can delay syncs until the next batch without impacting latency. This batching of syncs improves masters' throughput in two ways.

First, by batching replication RPCs, CURP reduces the number of RPCs a master must handle per client request. With 3-way primary-backup replication, a master must process 4 RPCs per client request (1 update RPC and 3 replication RPCs). If the master batches replication and syncs every 10 client requests, it handles 1.3 RPCs on average. On NoSQL storage systems, sending and receiving RPCs takes a significant portion of the total processing time since NoSQL operations are not compute-heavy.

Second, CURP eliminates wasted resources and other inefficiencies that arise when masters wait for syncs. For example, in the RAMCloud [27] storage system, request handlers use a polling loop to wait for completion of backup syncs. The syncs complete too quickly to context-switch to a different activity, but the polling still wastes more than half of the CPU cycles of the polling thread. With CURP, a master can complete a request without waiting for syncing and move on to the next request immediately, which results in higher throughput.

The batch size of syncs is limited in CURP to reduce witness rejections. Delaying syncs increases the chance of finding non-commutative operations in witnesses and masters, causing extra rejections in witnesses and more blocking syncs in masters. A simple way to limit the batching would be for masters to issue a sync immediately after responding to a client if there is no outstanding sync; this strategy gives a reasonable throughput improvement since at most one CPU core will be used for syncing, and it also reduces witness rejections by syncing aggresively. However, to find the optimal batch size, an experiment with a system and real workload is necessary since each workload has a different sensitivity to larger batch sizes. For example, workloads which randomly access large numbers of keys uniformly can use a very large batch size without increasing the chance of commutativity conflicts.

4.5 Garbage Collection

As discussed in §3.4, masters send garbage collection RPCs for synced updates to their witnesses. Right after syncing to backups, masters send **gc** RPCs (in Figure 5), so the witnesses can discard data for the operations that were just synced.

To identify client requests for removal, CURP uses 64-bit key hashes and RPC IDs assigned by RIFL [20]. Upon receiving a **gc** RPC, a witness locates the sets of slots using the *keyHash*es and resets the slots whose occupying requests have the matching RPC IDs. Witnesses ignore *keyHash*es and *rpcIds* that are not found since the record RPCs might have been rejected. For client requests that mutate multiple objects, **gc** RPCs include multiple ⟨*keyHash*, *rpcIds*⟩ pairs for all affected objects, so that witnesses can clear all slots

occupied by the request.

Although the described garbage collection can clean up most records, some slots may be left uncollected: if a client crashes before sending the update request to the master, or if the **record** RPC is delayed significantly and arrives after the master finished garbage collection for the update. Uncollected garbage will cause witnesses to indefinitely reject requests with the same keys.

Witnesses detect such uncollected records and ask masters to retry garbage collection for them. When it rejects a **record**, a witness recognizes the existing record as uncollected garbage if there have been many garbage collections since the record was written (three is a good number if a master performs only one **gc** RPC at a time). Witnesses notify masters of the requests that are suspected as uncollected garbage through the response messages of **gc** RPCs; then the masters retry the requests (most likely filtered by RIFL), sync to backups, and thus include them in the next **gc** requests.

4.6 Recovery Steps

To recover a crashed master, CURP first restores data from backups and then replays requests from a witness. To fetch the requests to replay, the new master sends a **getRecoveryData** RPC (in Figure 5), which has two effects: (1) it irreversibly sets the witness into recovery mode, so that the data in the witness will never change, (2) it provides the entire list of client requests saved in the witness.

With the provided requests, the new master replays all of them. Since operations already recovered from backups will be filtered out by RIFL [20], the replay step finishes very quickly. In total, CURP increases recovery time by the execution time for a few requests plus 2 RTT (1 RTT for **getRecoveryData** and another RTT for backup sync after replay).

4.7 Zombies

For a fault-tolerant system to be consistent, it must neutralize *zombies*. A zombie is a server that has been determined to have crashed, so some other server has taken over its functions, but the server has not actually crashed (e.g., it may have suffered temporary network connectivity problems). Clients may continue to communicate with zombies; reads or updates accepted by a zombie may be inconsistent with the state of the replacement server.

CURP assumes that the underlying system already has mechanisms to neutralize zombies (e.g., by asking backups to reject replication requests from a crashed master [27]). The witness mechanism provides additional safeguards. If a zombie responds to a client request without waiting for replication, then the client must communicate with all witnesses before completing the request. If it succeeds before the witness data has been replayed during recovery, then the update will be reflected in the new master. If the client contacts a witness after its data has been replayed, the witness will reject the request; the client will then discover that the old master has crashed and reissue its request to the new master. Thus, the witness mechanism does not create new safety issues with respect to zombies.

	RAMCloud cluster	Redis cluster
CPU	Xeon X3470 (4x2.93 GHz)	Xeon D-1548 (8x2.0 GHz)
RAM	24 GB DDR3 at 800 MHz	64 GB DDR4
Flash	2x Samsung 850 PRO SSDs	Toshiba NVMe flash
NIC	Mellanox ConnectX-2 InfiniBand HCA (PCIe 2.0)	Mellanox ConnectX-3 10 Gbps NIC (PCIe 3.0)
Switch	Mellanox SX6036 (2 level)	HPE 45XGc
OS	Linux 3.16.0-4-amd64	Linux 3.13.0-100-generic

Table 1: The server hardware configuration for benchmarks.

4.8 Modifications to RIFL

In order to work with CURP, the garbage collection mechanism of RIFL described in [20] must be modified. See §C.1 for details.

5 Evaluation

We evaluated CURP by implementing it in the RAMCloud and Redis storage systems, which have very different backup mechanisms. First, using the RAMCloud implementation, we show that CURP improves the performance of consistently replicated systems. Second, with the Redis implementation, we demonstrate that CURP can make strong consistency affordable in a system where it had previously been too expensive for practical use.

5.1 RAMCloud Performance Improvements

RAMCloud [27] is a large-scale low latency distributed key-value store, which primarily focuses on reducing latency. Small read operations take 5 µs, and small writes take 14 µs. By default, RAMCloud replicates each new write to 3 backups, which asynchronously flush data into local drives. Although replicated data are stored in slow disk (for cost saving), RAMCloud features a technique to allow fast recovery from a master crash (it recovers within a few seconds) [26].

With the RAMCloud implementation of CURP, we answered the following questions:

- How does CURP improve RAMCloud's latency and throughput?
- How many resources do witness servers consume?
- Will CURP be performant under highly-skewed workloads with hot keys?

Our evaluations using the RAMCloud implementation were conducted on a cluster of machines with the specifications shown in Table 1. All measurements used InfiniBand networking and RAMCloud's fastest transport, which bypasses the kernel and communicates directly with InfiniBand NICs. Our CURP implementation kept RAMCloud's fast crash recovery [26], which recovers from master crashes within a few seconds using data stored on backup disks. Servers were configured to replicate data to 1–3 different backups (and 1–3 witnesses for CURP results), indicated as a replication factor f. The log cleaner of RAMCloud did not run in any measurements; in a production system, the log cleaner can reduce the throughput.

For RAMCloud, CURP moved backup syncs out of the critical path of write operations. This decoupling not only improved latency but also improved the throughput of

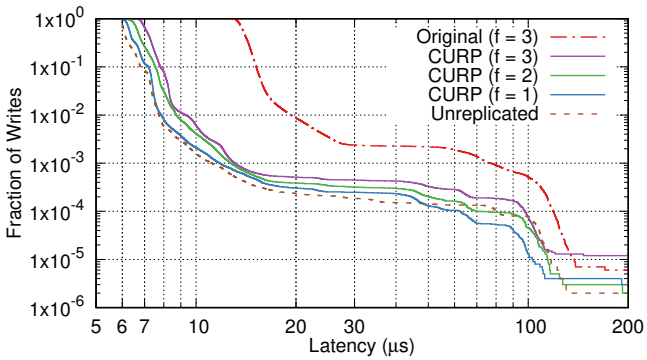

Figure 6: Complementary cumulative distribution of latency for 100B random RAMCloud writes with CURP. Writes were issued sequentially by a single client to a single server, which batches 50 writes between syncs. A point (x, y) indicates that y of the 1M measured writes took at least x µs to complete. f refers to fault tolerance level (i.e. number of backups and witnesses). "Original" refers to the base RAMCloud system before adopting CURP. "Unreplicated" refers to RAMCloud without any replication. The median latency for synchronous, CURP ($f = 3$), and unreplicated writes were 14 µs, 7.1 µs, and 6.1 µs respectively.

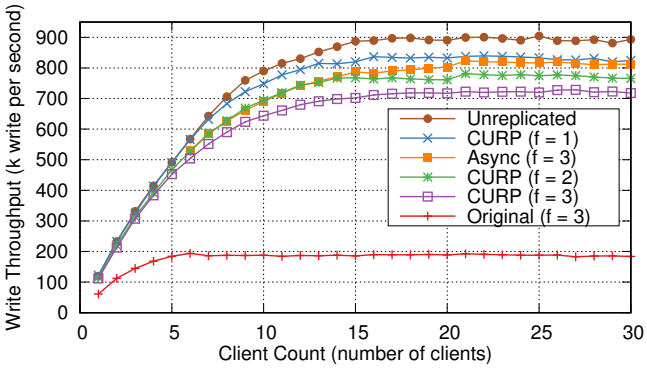

Figure 7: The aggregate throughput for one server serving 100B RAMCloud writes with CURP, as a function of the number of clients. Each client repeatedly issued random writes back to back to a single server, which batches 50 writes before syncs. Each experiment was run 15 times, and median values are displayed. "Original" refers to the base RAMCloud system before adding CURP. "Unreplicated" refers to RAMCloud without any replication. In "Async" RAMCloud, masters return to clients before backup syncs, and clients complete writes without replication to witnesses or backups.

RAMCloud writes.

Figure 6 shows the latency of RAMCloud write operations before and after applying CURP. CURP cuts the median write latencies in half. Even the tail latencies are improved overall. When compared to unreplicated RAMCloud, each additional replica with CURP adds 0.3 µs to median latency.

Figure 7 shows the single server throughput of write operations with and without CURP by varying the number of clients. The server batches 50 writes before starting a sync. By batching backup syncs, CURP improves throughput by about 4x. When compared to unreplicated RAMCloud, adding an additional CURP replica drops throughput by ~6%.

To illustrate the overhead of CURP on throughput (e.g. sending gc RPCs to witnesses), we measured RAMCloud with asynchronous replication to 3 backups, which is identical to CURP (f=3) except that it does not record information on witnesses. Achieving strong consistency with CURP reduces

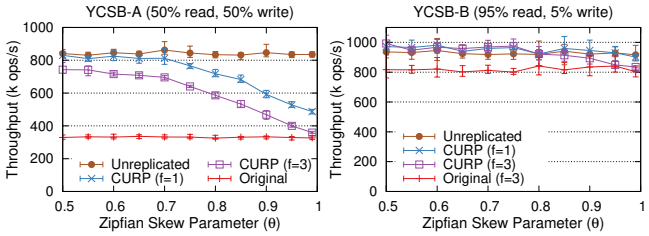

Figure 8: Throughput of a single RAMCloud server for YCSB-A and YCSB-B workloads with CURP at different Zipfian skewness levels. Each experiment was run 5 times, and median values are displayed with errorlines for min and max.

throughput by 10%. In all configurations except the original RAMCloud, masters are bottlenecked by a dispatch thread which handles network communications for both incoming and outgoing RPCs. Sending witness gc RPCs burdens the already bottlenecked dispatch thread and reduces throughput.

We also measured the latency and throughput of RAMCloud read operations before and after applying CURP, and there were no differences.

5.2 Resource Consumption by Witness Servers

Each witness server implemented in RAMCloud can handle 1270k record requests per second with occasional garbage collection requests (1 every 50 writes) from master servers. A witness server runs on a single thread and consumes 1 hyper-thread core at max throughput. Considering that each RAMCloud master server uses 8 hyper-thread cores to achieve 728k writes per second, adding 1 witness increases the total CPU resources consumed by RAMCloud by 7%. However, CURP reduces the number of distinct backup operations performed by masters, because it enables batching; this offsets most of the cost of the witness requests (both backup and witness operations are so simple that most of their cost is the fixed cost of handling an RPC; a batched replication request costs about the same as a simple one).

The second resource overhead is memory usage. Each witness server allocates 4096 request storage slots for each associated master, and each storage slot is 2KB. With additional metadata, the total memory overhead per master-witness pair is around 9MB.

The third issue is network traffic amplification. In CURP, each update request is replicated both to witnesses and backups. With 3-way replication, CURP increases network bandwidth use for update operations by 75% (in the original RAMCloud, a client request is transferred over the network to a master and 3 backups).

5.3 Impact of Highly-Skewed Workloads

CURP may lose its performance benefits when used with highly-skewed workloads with hot keys; in CURP, an unsynced update on a key causes conflicts on all following updates or reads on the same key until the sync completes. To measure the impact of hot keys, we measured RAMCloud's performance with CURP using a highly-skewed Zipfian distribution [14] with 1M objects. Specifically, we used two different workloads similar to YCSB-A and YCSB-B [9];

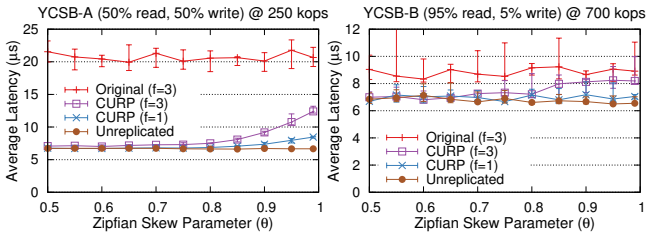

Figure 9: Average RAMCloud client request latency for YCSB-A and YCSB-B workloads with CURP at different Zipfian skewness levels. 10 clients issued requests to maintain a certain throughput level (250 kops for YCSB-A and 700 kops for YCSB-B). Each experiment was run 5 times, and median values are displayed with errorlines for min and max. Latency values are averaged over both read and write operations.

since RAMCloud is a key-value store and doesn't support 100B field writes in 1k objects, we modified the YCSB benchmark to read and write 100B objects with 30B keys.

Figure 8 shows the impact of workload skew (defined in [14]) on the throughput of a single server. For YCSB-A (write-heavy workload), the server throughput with CURP is similar to an unreplicated server when skew is low, but it drops as the workload gets more heavily skewed. For YCSB-B, since most operations are reads, the throughput is less affected by skew. CURP's throughput benefit degrades starting at a Zipfian parameter $\theta = 0.8$ (about 3% of accesses are on hot keys) and almost disappears at $\theta = 0.99$.

Figure 9 shows the impact of skew on CURP's latency; unlike the throughput benefits, CURP retains its latency benefits even with extremely skewed workloads. We measured latencies under load since an unloaded system will not experience conflicts even with extremely skewed workloads. For YCSB-A, the latency of CURP increases starting at $\theta = 0.85$, but CURP still reduces latency by 42% even at $\theta = 0.99$. For YCSB-B, only 5% of operations are writes, so the latency improvements are not as dramatic as YCSB-A.

Figure 10 shows the latency distributions of reads and writes separately at $\theta = 0.95$ under the same loaded conditions as Figure 9. For YCSB-A, CURP increases the tail latency for read operations slightly since reads occasionally conflict with unsynced writes on the same keys. CURP reduces write latency by 2–4x: write latency with CURP is almost as low as for unreplicated writes until the 50th percentile, where conflicts begins to cause blocking on syncs. Overall, the improvement of write latency by CURP more than compensates for the degradation of read latency.

For YCSB-B, operation conflicts are more rare since all reads (which compose 95% of all operations) are commutative with each other. In this workload, CURP actually improved the overall read latency; this is because, by batching replication, CURP makes CPU cores more readily available for incoming read requests (which is also why unreplicated reads have lower latency). For YCSB-A, CURP doesn't improve read latency much since frequent conflicts limit batching replication. In general, read-heavy workloads experience fewer conflicts and are less affected by hot keys.

5.4 Making Redis Consistent and Durable

Redis [30] is another low-latency in-memory key-value store, where values are data structures, such as lists, sets, etc. For Redis, the only way to achieve durability and consistency after crashes is to log client requests to an append-only file and invoke fsync before responding to clients. However, fsyncs can take several milliseconds, which is a 10–100x performance penalty. As a result, most Redis applications do not use synchronous mode; they use Redis as a cache with no durability guarantees. Redis also offers replication to multiple servers, but the replication mechanism is asynchronous, so updates can be lost after crashes; as a result, this feature is not widely used either.

For this experiment, we used CURP to hide the cost of Redis' logging mechanism: we modified Redis to record operations on witnesses, so that operations can return without waiting for log syncs. Log data is then written asynchronously in the background. The result is a system with durability and consistency, but with performance equivalent to a system lacking both of these properties. In this experiment the log data is not replicated, but the same mechanism could be used to replicate the log data as well.

With the Redis implementation of CURP, we answered the following questions:

- Can CURP transform a fast in-memory cache into a strongly-consistent durable storage system without degrading performance?
- How wide a range of operations can CURP support?

Measurements of the Redis implementation were conducted on a cluster of machines in CloudLab [29], whose specifications are in Table 1. All measurements were collected using 10 Gbps networking and NVMe SSDs for Redis backup files. Linux fsync on the NVMe SSDs takes around 50–100 µs; systems with SATA3 SSDs will perform worse with the fsync-always option.

For the Redis implementation, we used Redis 3.2.8 for servers and "C++ Client" [34] for clients. We modified "C++ Client" to construct Redis requests more quickly.

Figure 11 shows the performance of Redis before and after adding CURP to its local logging mechanism; it graphs the cumulative distribution of latencies for Redis SET operations. After applying CURP (using 1 witness server), the median latency increased by 3 µs (12%). The additional cost is caused primarily by the extra syscalls for send and recv on the TCP socket used to communicate with the witness; each syscall took around 2.5 µs.

When a second witness server is added in Figure 11, latency increases significantly. This occurs because the Redis RPC system has relatively high tail latency. Even for the non-durable original Redis system, which makes only a single RPC request per operation, latency degrades rapidly above the 80th percentile. With two witnesses, CURP must wait for three RPCs to finish (the original to the server, plus two witness RPCs). At least one of these is likely

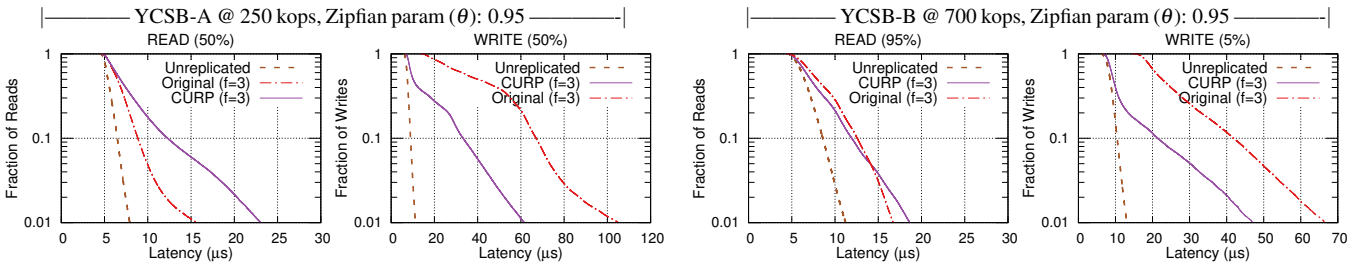

Figure 10: Complementary cumulative distribution of read and write latencies with CURP on a loaded server (250 kops for YCSB-A and 700 kops for YCSB-B). 10 clients issued read and write operations (using the read / write mix ratio of YCSB) for 1 min to a single server. The workloads used a Zipfian distribution with $\theta = 0.95$, which means 16% of operations are on keys that were accessed within the last 100 executed operations.

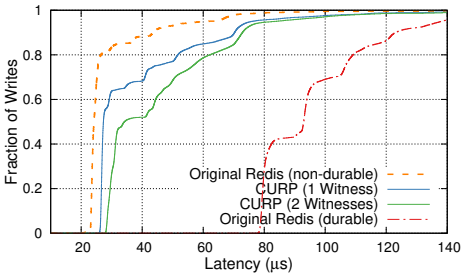

Figure 11: Cumulative distribution of latency for 100B random Redis SET requests with CURP. Writes were issued sequentially by a single client to a single Redis server. CURP used one or two additional Redis servers as witnesses. "Original Redis (durable)" refers to the base Redis without CURP, configured to invoke fsync on a backup file before replying to clients.

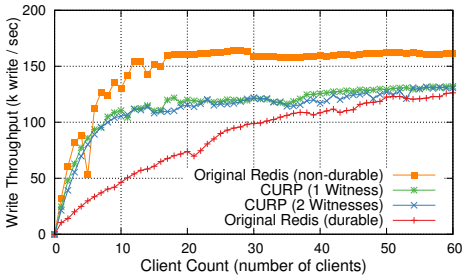

Figure 12: The aggregate throughput for one server serving 100B Redis SET operations with CURP, as a function of the number of clients. Each client repeatedly issued random writes back to back to a single server. "Original Redis (durable)" refers to the base Redis without CURP, but configured to invoke fsync before replying to clients.

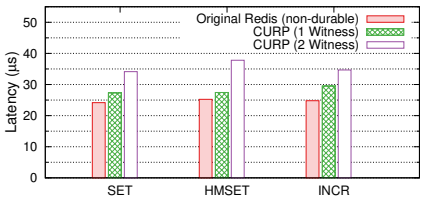

Figure 13: Median latencies before and after applying CURP on various Redis commands. All experiments select a random 30B key over 2M unique keys. SET used 100B random values, and each HMSET operation sets 1 member with a 100B value. The member key was 1B. Commands were issued sequentially by a single client to a single Redis server, with one or two additional Redis witness servers in CURP.

to experience high tail latency and slow down the overall completion. We didn't see a similar effect in RAMCloud because its latency is consistent out to the 99th percentile: when issuing three concurrent RPCs, it is unlikely that any of them will experience high latency.

Figure 12 shows the throughput of Redis SET operations for a single Redis server with varying numbers of clients. Applying CURP reduced the throughput of Redis about 18%. With a large number of clients, the original synchronous form of Redis can offer throughput approaching non-durable Redis. The reason for this is that Redis batches fsyncs in synchronous mode: in each cycle through its event loop, it processes all of the requests waiting on its incoming sockets, issues a single fsync, then responds to all of those requests. The disadvantage of this approach is that it results in very high latency for clients.

5.5 Applicability of CURP

CURP can be applied to a variety of operations, not just write operations in key-value stores. Redis supports many data structures, such as strings, hashmaps, lists, counters, and so on. All of these update operations (including ones that are non-idempotent or return read values) can benefit from CURP. Since each data structure is assigned to a specific key, CURP can execute many update operations on different keys without blocking on syncs.

Figure 13 shows the median latency with and without CURP on three different Redis commands: SET, which writes ASCII data to a string data structure; HMSET, which writes data to a member of a hashmap; and INCR, which increments an integer counter and returns its current value. For all three operations, latency overheads were small for CURP with 1 witness. CURP with 2 witnesses increased latency about 10 µs because of tail latency issues. We believe that the TCP transport library used by the C++ client is inefficient for waiting for multiple responses concurrently, and we will continue to investigate this.

6 Related work

Table 2 summarizes the performance of CURP and other fast replication protocols. The paragraphs below explain these numbers in detail. We present analytical performance instead of emprical results since empirical performance depends too much on implementation and underlying systems (e.g. CURP on RAMCloud and CURP on Redis have very different absolute performance).

Generalized Paxos [18] allows clients to complete operations (i.e. receive execution results) in 1.5 RTTs and supersedes *Fast Paxos* [19]. Both protocols allow clients to send requests directly to replicas and reduce latency from 2 RTTs to 1.5 RTT. Fast Paxos has a contention problem and performs well only at low throughput. Generalized Paxos resolves the contention problem by using commutativity; it groups commutative requests from concurrent clients into an unordered set, and it only orders between sets. Although Generalized Paxos allows a leader replica to learn that operations are committed in 1 RTT, clients need to wait another half RTT to receive the execution results from the leader; so

			CURP	Gen.Paxos	EPaxos	NOPaxos
Latency	LAN	read	1 RTT	1.5 RTTs	2 RTTs	1 RTT + α
		write	1 RTT	1.5 RTTs	2 RTTs	1 RTT + α
	WAN	read	~0 RTT	1.5 RTTs	~1 RTT	Not Avail.
		write	1 RTT	1.5 RTTs	~1 RTT	Not Avail.
load on leader		read	<1 RPC	~n RPCs	~2 RPCs	1 RPC
		write	1 RPC	~n RPCs	~2 RPCs	1 RPC

Table 2: Performance comparisons of replication protocols. "LAN" means intra-datacenter replications. "WAN" means geo-replication and assumes that all clients have a local replica; clients in a datacenter without local replicas must send requests to a remote replica and experience the WAN RTTs same as in "LAN". NOPaxos's RTT is longer than usual since network packets must detour through a sequencer. All latency numbers omitted the time to make data persistent, which is same for all protocols (1 persistence time per request) and insignificant with the use of modern fast storage technologies. "Load on leader" shows how many RPCs a leader (or master) processes per client request. "n" denotes the number of replicas.

its end-to-end latency becomes 1.5 RTTs, as opposed to 1 RTT for CURP. (See §B.3 for a detailed explanation why they cannot achieve 1 RTT.)

Egalitarian Paxos (EPaxos) [22] relies on commutativity to allow multiple leaders to propose and execute operations concurrently. This approach improves throughput. In geo-replicated environments, EPaxos allows clients to choose a nearby replica as leader, so operations can complete in 1 wide-area RTT. However, in LAN environments, EPaxos clients cannot hide the message delay to a leader, so operations take 2 RTT. Also, since EPaxos does not have a strong leader, read operations must run through full consensus and be written to replicated command logs; for read-heavy workloads, EPaxos will perform worse than traditional 2 RTT protocols with read leases, such as Raft [25]. On the other hand, CURP can directly execute read operations in masters or even in backups with the help of witnesses. Another limitation of EPaxos is that clients in a datacenter that doesn't host a replica must use a remote leader, increasing its latency to 2 wide-area RTTs.

Speculative Paxos [28] and *Network-Ordered Paxos* (NOPaxos) [21] reduce latency almost to 1 RTT by serializing client requests within network. Both protocols use SDNs to detour requests from all clients through a single network device (a root layer switch or middlebox); so, they can be deployed only in specialized environments (e.g. a privately-owned datacenter). Also, due to detouring of packets, they actually add latency overhead over unreplicated systems; Speculative Paxos (~25 µs) or NOPaxos(~16 µs) have higher latency overhead compared to CURP (~1 µs).

TAPIR [37] and Janus [23] commit distributed transactions in 1 wide-area RTT; before them, transaction commits took 2 RTTs: 1 for transaction prepares and 1 for geo-replicating the data of prepare. They flattened out these serial steps by replicating data before the prepare is executed. They modified concurrency control protocols to fix inconsistencies in replications. They also require commutativity of workloads for 1 RTT commits.

To avoid the performance penalty of consistent replications, *eventual consistency* [36] has been widely adopted in industry [10, 8, 5]. Systems using eventual consistency return from updates before replication is complete, and replications happen asynchronously; since nearby replicas are stale, clients must read from far-away masters for consistency. Pileus [35] and Tuba [2] allowed applications to declare their consistency and latency priorities, and they dynamically select replicas to read from.

Broadcast-broadcast (BB) protocols [4, 3, 12, 16] for total order broadcasts [11] have similarities to CURP. Senders in BB protocols broadcast a message to all destinations (replicated processes) plus a sequencer before ordering, followed by a second broadcast from the sequencer about the ordering information. Some variants of BB protocols [3, 12] exploit the fact that broadcasts are mostly delivered in-order in small LAN environments and let processes optimistically consume messages without waiting for the ordering information from the sequencer. If the suspected order turned out to be different from the order determined by the sequencer, the process must rollback to correct the inconsistency. On the other hand, in CURP, replicas wait for the ordered replication from a master instead of executing operations with a presumed ordering, so CURP doesn't require rollbacks, which is expensive and difficult to implement. Furthermore, even if client requests arrive in a master and witnesses out of order, CURP still achieves 1 RTT as long as the reordered requests are commutative.

7 Conclusion

In this paper we have uncovered an opportunity for introducing concurrency into mechanisms for consistent replication. By exploiting the commutativity of operations, replication without ordering can be performed in parallel with sending requests to an execution server. This general approach can be applied to improve a variety of replication mechanisms, including primary-backup approaches and consensus protocols with strong leaders. We presented Consistent Unordered Replication Protocol (CURP), which supplements standard primary-backup replication mechanisms. CURP reduces the latency to complete operations from 2 RTTs to 1 RTT while retaining strong consistency. We implemented CURP in RAMCloud and Redis to demonstrate its benefits.

Acknowledgements

We thank our shepherd, Manos Kapritsos, and our anonymous NSDI and OSDI reviewers for their feedback. Thanks to Stephen Yang and Collin Lee for helping on improving the clarity of this paper. This work was supported by the industrial affiliates of the Stanford Platform Lab and by the Samsung Scholarship.

References

[1] GlusterFS. https://www.gluster.org, 2017. Accessed: 2017-09-22.

[2] Ardekani, M. S., and Terry, D. B. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design*

and Implementation (OSDI 14) (Broomfield, CO, 2014), USENIX Association, pp. 367–381.

[3] BALAKRISHNAN, M., BIRMAN, K., AND PHANISHAYEE, A. PLATO: Predictive latency-aware total ordering. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems* (Leeds, UK, 2006), SRDS '06, IEEE Computer Society, pp. 175–188.

[4] BIRMAN, K., SCHIPER, A., AND STEPHENSON, P. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9*, 3 (Aug. 1991), 272–314.

[5] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 49–60.

[6] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[7] CHODOROW, K., AND DIROLF, M. *MongoDB: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2010.

[8] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1277–1288.

[9] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, IN, 2010), SoCC '10, ACM, pp. 143–154.

[10] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, WA, 2007), SOSP '07, ACM, pp. 205–220.

[11] DÉFAGO, X., SCHIPER, A., AND URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. 36*, 4 (Dec. 2004), 372–421.

[12] FELBER, P., AND SCHIPER, A. Optimistic active replication. In *Proceedings of the The 21st International Conference on Distributed Computing Systems* (Phoenix, AZ, USA, 2001), ICDCS '01, IEEE Computer Society, pp. 333–341.

[13] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. *SIGOPS Oper. Syst. Rev. 37*, 5 (Oct. 2003), 29–43.

[14] GRAY, J., SUNDARESAN, P., ENGLERT, S., BACLAWSKI, K., AND WEINBERGER, P. J. Quickly generating billion-record synthetic databases. *SIGMOD Rec. 23*, 2 (May 1994), 243–252.

[15] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA, 2010), USENIXATC'10, USENIX Association, pp. 11–11.

[16] KAASHOEK, M. F., AND TANENBAUM, A. S. Group communication in the amoeba distributed operating system. In *[1991] Proceedings. 11th International Conference on Distributed Computing Systems* (May 1991), pp. 222–230.

[17] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[18] LAMPORT, L. Generalized consensus and Paxos. Tech. rep., March 2005.

[19] LAMPORT, L. Fast Paxos. *Distributed Computing 19* (October 2006), 79–103.

[20] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITA, S., AND OUSTERHOUT, J. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, CA, 2015), SOSP '15, ACM, pp. 71–86.

[21] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. K. Just say no to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, 2016), OSDI'16, USENIX Association, pp. 467–483.

[22] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, PA, 2013), SOSP '13, ACM, pp. 358–372.

[23] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX*

Conference on Operating Systems Design and Implementation (Savannah, GA, 2016), OSDI'16, USENIX Association, pp. 517–532.

[24] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada, 1988), PODC '88, ACM, pp. 8–17.

[25] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, 2014), USENIX Association, pp. 305–319.

[26] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11, ACM, pp. 29–41.

[27] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[28] PORTS, D. R. K., LI, J., LIU, V., SHARMA, N. K., AND KRISHNAMURTHY, A. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA, 2015), NSDI'15, USENIX Association, pp. 43–57.

[29] RICCI, R., EIDE, E., AND TEAM, C. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE 39*, 6 (2014), 36–38.

[30] SANFILIPPO, S., ET AL. Redis. https://redis.io/, 2015. Accessed: 2017-04-18.

[31] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[32] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (May 2010), pp. 1–10.

[33] SIVASUBRAMANIAN, S. Amazon dynamoDB: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, AZ, 2012), SIGMOD '12, ACM, pp. 729–730.

[34] SPRENKER, L., AND HAMMOND, B. Redis C++ Client. https://github.com/mrpi/redis-cplusplus-client, 2011. Accessed: 2017-04-20.

[35] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, PA, 2013), SOSP '13, ACM, pp. 309–324.

[36] VOGELS, W. Eventually consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44.

[37] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, CA, 2015), SOSP '15, ACM, pp. 263–278.

[38] ZHAO, W. Fast Paxos made easy: Theory and implementation. *International Journal of Distributed Systems and Technologies (IJDST) 6*, 1 (2015), 15–33.

A Informal Proof of Correctness

With the normal operation behaviors described in §3.2, the recovery protocol in §3.3 guarantees the following correctness properties.

- **Durability**: if a client completes an operation, it survives server crashes.
- **Consistency**: if a client completes an operation, its result returned to an application remains consistent after server crash recoveries.
- **Linearizability**: an operation appears to be executed exactly once between start and completion.

Before presenting proofs, we reiterate some key behaviors of the CURP protocol.

(Rule 1) from §3.2.1, a client only completes an update operation if (1) it is recorded in all f witnesses or (2) it is replicated to f backups.

(Rule 2) a completed unsynced operation must be individually commutative with all preceding operations that are not synced yet. This is the behavior described in §3.2.3; a master must sync before responding if the current operation is not commutative with any other existing (preceding) unsynced operations.

Now, we present proof sketches for the properties.

Durability: recovery of a master only completes after recovery from 1 backup and 1 witness, and the completed operation must exist in the backup or the witness by (Rule 1); thus, the completed operation must be recovered when the recovery is completed. □

Consistency: Consider an individual completed operation α and its consistency. To prove that α's result doesn't change even after crash recovery, we will think about the operation execution sequence before α, which we will call *history* of α (or H_α).

Case 1: the operation α has been synced to the backup used for recovery. This operation will be recovered from the backup (phase 1) and any replay from witnesses (phase 2) will be ignored (by RIFL). Since backup syncs preserve the execution order of operations, the H_α didn't change; so the post-recovery execution sequence should regenerate the original execution result of α.

Case 2: the operation α has not been synced to the backup used for recovery. α must have been recorded in all witnesses by (Rule 1) and will be recovered during phase 2. We can split the original execution history of α into two parts as in Figure 3: $\langle$synced$\rangle$ followed by $\langle$unsynced$\rangle$. The 1st phase of recovery will recover the exactly same execution history for the $\langle$synced$\rangle$ part. By (Rule 2), we know that losing any $\langle$unsynced$\rangle$ part of history after crash will not change the execution result of α. During phase 2 of recovery (from a witness), we may replay some other operations before replaying α, but the result of α doesn't change since all operations recorded in the witness must be commutative. □

Linearizability: we assume that the underlying system before applying CURP guarantees linearizability for operations that are replicated to backups. CURP may break the linearizability of the underlying system since masters in CURP return before syncing to backups. So, we will reason about how CURP recovers from master crashes without breaking linearizability.

The definition of linearizability can be reworded as following: if the execution of an operation is observed by the issuing client or other clients, no contrary observation can occur afterwards (i.e. it should not appear to revert or be reordered). Since we only care about what happens after recovery, we prove the following proposition: if the execution of an individual operation α is observed *before crash*, no contrary observation can occur *after recovery*.

Case 1: the execution of α was observed by other dependent operations (e.g. reads). By (Rule 2), the master must have synced α to backups since dependent operations don't commute with α. Since it was replicated to backups, α will be linearizable as long as the underlying system is.

Case 2: the execution was observed only by the completion of α. α must be recovered because of the Durability property. The only observation about α before crash was the returned execution result, and it must be still consistent even after recovery because of the Consistency property.

Case 3: no observation was made before crash. α may be lost if it didn't reach to either the backup or witness used for recovery. In CURP, the client keeps retrying until it can complete α. Regardless of whether α was recovered or not, RIFL ensures the retry will only execute α at-most once and return the result of the sole execution. □

B Extra Discussions

B.1 Why Are Witnesses Separate from Backups?

By having witnesses separated from backups, CURP requires fewer changes to the existing systems and is more applicable to many wildly different backup mechanisms. Both of our two implementations leveraged this flexibility: in RAMCloud, a master keeps changing backups to which it replicates (to spread data over the entire cluster), so clients don't know which backups are currently used by the master; in Redis, operation logs are stored in local disks to ensure durability, so there are no separate backup servers to which CURP clients can record inputs. Thus, separating witnesses from backups improves CURPs applicability to many existing primary-backup systems.

On the other hand, when designing a new storage system, combining witnesses and backups can bring extra performance benefits. When they are combined, clients directly send requests to a master and backups, which now also serve as witnesses. The key change is masters now sync operation orders (by listing IDs as in witness **gc** RPCs) instead of full client requests; then backups lookup the matching requests from their witness storage and move them to backup logs. This approach will lower network bandwidth consumption. Also, most witness **gc** RPCs can be eliminated; immediately

after handling the sync, the requests in the witness storage can be deleted as they are now safe in the backup log. (For safety, the recovery protocol must pick 1 witness/backup combo and must not mix.) This saving of **gc** RPCs will improve masters' throughput and will reduce the chance of commutativity conflicts.

B.2 Extending CURP to Consensus Protocols

This section illustrates how CURP can be extended to reduce the latency of consensus protocols. CURP can be integrated in most consensus protocols with strong leaders (e.g. Raft [25], Viewstamped Replication [24]). In such protocols, clients send requests to the current leader, which serializes the requests into its command log. The leader then replicates its command log to a majority of replicas before executing the requests and replying back to clients with the results. This process takes 2 RTTs, and CURP can reduce it to 1 RTT.

As in primary-backup replication, CURP on consensus allows clients to replicate requests to witnesses in parallel with sending requests to the leader; the leader then speculatively executes the requests and responds to clients before replicating the requests to a quorum of replicas. A client can complete an operation if it is accepted by a superquorum of witnesses or committed in a quorum of replicas.

To mask f failures, consensus protocols use $2f+1$ replicas, and systems stay available with f failed replicas. For the same guarantee, CURP also uses $2f+1$ replicas, but each replica also has a witness component in addition to existing components for consensus. Although CURP can proceed with $f+1$ available replicas, it needs $f+\lceil f/2\rceil+1$ replicas (for superquorum of witnesses) to use 1 RTT operations. With less than $f+\lceil f/2\rceil+1$ replicas, clients must ask masters to commit operations in $f+1$ replicas before returning result (2 RTTs).

Like masters in regular CURP, leader replicas execute operations speculatively if they are commutative with existing unsynced operations; for an incoming client request, the leader serializes it into its command log, executes it, and responds to the client before committing it in a majority of replicas.

For clients to complete an operation in 1 RTT, it must be recorded in a *superquorum* of $f+\lceil f/2\rceil+1$ witnesses. The reason why CURP needs a superquorum instead of a simple majority is to ensure commutativity of replays from witnesses during recovery. During recovery, only $f+1$ out of $2f+1$ replicas (each of which embeds a witness) might be available. If a client could complete an operation after recording to $f+1$ witnesses, the completed operation may exist in only 1 witness out of available $f+1$ witnesses during recovery (since intersection of two quorums is 1 replica). If the other f witnesses accepted other operations that are not commutative with the completed operation (since each witness enforces commutativity individually), recovery cannot distinguish which one is the completed one; executing all appearing in any $f+1$ witnesses is also not safe since they are not commutative, so they must be replayed in a correct order.

For correctness, the client requests replayed from witnesses during recovery must be *commutative* and *inclusive* of all completed operations that are not yet committed in a majority of replicas. By recording to a superquorum, all completed operations (but not yet committed) are guaranteed to exist in a majority ($\lceil f/2\rceil+1$) of any quorum of $f+1$ witnesses, and any operations that don't commute with the completed operations cannot exist in more than $\lfloor f/2\rfloor$ (less than majority of any quorum). Thus, during recovery, all requests that appear in a majority ($\lceil f/2\rceil+1$) from any quorum of $f+1$ witnesses are guaranteed to be commutative and include all completed operations; so, recovery can replay requests that appear in more than $\lceil f/2\rceil+1$ witnesses out of any $f+1$ witnesses.

When leadership changes (e.g. leader election in Raft [25] or view change in Viewstamped Replication [24]), the new leader must recover from witnesses before accepting new operations. To do so, the new leader must collect saved requests from at least $f+1$ witnesses. This collection can be included in the existing data collection (e.g. Raft votes) that is required by most leadership change protocols. As mentioned in the previous paragraph, the new leader should only replay client requests that are recorded in at least $\lceil f/2\rceil+1$ witnesses to ensure commutativity.

After leadership changes, the state machine of the old leader could have diverged from other replicas due to speculatively executed operations that were not recovered from witnesses. To fix this, the old leader must reload from a checkpoint that does not have speculative executions. However, we can avoid reloading from checkpoints if the leadership change was not because of a crash or disconnect of the old leader; instead of requring old leader to reload from a checkpoint, we can require the new leader to fetch and commit all uncommitted operations in the old leader's command log.

The last problem introduced by speculative execution is that clients may use old zombie leaders (which believe they are current leaders). Zombie leaders were not possible before CURP since an operation must be committed in a majority before being executed and at least one replica would reject the operation. To prevent clients from completing operations with an old (possibly disconnected) leader, they tag record RPCs with a term number (e.g. a Raft term or a view-number in Viewstamped Replication), which increments every time when leadership changes. A witness checks the term number against the term used by its replica (recall that a witness is a part of a consensus replica); if the record RPC has an old term number, the witness rejects the request and tells the client to fetch new leader information.

CURP can use read leases like many consensus protocols so that read operations can be executed solely by leaders within 1 RTT without recording to witnesses. Optimizing read operations using read leases is common for consensus protocols with strong leaders. A leader replica with a valid read lease can safely execute read operations without committing the read operations through consensus. For the

optimization, each replica grants the read lease to the current leader, promising not to agree on a leader change for a lease period. With valid leases from a majority of replicas, the leader knows that no operations can be committed from other replicas, so it can safely execute read operations without consulting with other replicas. CURP does not interfere with this read lease mechanism.

B.3 Why Do Fast / Generalized Paxos require 1.5 RTTs?

There is a widespread misunderstanding that both Fast Paxos and Generalized Paxos already achieve 1 RTT operations. The confusion probably stems from the fact that both Fast and Generalized Paxos allow Paxos learners to know about acceptance of an operation in 1 RTT.

However, 1 RTT is sufficient to know only that an operation is committed but not enough to know the result: that requires another 0.5 RTT. The abstract for Generalized Paxos says that a server can *execute* the command in two message delays; however, it take an additional message delay for the result to reach a client, for a total of three message delays (1.5 RTT). It doesn't help for the client to be a Paxos learner, because even learners don't know the result after 1 RTT.

For most operations, results are not trivial and clients must wait for the results from real executions before completing operations. Many writes, such as conditional writes or read-modify-writes, have results that clients cannot know before executions. Blind writes (those that don't return results) could potentially complete in 1 RTT. However, truly blind writes are rarely feasible because they can return exceptions, such as "table no longer on this server" or "permission denied"; clients must be aware of these exceptions.

As a result, Fast/Generalized Paxos are generally considered to have 1.5 RTT latency for clients to complete operations. [21, 28, 38]

C Implementation Details

C.1 Modifications to RIFL

RIFL [20] is a mechanism for detecting duplicate invocations of RPCs. With RIFL, masters make a durable *completion record* of each RPC that updates state, which includes the RPC result. The completion record survives crashes and can be used to detect duplicate invocations of the RPC. When a duplicate is detected, the master skips the execution of the RPC and returns the result from the completion record.

RIFL has two mechanisms for garbage collecting completion records: (1) on RPC requests, clients piggyback acknowledgments of the results of their previous requests (so servers can safely delete these completion records), and (2) clients maintain leases in a central server; if a client's lease expires, masters can delete all completion records for that client. Both of these must be modified to work with CURP.

Since both garbage collection mechanisms assume that retries always come from the same client that made the original request, RIFL must be modified to accommodate retries from witnesses. Firstly, once clients acknowledge the receipts of results, masters remove their completion records and start to ignore (not returning results) the duplicate requests. Since replays from witnesses happen in random orders, acknowledgements piggybacked on later requests can make masters to ignore the replay of earlier requests. Thus, clients' acknowledgments included in RPC requests must be ignored during recovery from witnesses.

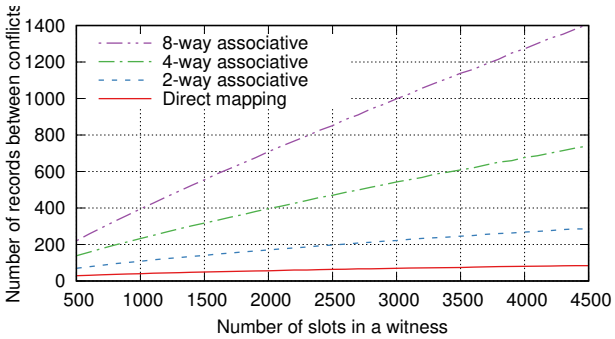

Figure 14: Simulation results for the expected number of recordings before a collision occurs in a witness' cache, assuming a random distribution of keys. Each data point is the average of 10000 simulations. Introducing associativity reduces the chance of collisions significantly.

Secondly, if a client crashes and its lease expires, masters remove all of the completion records for the client; then any requests from the expired client are ignored. This can be a problem in CURP since the replay of the expired client's requests will be ignored during witness-based recovery. To prevent this, masters must sync all operations to backups before expiring a client lease. In practice, the period of syncs is much smaller than the grace period between the time of a client crash and the time of its lease expiration; so, most systems are safe automatically.

C.2 Why Use Set-associative Cache for Witnesses?

We initially used a direct-mapped cache instead of set-associative cache, but this resulted in a high rate of rejections because of conflicts (i.e. no slot is available for the mapped set). Figure 14 shows the expected number of recordings before a conflict occurs on a witness slot. Using a direct mapping and 4096 total slots, it is expected to have a false conflict after about 80 insertions. Thus, we switched to 4-way associative cache, to reduce witness rejections. We didn't need 8-way associativity (a bit slower than 4-way) since the number of requests in witnesses is already limited by commutativity. (Once a master hits a non-commutative operation and syncs to backups, all saved requests in the witness are garbage collected.)

D Additional Evaluations

D.1 RAMCloud's Throughput by Batch Size

Figure 15 shows the single-server throughput of write operations with CURP while varying the aggressiveness of syncs. After introducing CURP, RAMCloud can delay the sync to backups after responding back to clients; delaying and batching sync to backups makes the server more efficient and improves throughput about 4 times. Since RAMCloud

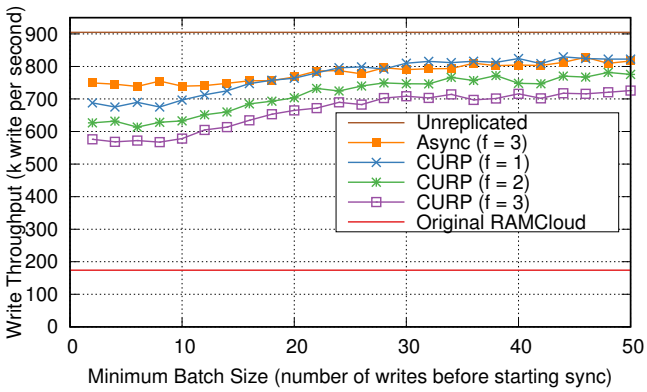

Figure 15: The aggregate throughput for one server serving 100B RAMCloud writes with CURP, as a function of sync batch size. Each client repeatedly issued random writes back to back to a single server. "Original RAMCloud" refers to the base RAMCloud system before adding CURP. "Unreplicated" refers to RAMCloud without any replication. Each datapoint was measured 15 times, and median values are displayed.

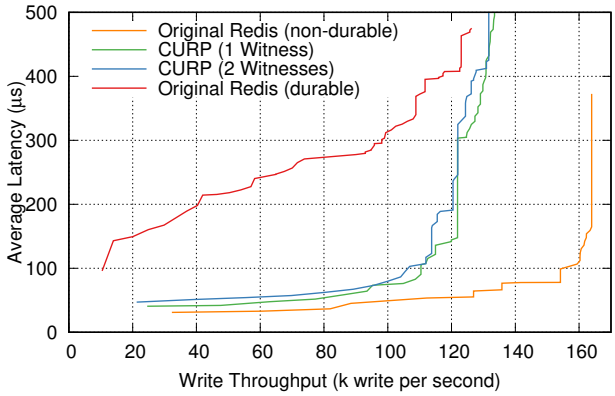

Figure 16: Observed latency at a specific throughput level for one server serving 100B Redis SET operations with CURP. "Original Redis (durable)" refers to the base Redis without CURP, but configured to invoke fsync before replying to clients. Original Redis processes requests from multiple clients, fsyncs once per eventloop, and replies to all clients.

allows only one outstanding sync, syncs are naturally batched for around 15 writes even at 1 minimum batch size.

D.2 Redis Latency vs. Throughput

Figure 16 shows observed latency during the throughput benchmark. Both CURP and non-durable Redis maintains latency low until it reaches 80% of max throughput. The latency of durable Redis increases almost linearly due to bathcing. The original Redis is designed to provide maximum throughput under high load and natively batches fsyncs; for each event-loop cycle, Redis iterates through TCP sockets for all clients and executes all requests from them; after the iteration, Redis fsyncs once and responds to the clients. This batching amortizes the cost of fsync, and throughput of durable Redis approaches that of non-durable Redis as the number of clients increases. However, this batching adds extra delay before responding back to clients, so latency increases linearly.