ACHIEVING BOTH LOW LATENCY AND STRONG CONSISTENCY IN LARGE-SCALE SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Seo Jin Park
October 2019

This dissertation is online at: http://purl.stanford.edu/xc977hj8024

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Mazieres

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Today's datacenter applications demand large-scale and low-latency systems. Unfortunately, consistency mechanisms were not designed in consideration of large-scale and low-latency settings. Most existing consistency mechanisms incur huge penalties on scalability or latency, so many datacenter systems have forgone consistency. As a result, application developers or end-users suffer from unpredictable system behaviors.

This dissertation presents two new consistency mechanisms for large-scale and low-latency systems: Reusable Infracture for Linearizability (RIFL) and Consistent Unordered Replication Protocol (CURP). RIFL provides a general-purpose mechanism for converting at-least-once RPC semantics to exactly-once semantics, thereby making it easy to turn non-linearizable operations into linearizable ones. RIFL is designed for large-scale systems and is lightweight enough to be used in low-latency environments. On the RAMCloud storage system, RIFL adds only 0.5 to the 13.5 µs base latency for durable writes, and it can support 1 million clients with minimal performance degradation ($\sim$5% latency increase).

We also used RIFL to construct a new multi-server transaction mechanism (RIFL-TX) in RAMCloud; RIFL's facilities significantly simplified the transaction implementation. The transaction mechanism can commit simple distributed transactions in about 20 µs and it outperforms the H-Store main-memory database system for the TPC-C benchmark.

Replication is a must-have for large-scale systems for masking individual server failures and maintaining high availability. However, consistent replication incurs significant overhead, such as doubling the latency of operations. As a result, many large-scale systems have settled for weakly consistent replication. To address this dilemma, I present Consistent Unordered Replication Protocol (CURP), which removes most of the overhead of consistent replication. CURP avoids performance penalties by allowing clients to directly replicate their requests, as long as the requests are commutative. This strategy allows most operations to complete in 1 RTT (the same as an unreplicated system). On RAMCloud, CURP improved write latency by $\sim$2x (14 µs $\rightarrow$ 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).

Acknowledgments

First and foremost, I would like to thank my advisor, John Ousterhout, for his support, guidance, and commitment. Besides being reasonable and supportive, John has been a devoted teacher for me. In the early years of my Ph.D., he taught me how to write system software, how to present ideas, and how to write academic papers. He also has held a high standard for research so that I can become a good researcher. He never allowed us to publish performance numbers generated from benchmark software without verifying them thoroughly, and he asked us to measure one-level deeper until we certained that we got the best possible performance. I learned so much from John, and I truly enjoyed my Ph.D. journey because of him.

In addition to my advisor, Mendel Rosenblum, David Mazières, Keith Winstein, and David Donoho served on my defense committee. I very much appreciate the time and effort they took to supervise my dissertation work.

This dissertation wouldn't be possible without my labmates in room 444 (and 322 later). Collin Lee was a fantastic collaborator for the RIFL work. Discussing design or implementation ideas with him was fun. Stephen Yang was always available when I need someone to discuss my code or writing, and he was my advisor for hardware DIY, including upgrading my car and electric scooter. Yilong Li was an awesome hacker and made it possible to run MilliSort (not included in this dissertation) in a real large-scale cluster. Henry Qin taught me many Linux tools, notably tmux, and I also thank him (and Rebecca) for letting me stay in his place while I am finishing this dissertation. Ankita Kejriwal was almost the only senior student when I joined the group, and I thank her for teaching me about RAMCloud and letting me take the RAMCloud transaction project. Jonathan Ellithorpe was the first user of RAMCloud transactions and found several bugs in my code. Behnam Montazeri's sarcastic humor made the office more amusing. Jacqueline Speiser is still a great friend even after she finished her master's. I also thank the RAMCloud seniors, Ryan Stutsman, Diego Ongaro, and Stephen Rumble, for their RAMCloud work upon which I built most of my research.

I thank Samsung Scholarship and the industrial affiliates of the Stanford Platform Lab and Stanford Experimental Datacenter Lab for their generous support during my graduate study.

Finally, I thank my parents, Myoungsuk and Hwasik, for their unconditional love, patience, and encouragement. I wouldn't be able to come to the US for college or join this Ph.D. program without their support. I feel very fortunate to have them as my parents.

Contents

List of tables

List of figures

Chapter 1

Introduction

Today's datacenter applications can be characterized as large scale and low latency. Regarding scale, many web applications service hundreds of millions or sometimes billions of users (Facebook has 2.1 billion daily users as of 2019 [82]). To store and process the data of their users, web applications operate on a large number of machines connected by a network, called large-scale systems. Back in 2013, the number of servers used by Microsoft and Google exceeded a million [10]. The need for large-scale systems is growing even more today since the increasing use of machine learning requires systems that can store a large amount of data. Also, the growing number of IoT devices further extends the need for large-scale systems to store and process their sensor data.

Regarding latency, both industry and academia have been striving to lower the latency of operations used by datacenter applications. Faster operations can enable complex applications that process more data within a given time budget. Facebook adopted the Memcached [52] in-memory cache to enable real-time rendering of timeline webpages; the adoption of Memcached reduced the latency of remote object reads to around 300 µs [56]. In academia, recent research works drove down the latency even more: RAMCloud [62] and FaRM [25] storage systems enabled sub-10 µs read latency without contention; Homa [53], Shenango [60], and Arachne [70] enabled low latency even in highly loaded datacenter settings.

Unfortunately, most systems featuring large scale and low latency forgo strong consistency; in other words, those systems sometimes exhibit unpredictable abnormal behaviors, making application development difficult (details in §1.1). The existing mechanisms for consistency were not designed in consideration of large scale and low latency, so applying the consistency mechanisms incurs a huge penalty on the scalability or latency of systems.

This thesis addresses the lack of consistency mechanisms for large scale and low latency. By

Alice posts “Free table! Giving out to the first commenter.”
Bob comments “I am interested!,” and confirms his comment shows up first.
Charlie also comments “I am interested!,” and confirms his comment shows up first.
Alice sees Bob’s comment before Charlie’s, and she gives the table to Bob.
Charlie gets confused and upset for the table was not given to him.

Figure 1.1: An example of anomalies visible to Facebook users [51]

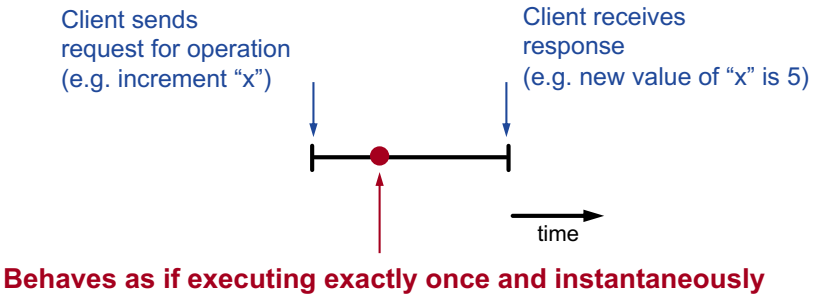

Figure 1.2: Linearizability consistency guarantee

presenting consistency mechanisms that do not significantly harm scalability or latency, this dissertation demontrates that it is possible to achieve consistency for large-scale and low-latency systems.

1.1 Consistency Challenge

The level of consistency indicates how predictably a system behaves. Systems with weak consistency guarantees exhibit behaviors that are not intuitive to users; so, it is notoriously difficult to write programs correctly on them. In many cases, applications propagate anomalies to end users, who become confused and frustrated [66].

Here are examples of consistency anomalies. The first example is duplicate execution. It is not unusual to see a single user request get executed multiple times. If someone sends a message on an Internet messaging service such as Google Hangout or iMessage, the same message often appears multiple times, especially with unstable Internet connections. Such systems cannot be used for critical operations like bank transfers. A second example is delayed propagation of updates. After a Facebook user posts a comment and confirms it is published, his/her friend may not see the new comment for a while. This anomaly can result in a more complex problem in Figure 1.1. In practice, Facebook reported that 0.0004% of their read operations returned stale values [51].

The ideal consistency guarantee is *linearizability* [34], and this dissertation work targets to provide linearizability. A client operation is considered linearizable if it appears to happen exactly

once, and instantaneously, at some time point between when the client sends a request and receives a response (See Figure 1.2). In other words, linearizability guarantees exactly-once atomic execution and synchronous execution (execution must complete before returning the response). If all operations supported by a system are linearizable, the system is considered linearizable. With linearizability, it is very intuitive to write application programs. A linearizable system doesn't allow any of the anomalies in the previous paragraph.

Unfortunately, linearizability is difficult to achieve for distributed systems. Since data is spread over multiple servers, maintaining consistency requires complex coordination among multiple servers. For example, let's consider an operation that updates multiple objects in different servers. Such an operation cannot be simply executed at each server independently. If one does that, linearizability can be violated. To illustrate the problem, let's consider a bank transfer between acounts whose balance data are stored in different servers. If each server individually deducts or adds the transfer amount, some other clients may see the increased balace without seeing the reduction from the source, or vice versa. The transfer operation appears not to happen exactly-once at a time point, so it is not linearizable. For the illusion of atomic exactly-once execution, a mechanism like two-phase locking must be used; first, all involved objects must be locked so that no other operations may read or modify the objects, and then each server can execute the updates. In general, systems need non-trival mechanisms to keep consistency.

Furthermore, the common mechanisms for strong consistency incur a huge penalty on scalability and latency, which prevents large-scale low-latency systems from employing the consistency mechanisms. As seen in the multi-object operation example, atomicity across many servers requires an extra prepare step before taking action, which increases latency at least by one round-trip time. Regarding scalability, a single coordinator server usually drives such a consistency protocol (e.g., a transaction coordinator server, a primary server in primary-backup replication); such a single coordinator server becomes a bottleneck for scalability and limits throughput.

The complexity and performance penalty made many large-scale systems forgo consistency. For example, updates can be delayed (eventual consistency) in [16, 72, 67, 23, 43], and multi-object atomic updates (ACID transactions) are not supported in [23, 8, 18, 16].

However, consistency mechanisms' impacts on scalability or latency can be minimized (almost avoided) with careful protocol designs and implementations. In this disseration, I present three consistency mechanisms for linearizability with minimal performance penalties (details in §1.3) so that large-scale low-latency systems can achieve strong consistency.

1.2 Latency Challenge

Today's datacenter applications run not only on slow networking (>10 ms) but also on extremely fast networking (∼10 µs). Ideal consistency mechanisms should avoid penalty on both slow and fast networking.

Global-scale datacenter applications use high latency networking to access data in remote datacenters (e.g., geo-replication), whose latency is usually tens or hundreds of milliseconds. When we make operations across datacenters consistent, it is important to avoid additional inter-datacenter communication (each additional communication step would add tens or hundreds of milliseconds to the operation latency).

Those applications also have many operations within fast intra-datacenter networking. New datacenter networking hardware such as Infiniband and 10 Gigabit Ethernet (currently being upgraded to 100 Gigabit) has enabled extremely low latency communication; a small message can travel from a host's network interface card (NIC) to another host's NIC within 2µs [1, 65].

Unfortunately, the benefits of microsecond-scale networking devices can be nullified quickly by slow software [11]. For example, traditional datacenter networking stacks (e.g., thread dispatch, interrupt, RPC, TCP/IP) may add 75µs overhead to the 2µs NIC-to-NIC round-trip time; so, reducing the software overheads for datacenter networking stacks has been an active area of reasearch [62, 25, 12, 69, 39, 48].

Consistency mechanisms tend to have high software overhead as well since they usually have to send/receive extra messages and track more states. This is a problem since consistency mechanisms must run on majority of client operations (e.g., updates). Thus, with today's low-latency networking, minimizing the software overhead of consistency mechanisms is as important as minimizing message exchange steps.

By minimizing message exchange steps and software overhead, consistency mechanisms can be employed without hurting the performance of datacenter applications.

1.3 Contribution: Consistency Mechanisms For Large-scale Low-latency Systems

This dissertation addresses the lack of consistency mechanisms for today's datacenter applications which run at extremely low latency and large scale. My contribution is re-designing and re-implementing the following three distributed systems primitives (Figure 1.3) so that datacenter

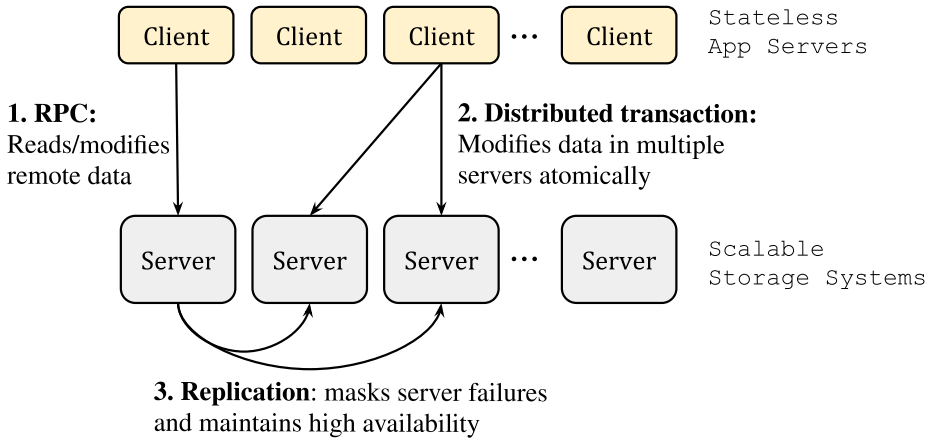

Figure 1.3: The three distributed systems primitives I address in this dissertation. A typical user request arrives at a client application server, which reads and/or updates the data in storage systems through RPCs or distributed transactions. All updates on data in servers are replicated to other servers.

applications can enjoy strong consistency without sacrificing either scalability or latency much.

1) Remote Procedure Call (RPC): distributed systems use RPCs to operate on data in remote machines. Using RPCs with consistency is difficult when failures can happen. When a client doesn't receive a response, the client usually retries by resending the request; this can result in a duplicate execution of the same operation, leading to consistency issues such as double increments or overwriting later operations.

I developed *Reusable Infrastructure for Linearizability (RIFL)* [47] which transforms an inconsistent RPC to a linearizable one. It prevents duplicate executions by saving the results of RPC executions and allowing systems to handle retries by returning the saved result. I implemented it on a low-latency large-scale storage system, RAMCloud [62], whose write operation takes only 14 µs. RIFL on RAMCloud incurs only 0.5 µs overhead (3.5% of RAMCloud's write latency), and RIFL can support 1 million clients.

2) Distributed Transaction: a transaction atomically updates data in multiple servers. In general, a client sends a transactional update request to a transaction coordination server which drives a two-phase commit procedure to all participating servers. However, relaying through a coordination server doubles latency and limits scalability. Those overheads can be avoided by driving two-phase commits from clients [6]. However, this complicates recovery since servers now have to clean up client failures.

Along with Collin Lee, I leveraged RIFL to simplify the recovery of a client-driven commit protocol and implemented a new multi-object distributed transaction mechanism in RAMCloud (*RIFL-TX* [47]). By using clients to manage two-phase commits, RIFL-TX can commit small distributed transactions in 20 µs, which is just 6 µs more than 14 µs for non-atomic updates on RAMCloud. By avoiding a centralized transaction coordinator server, RIFL-TX's throughput scales almost linearly with the number of servers.

3) Replication: large-scale systems mask individual server failures and maintain high availability by replicating data to backup servers. Providing linearizability on replicated data incurs performance penalties; consistent replication (which doesn't break linearizability) often doubles latency and reduces throughput, causing systems to accept weaker consistency such as eventually consistent replication.

I developed *Consistent Unordered Replication Protocol (CURP)* [63], which avoids the performance overhead of consistent replication. By leveraging the commutativity of concurrent operations, CURP allows clients to replicate each operation to backups in parallel with sending the request to the execution server. Since replication is overlapped with the execution of the operation, CURP mostly eliminated the cost of replication (i.e., client operations complete in 1 RTT). On RAMCloud, CURP cuts latency in half (14 µs $\rightarrow$ 7.1 µs; 1 µs overhead compared to no replication) and quadrupled throughput.

1.4 Approaches to Avoid Harming Latency or Scalability

When designing the three distributed systems primitives, I used the following approaches to minimize their impacts on scalability or latency.

1. **Minimize the use of a single coordinator.** For scalability, we must refrain from using a single server for common operations. Either splitting the coordination work among multiple servers or delegating the coordination to clients can improve scalability.

2. **Minimize message delays by using client-driven protocols.** Having a coordinator server as a middleman between a client and data servers incurs extra one rount-trip time (RTT) delay for relaying client requests to the coordination server. This extra 1 RTT can be cut if clients directly drive consistency protocols such as a transaction commit protocol or replication protocol.

Approach	How each approach is applied		
	Exactly-once RPC (RIFL)	**Distributed Transaction (RIFL-TX)**	**Replication (CURP)**
1. Avoding single coordinator	Instead of using a single server for assigning the globally unique IDs, RIFL delegates the ID assignment to each client. (§2.2)	No single transaction coordinator server. (§2.5.3)	Instead of sending data from a single primary server to all backups, each client sends the data to all servers. (§4.2.2.1)
2. Client-driven protocol	N/A	Two phase commit is driven by each client. (§2.5.3)	Data replication is driven by each client. (§4.2.2.1)
3. RIFL as a building block	N/A	RIFL is used to prevent races between slow clients and the servers coordinating recovery. (§2.5.5)	Exactly-once semantics are used to prevent recovering the same operation twice (first time from a witness and second time from a backup). (§4.2.3)
4. Moving work from critical path	N/A	The decision phase (2nd) of two phase commit happens asynchronously. (§2.5.3)	A primary server replicates ordered operations to backups asynchronously in batches. (§4.4.4)

Table 1.1: Examples of how each approach is used.

3. **RIFL as a building block.** Delegating coordination to stateless clients significantly complicates crash recovery; a protocol must handle client crashes by cleaning up the unfinished work (e.g., aborting an unfinished transaction). However, servers cannot distinguish if a protocol is hanging because the client is crashed or just slow. Preventing the race between a slow client and the recovering server requires extra mechanisms. Fortunately, if RIFL is used for RPCs for consistency protocols (e.g., transaction prepare RPCs), both the recovering server and the slow client can issue RPC requests without worrying about the race between them. RPCs will be executed only once and both the client and server will receive the execution results so that they can proceed on executing the consistency protocol.

4. **Move work from the critical path.** Inside consistency mechanisms such as transaction commits and replication, some steps don't need to be finished before returning to the application. Returning the execution result to an application before completing the steps can improve latency (it may also improve throughput by batching).

Table 1.1 summaries how the approaches mentioned above are applied to my work.

1.5 Dissertation Outline

Chapter 2 describes RIFL and RIFL-TX, which make remote operations linearizable when servers are internally linearizable (e.g., using consistent replication). Chapter 3 discusses some possible design simplifications if RIFL is implemented on other systems. Chapter 4 describes CURP, which removes the performance overhead of consistent primary-backup replication, which is a must-have for large-scale systems for availability. Chapter 5 discusses two extensions of CURP to different underlying systems, such as quorum-based consensus protocols.

Chapter 2

Making Remote Operations Linearizable

Consistency is one of the most important issues in the design of large-scale storage systems; it represents the degree to which a system's behavior is predictable, particularly in the face of concurrency and failures. Stronger forms of consistency make it easier to develop applications and reason about their correctness, but they may impact performance or scalability and they generally require greater degrees of fault tolerance. The strongest possible form of consistency in a concurrent system is *linearizability*, which was originally defined by Herlihy and Wing [34]. However, few large-scale storage systems implement linearizability today.

Almost all large-scale systems contain mechanisms that contribute to stronger consistency, such as reliable network protocols, automatic retry of failed operations, idempotent semantics for operations, and two-phase commit protocols. However, these techniques are not sufficient by themselves to ensure linearizability. They typically result in "at-least-once semantics," which means that a remote operation may be executed multiple times if a crash occurs during its execution. Re-execution of operations, even seemingly benign ones such as simple writes, violates linearizability and makes the system's behavior harder for developers to predict and manage.

In this chapter, we describe RIFL (Reusable Infrastructure for Linearizability), which is a mechanism for ensuring "exactly-once semantics" in large-scale systems. RIFL records the results of completed remote procedure calls (RPCs) durably; if an RPC is retried after it has completed, RIFL ensures that the original result is returned without re-executing the RPC. RIFL guarantees safety even in the face of server crashes and system reconfigurations such as data migration. As a result, RIFL makes it easy to turn non-linearizable operations into linearizable ones.

RIFL may fail to achieve exactly-once semantics under two circumstances. First, when a client crashes (without recovery later), some of its operations may be left not executed (§2.8). Second, when the network delays a client's messages extraordinarily long (e.g., more than 24 hours), then the RIFL's garbage collection mechanism may consider the client to be crashed and discard all metadata necessary for ensuring exactly-once semantics (§2.2).

RIFL is novel in several ways:

- **Reusable mechanism for exactly-once semantics**: RIFL is implemented as a general-purpose package, independent of any specific remote operation. As a result, it can be used in many different situations, and existing RPCs can be made linearizable with only a few additional lines of code. RIFL's architecture and most of its implementation are system-independent.

- **Reconfiguration tolerance**: large-scale systems migrate data from one server to another, either during crash recovery (to redistribute the possessions of a dead server) or during normal operation (to balance load). RIFL handles reconfiguration by associating RIFL metadata with particular objects and arranging for the metadata to migrate with the objects; this ensures that the appropriate metadata is available in the correct place to handle RPC retries.

- **Low latency**: RIFL is lightweight enough to be used even in ultra-low-latency systems such as RAMCloud [62] and FaRM [25], which have end-to-end RPC times as low as 5 µs.

- **Scalable**: RIFL has been designed to support clusters with tens of thousands of servers and one million or more clients. Scalability impacted the design of RIFL in several ways, including the mechanisms for generating unique RPC identifiers and for garbage-collecting metadata.

We have implemented RIFL in the RAMCloud storage system in order to evaluate its architecture and performance. Using RIFL, we were able to make existing operations such as writes and atomic increments linearizable with less than 20 additional lines of code per operation. We also used RIFL to construct a new multi-object transaction mechanism in RAMCloud; the use of RIFL significantly reduced the amount of mechanism that had to be built for transactions. The RAMCloud implementation of RIFL exhibits high performance: it adds less than 4% to the 13.5 µs base cost for writes, and simple distributed transactions execute in about 20 µs. RAMCloud transactions outperform H-Store [40] on the TPC-C benchmark, providing at least 10x lower latency and 1.35x–7x as much throughput.

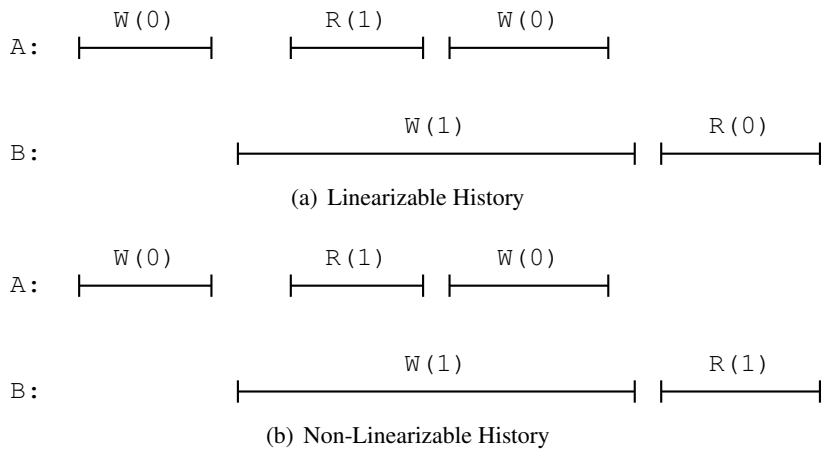

Figure 2.1: Examples of linearizable (a) and non-linearizable (b) histories for concurrent clients performing reads (`R()`) and writes (`W()`) on a single object, taken from [34]. Each row corresponds to a single client's history with time increasing to the right. The notation "`W(1)`" means that client B wrote the value 1 into the object, and "`R(0)`" means that client B read the value 0 of the object. Horizontal bars indicate the time duration of each operation.

2.1 Background and Goals

Linearizability is a safety property concerning the behavior of operations in a concurrent system. An operation is *linearizable* if it appears to occur instantaneously and exactly once at some point in time between its invocation and its completion. "Appears" means that it must not be possible for any client of the system, either the one initiating an operation or other clients operating concurrently, to observe contradictory behavior. Figure 2.1 shows examples of linearizable and non-linearizable operation histories. Linearizability is the strongest form of consistency for concurrent systems.

Early large-scale storage systems settled for weak consistency models in order to focus on scalability or partition-tolerance [67, 28, 23, 43], but newer systems have begun providing stronger forms of consistency [50, 8, 22]. They employ a variety of techniques, such as:

- Network protocols that ensure reliable delivery of request and response messages.
- Automatic retry of operations after server crashes, so that all operations are eventually completed.
- Operations with idempotent semantics, so that repeated executions of an operation produce the same result as a single execution.

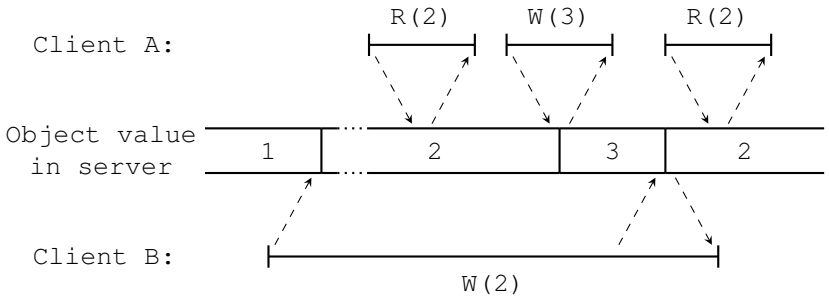

Figure 2.2: Non-linearizable behavior caused by crash recovery. In this example, the server completes a write from Client B but crashes before responding. "..." indicates that the server is unavailable due to a crash. After the server restarts, Client B reissues the write, but meanwhile Client A has written a different value. As a result, Client A observes the value 2 being written twice.

- Two-phase commit and/or consensus protocols [32, 44], which ensure atomic updates of data on different servers.

However, few large-scale systems actually implement linearizability, and the above techniques are insufficient by themselves. For example, Figure 2.2 shows how retrying an idempotent operation after a server crash can result in non-linearizable behavior. The problem with most distributed systems is that they implement *at-least-once semantics*. If a client issues a request but fails to receive a response, it retries the operation. However, it is possible that the first request actually completed and the server crashed before sending a response. In this situation the retry causes the operation to be performed twice, which violates linearizability.

In order for a system to provide linearizable behavior, it must implement *exactly-once semantics*. To do this, the system must detect when an incoming request is a retry of a request that already completed. When this occurs, the server must not re-execute the operation. However, it must still return whatever results were generated by the earlier execution, since the client has not yet received them.

Some storage systems, such as H-Store [40] and FaRM [25], implement strongly consistent operations in the servers but they don't provide exactly-once semantics for clients: after a server crash, a client may not be able to determine whether a transaction completed. As a result, these systems do not guarantee linearizability (linearizability must be implemented on top of the transaction mechanism, as discussed in Section 2.7).

The overall goal for RIFL is to implement exactly-once semantics, thereby filling in the missing

piece for linearizability. Furthermore, we designed RIFL as general-purpose infrastructure, independent of any particular linearizable operation. Our hope in doing this was to make it easy to implement a variety of linearizable operations, ranging from complex transactions down to simple operations such as writes or atomic increments. Although we implemented RIFL in a single system, we designed the mechanism to be applicable for a variety of systems.

We designed RIFL for use in the most demanding datacenter applications, which created two additional goals: scalability and low latency. Large-scale Web applications today can include tens of thousands of storage servers and an even larger number of clients, which are typically servers in the same datacenter that handle incoming Web requests. We can envision applications in the near future with one million or more client threads, each communicating with all of the servers. This means that the per-client state stored on each server must be small. Large-scale systems reconfigure themselves frequently, such as migrating data after server crashes; this means that saved results must migrate as well. We also wanted RIFL to be suitable for high-performance applications such as RAMCloud [62] and FaRM [25], which keep their data in DRAM. These systems offer latencies as low as 5 µs end-to-end for remote operations; the overheads introduced by RIFL must not significantly impact these latencies. For example, RIFL must piggyback its metadata on existing messages whenever possible, so as not to introduce additional communication.

RIFL assumes the existence of a remote procedure call (RPC) mechanism in the underlying system. Linearizability requires a request-response protocol of some sort, rather than asynchronous messages, since there is no way of knowing that an operation completed without receiving a response. RIFL also assumes automatic retries in the RPC system, to produce at-least-once semantics. An RPC must not abort and return an error to higher-level software after a server crash, since it will then be indeterminate whether the operation has completed.

Given these underlying mechanisms, RIFL ensures that RPCs will be executed exactly once as long as clients don't crash (see Section 2.8) and servers can store RIFL's metadata reliably.

2.2 RIFL Architecture

In order to implement exactly-once semantics, RIFL must solve four overall problems: RPC identification, completion record durability, retry rendezvous, and garbage collection. This section discusses these issues and introduces the key techniques RIFL uses to deal with them; Section 2.3 describes the mechanisms in more detail.

In order to detect redundant RPCs, each RPC must have a unique identifier, which is present

in all invocations of that RPC. In RIFL, RPC identifiers are assigned by clients and they consist of two parts: a 64-bit unique identifier for the client and a 64-bit sequence number allocated by that client. This requires a system-wide mechanism for allocating unique client identifiers. RIFL manages client identifiers with a lease mechanism described later in this section.

The second overall problem is completion record durability. Whenever an operation completes, a record of its completion must be stored durably. This *completion record* must include the RPC identifier as well as any results that are returned to the client. Furthermore, the completion record must be created atomically with the mutations of the operation, and it must have similar durability properties. It must not be possible for an operation to complete without a visible completion record, or vice versa. RIFL assumes that the underlying system provides durable storage for completion records.

The third problem is retry rendezvous: if an RPC completes and is then retried at a later time, the retries must find the completion record to avoid re-executing the operation. However, in a large-scale system the retry may not be sent to the same server as the original request. For example, many systems migrate data after a server crash, transferring ownership of the crashed server's data to one or more other servers; once crash recovery completes, RPCs will be reissued to whichever server now stores the relevant data. RIFL must ensure that the completion record finds its way to the server that handles retries and that the server receives the completion record before any retries arrive. Retry rendezvous also creates issues for distributed operations that involve multiple servers, such as multi-object transactions: which server(s) should store the completion record?

RIFL uses a single principle to handle both migration and distributed operations. Each operation is associated with a particular object in the underlying system, and the completion record is stored wherever that object is stored. If the object migrates, then the completion record must move with it. All retries must necessarily involve the same object(s), so they will discover the completion record. If an operation involves more than one object, one of them is chosen as a distinguished object for that operation, and the completion record is stored with that object. The distinguished object must be chosen in an unambiguous fashion, so that retries use the same distinguished object as the original request.

The fourth overall problem for RIFL is garbage collection: eventually, RIFL must reclaim the storage used for completion records. A completion record cannot be reclaimed until it is certain that the corresponding request will never be retried. This can happen in two ways. First, once the client has received a response, it will never retry the request. Clients provide acknowledgments to the servers about which requests have successfully completed, and RIFL uses the acknowledgments to

newSequenceNum() → *sequenceNumber*
Assigns and returns a unique 64-bit sequence number for a new RPC. Sequence numbers are assigned in increasing integer order.

firstIncomplete() → *sequenceNumber*
Returns the lowest sequence number for which an RPC response has not yet been received.

rpcCompleted(*sequenceNumber*)
Invoked when a response has been received for *sequenceNumber*.

Figure 2.3: The API of the RequestTracker module, which manages sequence numbers for a given client machine.

delete completion records. Completion records can also be garbage collected when a client crashes. In this situation the client will not provide acknowledgments, so RIFL must detect the client's crash in order to clean up its completion records.

Detecting and handling client crashes creates additional complications for garbage collection. One possible approach is to associate an expiration time with each completion record and delete the completion record if it still exists when the time expires; presumably the client must have crashed if it hasn't completed the RPC by then. However, this approach could result in undetectable double-execution if a client is partitioned or disabled for a period longer than the expiration time and then retries a completed RPC.

We chose to use a different approach to client crashes, which ensures that the system will detect any situation where linearizability is at risk, no matter how rare. RIFL's solution is based on leases [31]: each client has a private lease that it must renew regularly, and the identifier for the lease serves as the client's unique identifier in RPCs. If a client fails to renew the lease, then RIFL assumes the client has crashed and garbage-collects its completion records. RIFL checks the lease validity during each RPC, which ensures that a client cannot retry an RPC after its completion record has been deleted. If a live client fails to renew its lease for some reason (e.g., a prolonged network partition), the client will not be able to figure out the completion of the RPCs issued before the network partition. However, the lease mechanism can notify the client of the potential duplicate executions (attempts by the client to reissue RPCs will fail with a special error code).

Managing leases for a large number of clients introduces additional scalability issues. For example, one million clients could create a significant amount of lease renewal traffic for a centralized lease manager. In addition, lease information must be durable so that it survives server failure; this increases the cost of managing leases. The lease timeout must be relatively long, in order to reduce

getClientId() → *clientId*
Returns a unique 64-bit identifier for this client; if a lease does not already exist, initializes a new one. Used only on clients.

checkAlive(*clientId*) → {ALIVE or EXPIRED}
Returns an indication of whether the given client's lease has expired. Used only on servers.

Figure 2.4: The API of the LeaseManager module, which runs on both clients and servers and communicates with the lease server to keep track of leases. On the client side, LeaseManager automatically creates a lease on the first call to `getClientId` and renews it in the background. The unique identifier for each client is determined by its lease and is valid only as long as the lease is alive.

checkDuplicate(*clientId*, *sequenceNumber*) → {NEW, COMPLETED, IN_PROGRESS, or STALE}, *completionRecord*
Returns the state of the RPC given by *clientId* and *sequenceNumber*. NEW means that this is the first time this RPC has been seen; internal state is updated to indicate that the RPC is now in progress. COMPLETED means that the RPC has already completed; a reference to the completion record is also returned. IN_PROGRESS means that another execution of the RPC is already underway but not yet completed. STALE means that the RPC has already completed, and furthermore the client has acknowledged receiving the result; there may no longer be a completion record available.

recordCompletion(*clientId*, *sequenceNumber*, *completionRecord*)
This method is invoked just before responding to an RPC; *completionRecord* is a reference to the completion record for this RPC, which will be returned by future calls to `checkDuplicate` for the RPC.

processAck(*clientId*, *firstIncomplete*)
Called to indicate that *clientId* has received responses for all RPCs with sequence numbers less than *firstIncomplete*. Completion records and other state information for those RPCs will be discarded.

recoverRecord(*clientId*, *sequenceNumber*, *completionRecord*)
This method is a variant of recordCompletion method and invoked during the recovery or migration. It reconstructs the mapping from RPC identifiers to *completionRecord*s.

Figure 2.5: The API of the ResultTracker module, which runs on servers to keep track of all RPCs for which results are not known to have been received by clients.

the likelihood of lease expiration due to temporary disruptions in communication, but this increases the amount of state that servers need to retain.

2.3 Design Details

The previous section introduced the major problems that RIFL must solve and described some of the key elements of RIFL's solutions. This section fills in the details of the RIFL design, focusing on the aspects that will be the same in any system using RIFL. System-specific details are discussed in Section 2.4.

RIFL appears to the rest of the system as three modules. The first, RequestTracker, runs on client machines to manage sequence numbers for outstanding RPCs (Figure 2.3). The second module, LeaseManager, runs on both clients and servers to manage client leases (Figure 2.4). On clients, LeaseManager creates and renews the client's lease, which also yields a unique identifier for the client. On servers, LeaseManager detects the expiration of client leases. The third module, ResultTracker, runs only on servers: it keeps track of currently executing RPCs and manages the completion records for RPCs that have finished (Figure 2.5).

2.3.1 Lifetime of an RPC

When a client initiates a new RPC, it forms a unique identifier for that RPC by combining the client's unique identifier with a new sequence number provided by RequestTracker. It includes this identifier in the RPC and must record the identifier so that it can use the same identifier in any subsequent retries of the RPC.

When a server receives an RPC, it must check to see if the RPC represents a retry of a previously completed RPC. To do this, it calls ResultTracker's `checkDuplicate` method and does one of four things based on the result:

- In the normal case, this is a new RPC (one whose identifier has never been seen previously); `checkDuplicate` records that the RPC is underway and returns NEW. The server then proceeds to execute the RPC.

- If this RPC has already completed previously, then `checkDuplicate` returns COMPLETED. It also returns a reference to the RPC's completion record; the server uses this to return a response to the client without re-executing the RPC.

- Is also possible that the RPC is currently being executed but has not completed. In this case, `checkDuplicate` returns IN_PROGRESS; depending on the underlying RPC system, the server may discard the incoming request or respond to the client with an indication that execution is still in progress.

- Finally, it is possible for a stale retry to arrive even after a client has received a response to the RPC and acknowledged receipt to the server, and the server has garbage-collected the completion record. This indicates either a client error or a long-delayed network packet arriving out of order. In this case `checkDuplicate` returns STALE and the server returns an error indication to the client.

In the normal case of a new RPC, the server executes the operation(s) indicated in the RPC. In addition, it must create a completion record containing the following information:

- The unique identifier for the RPC.

- An object identifier, which uniquely identifies the storage location for a particular object in the underlying system, such as a key in a key-value store. The specific format of this identifier will vary from system to system. This field is used to ensure that the completion record migrates along with the specified object.

- The result that should be returned to the client for this RPC. The result will vary from operation to operation: a write operation might return a version number for the new version of the object; an atomic increment operation might return the new value; some operations may have no result.

The underlying system must provide a mechanism for storing completion records. The completion record must be made durable in an atomic fashion along with any side effects of the operation, and durability must be assured before the RPC returns. The exact mechanism for doing this will vary from system to system, but many large-scale storage systems include a log of some sort; in this case, the operation side effects and completion record can be appended to the log atomically.

Once the completion record has been made durable, the server invokes the `recordCompletion` method of ResultTracker and passes it a reference to the completion record. The format of the reference is system-specific and opaque to the ResultTracker module. The reference must provide enough information to locate the completion record later, such as a location in the log. ResultTracker associates this reference with the RPC identifier, so that it can return the reference in future calls to

`checkDuplicate`. Once `recordCompletion` has returned, the server can return the RPC's result to the client.

When the client receives a response to an RPC, it invokes the `rpcCompleted` method of RequestTracker. This allows RequestTracker to maintain accurate information about which RPCs are still in progress.

If a client fails to receive a response to an RPC or detects a server crash via some other mechanism, then it is free to reissue the RPC. If data has migrated, the reissued RPC may be sent to a different server than the original attempt (the mechanisms for managing and detecting migration are implemented by the underlying system). However, the client must use the same unique identifier for the reissued RPC that it used for the original attempt. When the server receives this RPC, it will invoke `checkDuplicate` as described above and either execute the RPC or return the result of a previous execution.

2.3.2 Garbage collection

RIFL uses sequence numbers to detect when completion records can be safely reclaimed. The RequestTracker module in each client keeps track of that client's "active" RPC sequence numbers (those that have been returned by `newSequenceNum` but have not yet been passed to `rpcCompleted`). The `firstIncomplete` method of RequestTracker returns the smallest of these active sequence numbers, and this value is included in every outgoing RPC. When an RPC arrives on a server, the server passes that sequence number to the ResultTracker module's `processAck` method, which then deletes all of its internal state for RPCs from that client with smaller sequence numbers. `ProcessAck` also invokes the underlying system to reclaim the durable storage occupied by the completion records.

Using a single sequence number to pass garbage collection information between clients and servers is convenient because the sequence number occupies a small fixed-size space in outgoing RPCs and it can be processed quickly on servers. However, it forces a trade-off between RPC concurrency on clients and space utilization on servers. If a client issues an RPC that takes a long time to complete, then concurrently issues other RPCs, the completion records for the later RPCs cannot be garbage collected until the stalled RPC completes. This could result in an unbounded amount of state accumulating on servers.

In order to limit the amount of state on servers, which is important for scalability, RIFL sets an upper limit on the number of non-garbage-collectible RPCs a given client may have at one time. This number is currently set at 512. Once this limit is reached, no additional RPCs may be issued

until the oldest outstanding RPC has completed. The limit is high enough to allow considerable concurrency of RPCs by a single client, while also limiting worst-case server memory utilization.

The garbage collection mechanism could be implemented using a more granular approach that allows information for newer sequence numbers to be deleted while retaining information for older sequence numbers that have not completed. However, we were concerned that this might create additional complexity that impacts the latency of RPCs; as a result, we have deferred such an approach until there is evidence that it is needed.

2.3.3 Lease management

RIFL uses leases to allocate unique client identifiers and detect client crashes. Leases are managed by a centralized lease server, which records information about active leases on a stable storage system such as ZooKeeper [35] so it can be recovered after crashes. When a client issues its first RPC and invokes the `getClientId` method of LeaseManager (Figure 2.4), LeaseManager contacts the lease server to allocate a new lease. LeaseManager automatically renews the lease. The identifier for this lease serves as the unique identifier for the client.

The lease mechanism presents two challenges for RIFL's scalability and latency: renewal overhead and validation overhead. The first challenge is the overhead for lease renewal on the lease server, especially if the system approaches our target size of one million clients; the problem becomes even more severe if each renewal requires operations on stable storage. To reduce the renewal overhead, the lease server keeps all of the lease expiration times in memory, and it does not update stable storage when leases are renewed. Only the existence of leases is recorded durably, not their expiration times. To compensate for the lack of durable lease expiration times, the lease server automatically renews all leases whenever it reconstructs its in-memory data from stable storage after a restart.

The second problem with leases is that a server must validate a client's lease during the execution of each RPC, so it can reject RPCs with expired leases. The simplest way to implement this is for the server to contact the lease server during each request in order to validate the lease, but this would create unacceptable overheads both for the lease server and for the RPC's server.

Instead, RIFL implements a fast validation mechanism that allows servers to validate leases quickly using information received from clients during normal RPCs; a server only needs to contact the lease server if a fast validation fails. The lease server implements a *cluster clock*, which is a time value that increases monotonically at the rate of the lease server's real-time clock and is durable across lease server crashes. Neither clients nor servers estimate the current value of the cluster clock

at the lease server; they only keep the maximum of the observed cluster clock values. Clients get new cluster clock values from the lease server during lease creation or renewal. Every client request incorporates the last observed cluster clock value and the current lease expiration time. For each request, a server updates its cluster clock value if the one from the client is more recent, and the server checks if the given lease expiration time is past its cluster clock value. If the expiration time is not past the cluster clock value, the server assumes that the lease has not expired and executes the client request. This fast validation is efficient since it solely uses the information given with client requests.

However, fast validation must deal with two potential problems. The first problem is that a significantly delayed client request may fail fast validation even if the client's lease is live at the lease server. To prevent discarding completion records for a live lease, servers must contact the lease manager to validate any lease that failed the fast validation. If the lease server confirms that the lease has expired, then the server returns an error to the client without executing the operation. During garbage collection, servers must also confirm with the lease server before discarding completion records for clients whose leases failed the fast validations. The second problem is that, during recovery or migration, the new server may accept requests from a client whose completion records were discarded due to the lease expiration. This problem can occur when the new server's cluster clock value is staler than the original server's value so that the client request with an expired lease can pass the fast validation at the new server. To prevent this scenario, a new server must obtain the up-to-date cluster clock value from the lease server before finishing recovery or migration.

2.3.4 Migration

As discussed in Section 2.2, a large-scale system may migrate data from one server to another, either to balance load during normal operation or during crash recovery. The details of this are system-specific, but in order for RIFL to function correctly, the system must always migrate each completion record to the same machine that stores the object identified by that completion record. When a completion record arrives on a new server, the underlying system must call the `recoverRecord` method of ResultTracker so that it can reconstruct its in-memory metadata that maps from RPC identifiers to completion records.

2.4 Implementation

In order to evaluate RIFL, we have implemented it in RAMCloud [62], a key-value store that keeps all data in DRAM. RAMCloud has several properties that make it an attractive target for RIFL. It is already designed for large scale and it offers low latency (small remote reads take 4.7 µs end to end, small durable writes take 13.5 µs); the RAMCloud implementation allows us to evaluate whether RIFL can be used for large-scale low-latency applications. In addition, RAMCloud already implemented at-least-once semantics, so RIFL provides everything needed to achieve full linearizability. Finally, RAMCloud is available in open-source form [3].

Previous sections have described the system-independent aspects of RIFL; this section describes the RAMCloud-specific facilities that had to be created as part of implementing RIFL. All of the changes to RAMCloud were localized and small.

RAMCloud uses a unified log-structured approach for managing data both in DRAM and on secondary storage [62], and it uses small amounts of nonvolatile memory to perform durable replication quickly. RIFL stores its completion records as a new type of entry in the RAMCloud log; we extended the logging mechanism to ensure that completion records can be written atomically with other records such as new values for objects. A reference to a completion record (as passed to and from ResultTracker) consists of its address in the in-memory copy of the log.

Garbage collection of completion records is handled by the log cleaner using its normal mechanism. The cleaner operates by scanning a region of log entries and calling a type-specific method for each entry to determine whether the entry is still live; live entries are copied forward in the log, then the entire region is reclaimed. A log entry containing a completion record is live if ResultTracker still retains the mapping from the completion record's RPC id to the completion record.

In RAMCloud, each object is assigned to a server based on its table identifier and a hash of its key; RIFL uses these two 64-bit values as the object identifier in completion records. We made two small modifications to RAMCloud's migration and crash recovery code so that (a) completion records are sent to the correct server during crash recovery (b) when a completion record arrives on a new server, the `recoverRecord` method of ResultTracker is invoked to incorporate that completion record into its metadata.

In RAMCloud the lease server is integrated into the cluster coordinator and uses the same ZooKeeper instance for storing lease information that the coordinator uses for its other metadata.

Once RIFL was implemented in RAMCloud, we used it to create a variety of linearizable operations. We first converted a few simple operations such as write, conditional write, and atomic

read(*tableId*, *key*) → *value*
Returns the *value* of the object for *tableId* and *key*.

write(*tableId*, *key*, *value*)
Updates the *value* of the object for *tableId* and *key*.

delete(*tableId*, *key*)
Removes the object for *tableId* and *key*.

commit() → {COMMITTED or ABORTED}
Tries to commit the current transaction and returns the outcome. After this method is invoked, calling read, write or delete methods is not allowed.

Figure 2.6: The API of the transaction object, which is created by clients for every transaction. Clients can optimistically perform operations and then attempt to commit the transaction by invoking `commit`. Each Transaction object represents a single transaction attempt. Transaction objects should be discarded after the transaction commits or aborts.

increment to be linearizable. Each of these operations affects only a single object on a single server, so the code modifications followed naturally from the description in Section 2.3.

We also used RIFL to implement a new multi-object transaction mechanism that was not previously present in RAMCloud. This mechanism is described in the following section.

2.5 Implementing Transactions with RIFL

This section describes RIFL-TX, which is a new RIFL-based distributed transaction mechanism in RAMCloud. RIFL-TX is a more complex use case for RIFL, since transactions involve multiple objects on different servers. We found that RIFL significantly simplified the implementation of transactions, and the resulting mechanism offers high performance, both in absolute terms and relative to other systems.

The two-phase commit protocol for RIFL-TX is based on Sinfonia [6]; we chose this approach because Sinfonia offers the lowest possible latency for distributed transactions. However, we did not need to implement all of Sinfonia's mechanisms because RIFL made some of them unnecessary. The description below focuses on the overall mechanism and its use of RIFL.

2.5.1 APIs

The application-visible API for RIFL-TX is based on a new Transaction class. To use the transaction mechanism, an application creates a Transaction object, uses it to read, write, and delete RAMCloud

objects, then invokes a commit operation, which will succeed or abort (APIs are in Figure 2.6). If the commit succeeds, it means that all of the operations were executed atomically as a single linearizable operation. If the commit aborts, then none of the transaction's mutations were applied to the key-value store or were in any way visible to other clients. The system may abort the commit for a variety of reasons, including data conflicts, busy locks, and client crashes. If an application wishes to ensure that a transaction succeeds, it must retry aborted transactions. With this API, RAMCloud provides ACID transactions with strict serializability [74] using optimistic concurrency control [42].

2.5.2 The commit operation

The Transaction object defers all updates to the key-value store until commit is invoked. When reads are requested, the Transaction reads from the key-value store and caches the values along with the version number for each RAMCloud object, which is incremented every time the object's value changes. When writes and deletes are requested, the Transaction records them for execution later, without modifying the key-value store. When commit is invoked, the Transaction applies all of the accumulated mutations in an atomic fashion.

The Transaction object implements its commit method using a single internal operation that is similar to a Sinfonia mini-transaction; we will use the term "Commit" for this operation. Commit is a distributed operation involving one or more objects, each of which could be stored on a different server. The arguments to Commit consist of a list of objects, with the following information for each object:

- The table identifier and key for the object.
- The operation to execute on the object: read, write, or delete.
- A new value for the object, in the case of writes.
- The expected version number for the object (or "any").

Commit must atomically verify that each object has the required version number, then apply all of the write and delete operations. If any of the version checks fail, it means there has been a conflicting write to the object, so the commit aborts and no updates occur.

2.5.3 Client-driven two-phase commit

Commit is implemented using a two-phase protocol where the client serves as coordinator. In the first phase, the client issues one `prepare` RPC for each object involved in the transaction (see

prepare(*tableId*, *key*, *version*, *operation(READ, WRITE, DELETE)*, *newValue*, *rpcId*, *firstIncomplete*, *leaseInfo*, *allObjects*) → {PREPARED, ABORT, COMMITTED},
Sent from clients to participants for the first stage of commit: verifies that the object given by *tableId* and *key* is not already locked by another transaction and has a version number matching *version*; if so, locks the object, writes a durable record describing the lock as well as *operation* and *newValue*, and returns PREPARED; otherwise returns ABORT. On a special occasion where all objects involved in the transaction reside in a single server, the server may process the decision phase and return COMMITTED. *operation* specifies the operation that will eventually be performed on the object and *newValue* is the new object value for writes. *rpcId*, *firstIncomplete*, and *leaseInfo* are used by RIFL for at-most-once semantics. *allObjects* describes all of the objects in the transaction (*tableId*, *keyHash*, and *rpcId* for each).

decision(*rpcId*, *action(COMMIT, ABORT)*)
Sent from clients or recovery coordinators to participants for the second stage of two-stage commit; *rpcId* indicates a particular object (must match the *rpcId* of a previous `prepare`). If *action* is COMMIT, then the operation specified in the corresponding `prepare` is performed. In any case, the lock is removed and the durable lock record is deleted.

startRecovery(*allObjects*)
Sent from participants to the recovery coordinator to start recovery. *allObjects* specifies all of the objects of the transaction (same format as for `prepare`).

requestAbort(*rpcId*) → {PREPARED, ABORT},
Sent from the recovery coordinator to participants during the first phase of crash recovery. Returns PREPARED if a completion record indicates a prior PREPARED response for *rpcId*, otherwise returns ABORT.

Figure 2.7: The APIs for the RPCs used to implement the commit protocol for RIFL-TX.

Figure 2.7). The server storing the object (called a *participant*) locks the object and checks its version number. If it doesn't match the desired version then the participant unlocks the object and returns ABORT; it also returns ABORT if the object was already locked by another transaction. Otherwise the participant stores information about the lock in a transaction lock table and creates a durable record of the lock in its log. It then returns PREPARED to the client. The client issues all of the `prepare` RPCs concurrently and it batches requests to the same participant.

If all of the `prepare` RPCs return PREPARED, then the commit will succeed; if any of the `prepare` RPCs return ABORT, then the transaction will abort. In either case, the client then enters the second phase, where it issues a `decision` RPC for each object. The participant for each object checks whether the RPC indicates "commit" or "abort". If the decision was to commit, it applies the mutation for that object, if any. Then, whether committing or aborting, it removes the lock table entry and adds a RAMCloud log record that indicates the lock record is no longer valid and should be ignored during recovery or migration.

The transaction is effectively committed once a durable lock record has been written for each of the objects. At this point, regardless of the client's actions, the mutations will eventually be applied (the details will be discussed below). The Transaction object's commit method can return as soon as positive responses have been received from all of the servers; the `decision` RPCs can be issued in the background. Thus, the latency for Commit consists of one round-trip RPC time (more precisely, the time for a concurrent collection of RPCs to all of the participants) plus the time for a durable write on each participant. In RAMCloud, a durable write is implemented with concurrent RPCs that replicate log records in nonvolatile memory on three backup servers. Chapter 4 will present a new replication mechanism that avoids the latency increase caused by the durable write. Except for the durable write latency (which is avoidable), the RAMCloud transaction has the lowest possible time for a transaction commit; we chose the client-driven approach because it is at least one half round-trip faster than approaches that use a server as the transaction coordinator.

The commit protocol is optimized for two special cases. The first consists of transactions whose objects all reside on a single participant. When a participant receives a `prepare` request, it checks to see if it owns all of the objects in the transaction. If so, it executes both the prepare and decision operations and returns a special COMMITTED status to the client; the client then skips its decision phase and doesn't send `decision` RPCs. The server propagates log records to backups only once, after the decision phase. Since the optimized form does not return to the client until after the decision operation has completed, its latency is slightly higher than it would be without the optimization, but the optimization improves throughput significantly and eliminates the need for clients to invoke

`decision` RPCs. (see Section 2.6).

The second optimization is for read-only transactions. If a transaction does not modify any objects, the client indicates this in each `prepare` RPC. In this case, the participant simply checks to make sure the object is not locked and then verifies its version number; there is no need to acquire a lock or record any durable data. As with single-server transactions, clients need not issue `decision` RPCs.

2.5.4 Client crashes

If the client crashes before completing all of the `decision` RPCs, then a server must complete the process on its behalf. RIFL-TX uses a mechanism similar to Sinfonia for this. If the client is suspected to have crashed, the participant for the transaction's first object acts as *recovery coordinator*. The recovery coordinator executes a two-phase protocol similar to that of the client, except that its goal is to abort the transaction unless it has already committed (in general, there may not be enough information to complete an incomplete transaction, since the client may have crashed before issuing all of the `prepare` RPCs). In the first phase the recovery coordinator issues a `requestAbort` RPC for each object, whose participant will agree to abort unless it has already accepted a `prepare` for the object. If `requestAbort` returns PREPARED for every object in the transaction, then the transaction has already committed. Otherwise, the recovery coordinator will abort the transaction. In either case, the recovery coordinator then sends `decision` RPCs for each object.

Transaction recovery is initiated using a timeout mechanism. Whenever a participant creates an entry in its lock table, it starts a timer. If the timer expires before the participant has received a `decision`, then the participant sends a `startRecovery` RPC to the recovery coordinator.

In order to provide enough information for crash recovery, the client includes identifiers for all objects in the transaction as part of each `prepare`. This information serves two purposes. First, it allows each participant to identify the recovery coordinator (the server for the first object in the list). Second, the object list is needed by the recovery coordinator to identify participants for its `requestAbort` and `decision` RPCs. The recovery coordinator may not have received a `prepare` if the client crashed, so when a participant invokes `startRecovery` it includes the list of objects that it received in its `prepare`.

2.5.5 RIFL's role in transactions

The issue of exactly-once semantics arises in several places in the RAMCloud transaction mechanism. For example, a server may crash after completing a `prepare` but before responding; its objects and lock table will migrate to another server during crash recovery, and the client will eventually retry with that server. In addition, a client that is presumed dead may not actually be dead, so it could send `prepare` RPCs at the same time the recovery coordinator is sending `requestAbort` RPCs for the same objects: once one of these RPCs has reached a decision for an object, the other RPC must see the same decision. Finally, a "dead" client may wake up after recovery is complete and go through its two-phase protocol; participants must not re-execute `prepare` RPCs for which `requestAbort` RPCs were executed during recovery, and the client must reach the same overall decision about whether the transaction committed.

All of these situations are handled by using RIFL for the `prepare` and `requestAbort` RPCs. Each `prepare` is treated as a separate linearizable operation. For example, when a client begins the Commit process it uses RIFL to assign a unique identifier for each `prepare`, and the participant logs a completion record for the RPC atomically with the lock record. If the version check fails, the participant will not create a lock record, but it will still create a completion record indicating that it rejected the RPC. The participant also uses the RPC identifier in its lock table as a unique identifier for a particular object participating in a particular transaction. If a client retries a `prepare` (e.g. because of a participant crash), the completion record ensures exactly-once semantics. Each object can potentially migrate independently during crash recovery, but the per-object completion records will follow them.

When a `requestAbort` is issued during client crash recovery, the recovery coordinator uses the *same* RPC identifier that was used by the corresponding `prepare` (the identifiers are included in `prepare` RPCs along with object identifiers, and they are passed to the recovery coordinator in `startRecovery` RPCs). This allows a participant handling a `requestAbort` to determine whether it already processed a `prepare` for that object; if there is no pre-existing completion record, then the server creates a new one indicating that it has agreed to abort the transaction. The shared RPC identifier ensures that races are resolved properly: if a `prepare` arrives from the client later, it will find the `requestAbort` completion record and return ABORT to the client.

Completion records are not needed for `decision` RPCs, since redundant executions can be detected using the lock table. If a `decision` is retried, there will no longer exist a lock table entry for the object. This indicates that the current `decision` is redundant, so the participant returns immediately without taking any action.

2.5.6 Garbage collection

Garbage collection is more complex for distributed transactions than for single-object operations because it requires knowledge of distributed state. In Sinfonia this complexity is reflected in two additional protocols: a special mechanism to collect ids for completed transactions and exchange them among servers in batches, and a separate epoch-based mechanism for garbage-collecting information about transaction aborts induced during crash recovery. Our use of RIFL in RIFL-TX allowed us to avoid both of these additional protocols and ensure proper garbage collection with only a few small additions.

In a distributed transaction, none of the completion records for the `prepare` RPCs can be deleted until the `decision` RPCs have been processed for all of the transaction's objects (otherwise the transaction could be re-executed with a different result). This problem led to the transaction id exchange mechanism in Sinfonia. Our transaction mechanism uses a much simpler solution: a client does not call `rpcCompleted` for any of its `prepare` RPCs until it has received responses for all of the `decision` RPCs.

The epoch mechanism in Sinfonia was required to garbage-collect transactions aborted during client crash recovery; in RAMCloud transactions the client leases already handle this situation. If a client crashes during a transaction, then it will never call `rpcCompleted` for its `prepare` RPCs, so servers will retain completion records until the client lease expires. The timers for transaction recovery are considerably shorter than the lease timeout; this ensures that the completion records will still be available for the recovery mechanism described in Section 2.5.4. If the client has not really crashed and finishes executing the transaction protocol after recovery has occurred, the completion records will still be available, so the client will observe the commit/abort decision made during recovery. It will then call `rpcCompleted` and the completion records can be deleted without waiting for lease expiration.

RIFL's completion records provide a clean separation of information with different lifetimes, so different kinds of information can be garbage-collected independently. For example, the completion record for a `prepare` is distinct from the lock record created by that RPC. The lock record's lifetime is determined by the transaction (the record is deleted as soon as a `decision` RPC is received), but the completion record must be retained until the client acknowledges receipt of the `prepare` result or its lease expires, which could be considerably later.

CPU	Xeon X3470 (4x2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
Flash Disks	2x Samsung 850 PRO SSDs (256 GB)
NIC	Mellanox ConnectX-2 Infiniband HCA
Switch	Mellanox SX6036 (4X FDR)

Table 2.1: The server hardware configuration used for benchmarking. All nodes ran Linux 2.6.32 and were connected to a two-level Infiniband fabric with full bisection bandwidth; the NICs support kernel-bypass. The Infiniband fabric supports 32 Gbps bandwidth, but PCI Express limits the nodes to about 24 Gbps.

2.5.7 Server crash recovery

If a server crashes during the transaction commit protocol, its objects will be reconstructed on one or more other servers. The log records for locks and `prepare` completions will migrate with the corresponding objects and the new servers will use this data to populate lock tables and ResultTrackers, so the two-phase commit protocol can be completed in the normal way.

If the recovery coordinator crashes during transaction recovery, its data will be reconstructed on another server and transaction recovery will eventually start again. The completion records ensure that any past decisions are visible to future recovery coordinators.

2.6 Evaluation

We evaluated the RAMCloud implementation of RIFL to answer the following questions:

- What is RIFL's impact on latency and throughput?
- Does RIFL limit scalability?
- How hard is it to use RIFL to implement linearizability?
- How does the performance of transactions implemented with RIFL compare to other state-of-the-art systems?

All performance evaluations were conducted on a cluster of machines with the specifications shown in Table 2.1. All measurements were made using Infinband networking. Unless otherwise indicated, all RAMCloud measurements were made using RAMCloud's fastest transport, which bypasses the kernel to communicate directly with NICs. In a few cases we used a different transport based on

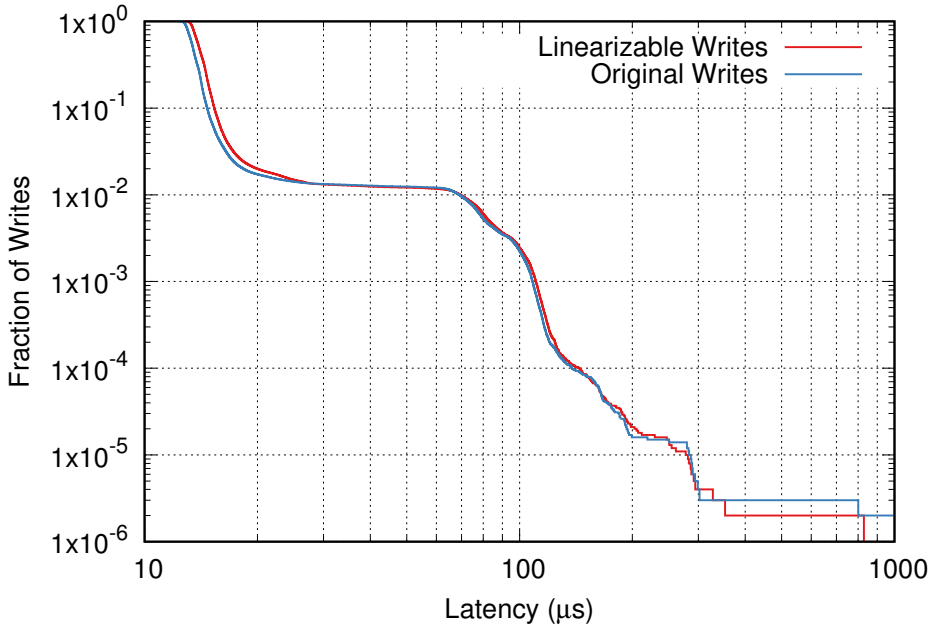

Figure 2.8: Complementary cumulative distribution of latency for 100B random RAMCloud write operations with and without RIFL. Writes were issued sequentially by a single client to a single server in a 4-server cluster. A point (x,y) indicates that y of the 1M measured writes took at least x µs to complete. "Original Write" refers to the base RAMCloud system before adding RIFL. The median latency for linearizable writes was 14.0 µs vs. 13.5 µs for the original system.

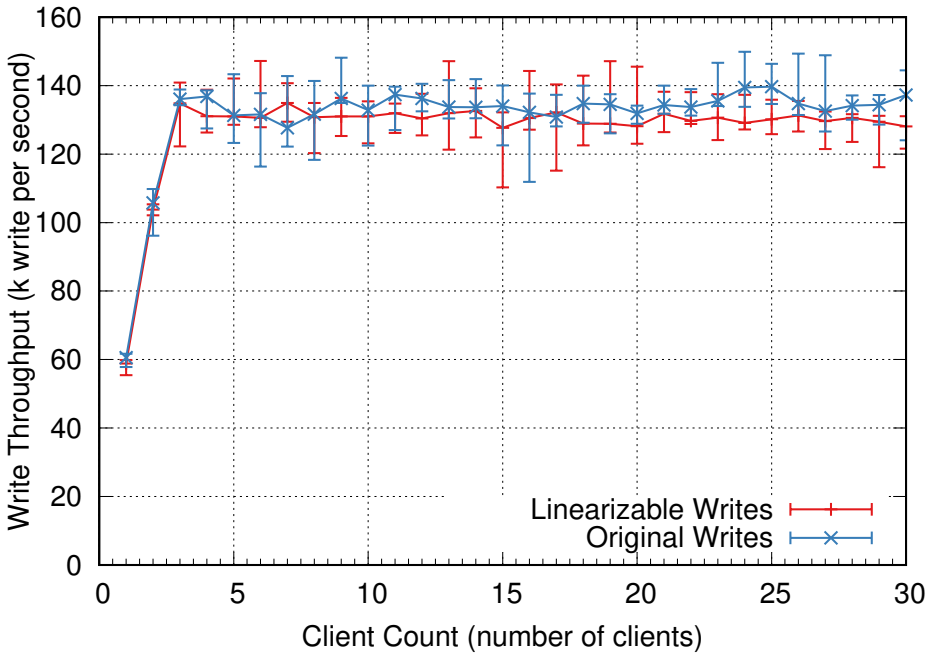

Figure 2.9: The aggregate throughput for one server serving 100B RAMCloud writes with and without RIFL, as a function of the number of clients. Each client repeatedly issued random writes back to back to a single server in a 4-server cluster. "Original Write" refers to the base RAMCloud system before adding RIFL. Each experiment was run 5 times. The data points show the median values; error bars show min and max.

TCP implemented in the kernel (packets were still delivered via Infiniband). RAMCloud servers were configured to use 3-way replication. The log cleaner did not run in any of these experiments; in a production system cleaning overheads could reduce throughput by as much as 25% from the numbers reported here, depending on memory utilization and workload (see Figure 6 in [62] for details).

2.6.1 Performance Impact of RIFL

Performance is often used as an argument for weak consistency in large-scale systems. However, we found that RIFL is able to add full linearizable semantics to RAMCloud with negligible impact on performance.

Figure 2.8 shows the latency of RAMCloud write operations before and after adding RIFL. RIFL increases the median write latencies by less than 4% (530 ns). More generally, the overall shape of the latency distribution did not change with the addition of linearizability.

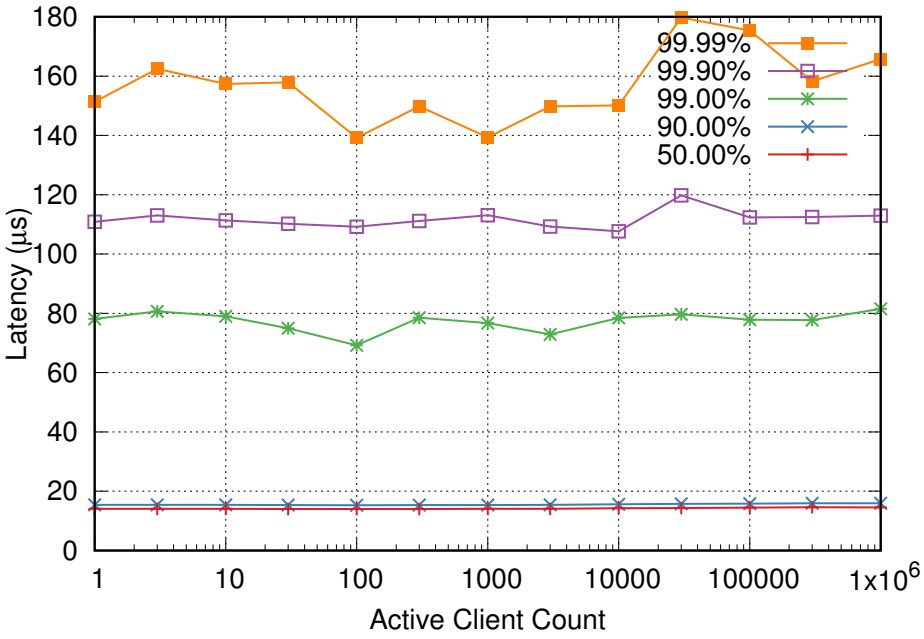

Figure 2.10: Latency of linearizable writes as a function of the amount of client state managed by the server. A single client application simulated many virtual clients, each with its own RequestTracker and client id. The client issued a series of random 100B write requests, with each write request using a randomly chosen virtual client. All requests targeted a single server in a 4-server cluster. The graph displays the median write latency as well as a number of tail latencies ("99%" refers to 99th-percentile latency). The median latency increased by less than 5% from 14.0 µs with 1 active client to 14.6 µs with 1M active clients.

Figure 2.9 shows the throughput of write operations for a single server, before and after introducing RIFL. Overall, RIFL had no measurable impact on throughput (experimental error is greater than the differences between the curves).

2.6.2 Scalability Impact of RIFL

RIFL creates three potential scalability issues. The first is memory space required on servers for completion records and client lease information. RIFL uses 40B per client for overall client state, plus roughly 76B for each completion record, or a total of 116B per client (assuming one active completion record per client per server). If a server has 1M active clients, the total memory requirements for RIFL data will be 160MB per server; given current server capacities of 256GB or more, this represents less than 0.1% of server memory.

The second scalability risk is the possibility of performance degradation if a server must manage

state for a large number of clients (for example, this could result in additional cache misses). To evaluate this risk, we used a single client executable to simulate up to one million active clients, all issuing linearizable write requests to the same server (Figure 2.10). Ideally, the write latency should be constant as amount of client state on the server increases. Instead, we found a 5% increase in latency when scaling from one active client to one million. While the overhead does increase with the number of clients, the overhead is small enough that it doesn't limit system scalability.

The third scalability issue is the lease renewal traffic that the lease server must process; the number of active clients is limited by the rate at which the lease server can serve lease renewal requests. We measured the lease server's maximum throughput to be 750k renewals per second. Lease terms are currently set at 30 min with renewals issued after half the term has elapsed. At this rate, 1M active clients will consume less than 0.2% of a lease server's capacity.

2.6.3 Is Implementing Linearizability with RIFL Hard?

Given RIFL, which consists of about 1200 lines of C++ code, we were able to add linearizability support to existing RAMCloud operations with only few additional lines of code. We did this for write, conditional write, increment, and delete operations. On the client side, RAMCloud RPCs are implemented using *wrapper* objects that implement core functionality common to a set of RPCs. As part of our RIFL implementation, we created a new linearizable RPC wrapper with 109 lines of code. Using this wrapper, each operation's client-side code required only 4 lines of code modification. Each operation's server-side code needed 13 lines of code modification. In all, we were able to add full linearizability support for all 4 operations in 68 lines of code, or 177 lines including the new wrapper.

RIFL also significantly simplified the implementation of distributed transactions in RAMCloud. This was discussed in detail in section 2.5.5.

2.6.4 Evaluating RIFL-based Transactions

Figure 2.11 shows the latency for simple transactions with 1–5 objects, each on a different participant server. A transaction with only a single object commits in 17.8 µs at the median while a transaction with 5 objects commits in 27.3 µs. Figure 2.12 shows the transaction commit latencies for larger transactions up to 300 objects on 60 participants; latency increases roughly linearly with the number of participants. This scaling behavior is expected as the client must issue a separate RPC for each additional participant.

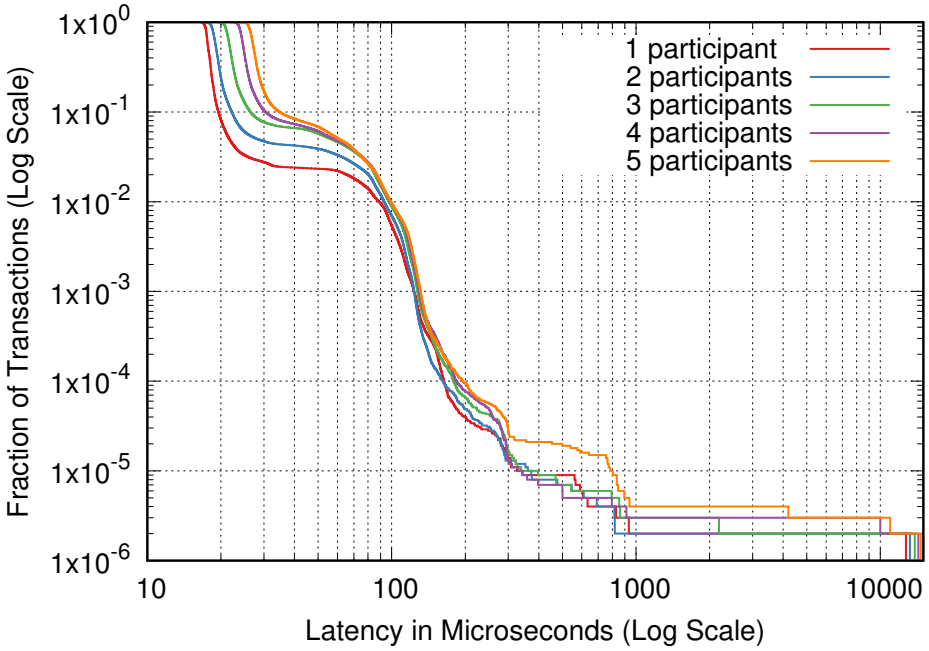

Figure 2.11: Complementary cumulative distribution of transaction commit latency measured from the beginning of the commit process to the completion of all `prepare` RPCs. Each transaction reads and writes one randomly chosen object per participant; the cluster contained 10 servers. A point (x, y) indicates that y of the 1M measured transaction commits took at least x µs to complete. The median latency for 1, 2, 3, 4, and 5 participants is 17.8 µs, 19.2 µs, 21.8 µs, 24.8 µs, and 27.3 µs respectively.

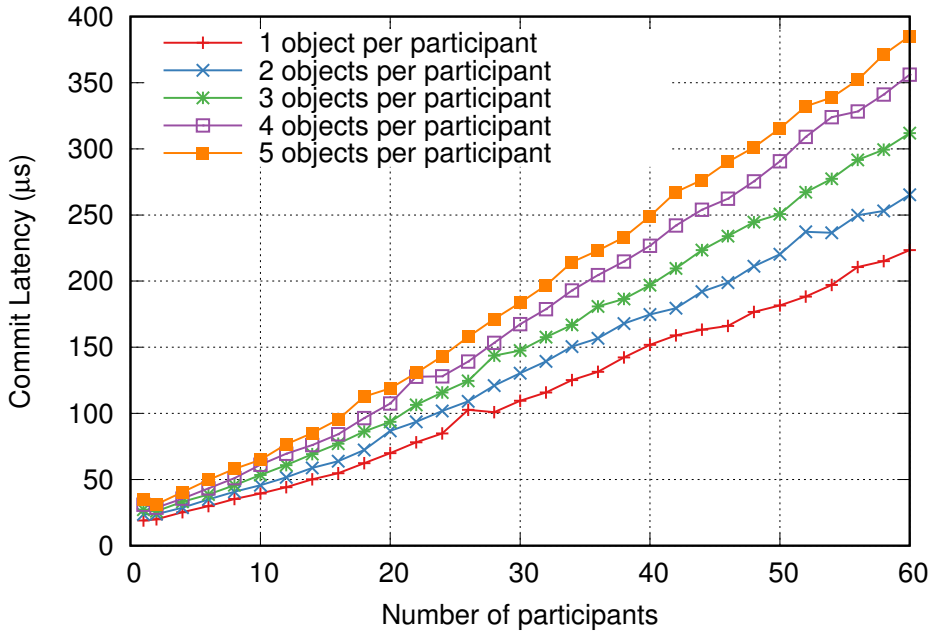

Figure 2.12: Median transaction commit latency for a single client measured from the beginning of the commit process to the completion of all `prepare` RPCs, as a function of the number of participants. For instance, with 5 objects per participant and 60 participants the transaction is committing 300 objects spanning 60 participants. All experiments used a cluster with 60 servers.

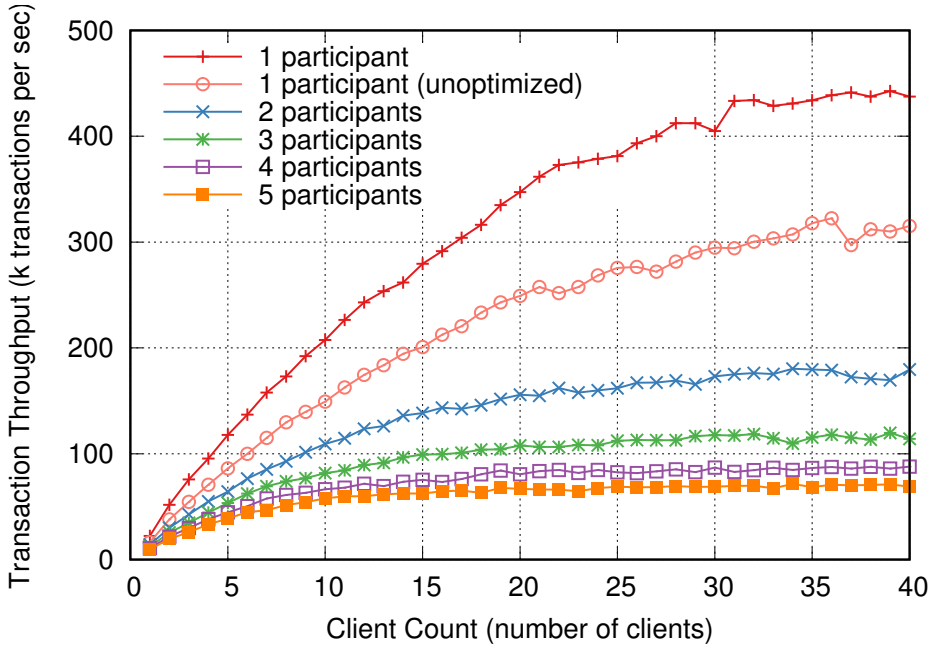

Figure 2.13: Aggregate transaction throughput for a cluster of 10 servers as a function of the number of clients. Each client repeatedly creates and commits transactions back to back where each transaction reads and writes one 100-byte object per participant; participants were chosen at random for each transaction. Each experiment was run 5 times; the figure shows the median measured throughput. The "unoptimized" case shows throughput with the single-server optimization disabled.

Figure 2.13 graphs the throughput of RIFL-based transactions. For transactions involving a single participant, a cluster with 10 servers can support about 440k transactions per second. With more participants per transaction, throughput drops roughly in proportion to the number of participants: with five participants in each transaction, the cluster throughput is about 70k transactions per second. Figure 2.13 also shows that the single-server optimization described in Section 2.5 improves throughput by about 40%.

2.6.5 Comparing Transaction Performance with H-Store

We implemented the TPC-C benchmark [80] with RAMCloud transactions and used it to compare performance between RAMCloud and H-Store [40], a main-memory DBMS for OLTP applications. TPC-C simulates an order fulfillment system with a workload consisting of five transaction types (see Table 2.2). Data is logically partitioned into *warehouses*, and most transactions only manipulate data within a single warehouse. The primary metric for TPC-C is the number of committed

Type	Mix	Working Set (DB rows)		Multi-warehouse
		Read Set	Write Set	
NewOrder	45%	23 rows	23 rows	9.5%
Payment	43%	4 rows	4 rows	15%
OrderStatus	4%	13 rows	0 rows	0%
Delivery	4%	130 rows	130 rows	0%
StockLevel	4%	390 rows	0 rows	0%

Table 2.2: The five transaction types used by the TPC-C benchmark. "Mix" gives the frequency for each transaction type. "Working Set" indicates how many database records are read and written by each transaction type, on average. "Multi-warehouse" indicates the fraction of each transaction type that involves multiple warehouses.

NewOrder transactions per minute (TpmC).

Our implementation of the TPC-C benchmark for RIFL-TX differs slightly from the standard in that we removed wait time and keying time in clients (this allowed us to generate a higher workload with fewer clients). We modeled each table row in TPC-C as an object in RAMCloud and simulated secondary indexes using auxiliary RAMCloud tables. For instance, we used an auxiliary table to map from each customer last name to a list of matching customer IDs. Updates to the auxiliary tables are included in the TPC-C transactions.

H-Store provides many parameters to tune the system's performance. With help from the H-Store developers [64] and to the best of our ability, we tuned two H-Store configurations for measurement: one for best latency and one for best throughput. In some experiments we disabled durability in the H-Store configuration optimized for latency. RAMCloud does not provide any tuning parameters so RIFL-based transactions on RAMCloud were measured as-is (we view RAMCloud's lack of parameters as an advantage).

Figure 2.14 shows TPC-C latency with a single client and a single warehouse. Warehouses are relatively small, so TPC-C normally runs with each warehouse located entirely on a single server; this is the leftmost point in Figure 2.14. In this configuration H-Store's latency is more than 8x higher than RAMCloud's when durability is enabled in H-Store, and 23% higher even when durability is disabled in H-Store. Furthermore, RAMCloud achieves its low latency while providing 3-way distributed replication of all data, whereas H-Store does not perform replication. H-Store is exceptionally efficient for single-server transactions because it uses pre-defined stored procedures for all transactions. In the single-server case the server can execute all reads and writes locally and no data needs to be transferred either to clients or other servers.

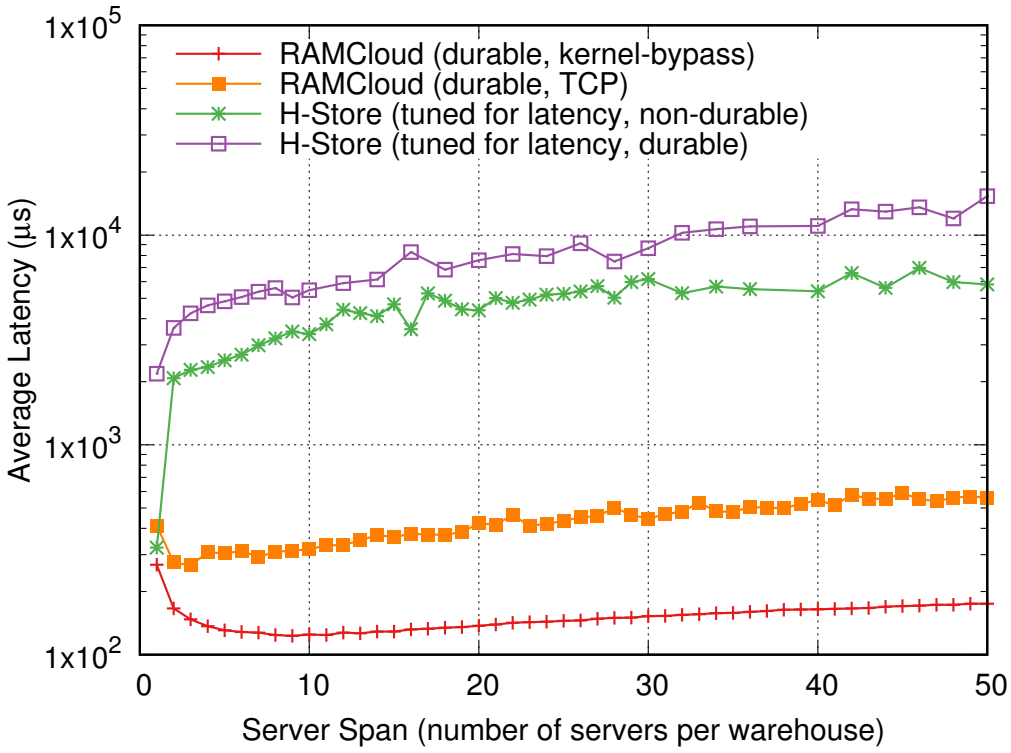

Figure 2.14: Average latency of TPC-C NewOrder transactions with a single warehouse, as a function of the degree of sharding for the warehouse. "Server Span" of 10 indicates that the data of a single warehouse is sharded across 10 servers. H-Store was tuned for low latency and measured with durability both enabled and disabled. RIFL-based transactions were measured on both RAMCloud's standard kernel-bypassing transport as well as over kernel TCP.

Figure 2.14 also shows the latency for a modified TPC-C benchmark configuration where the warehouse dataset is sharded across multiple servers. This evaluates how the systems will perform on datasets too large to fit on a single server. RIFL-based transactions execute faster with sharding, because they issue RPCs concurrently to all the participants. In H-Store, performance degrades with sharding: RAMCloud latencies are 1–2 orders of magnitude less than H-Store.

RAMCloud's kernel-bypass transport is significantly faster than TCP, which is used by H-Store. To level the playing field, Figure 2.14 also contains measurements of RAMCloud using TCP. Even with TCP, RAMCloud is still significantly faster than H-Store except in the single-server case. The gap in latency between H-Store and RAMCloud over TCP indicates that H-Store's latency is not limited by network speed, so H-Store's latency would not improve much if it used kernel-bypass.

Figure 2.15 shows latency and throughput for the TPC-C benchmark with the standard mix detailed in Table 2.2, with each warehouse on a separate server. The measurements for H-Store tuned for throughput overstate its throughput because we were not able to run the benchmark long enough for H-Store to reach steady state (the servers crashed). During the shortened runs, H-Store completed only about half as many distributed transactions as required by the mix; the others were still in progress when the runs ended. In the steady state, the more expensive distributed transactions would have limited performance.

RIFL-TX outperforms H-Store for TPC-C both in latency and throughput. RIFL-TX's advantage is smallest when comparing throughput between RAMCloud over TCP and H-Store optimized for throughput; even so, RIFL-TX's throughput is 35% higher at 16 servers, even though RAMCloud is performing 3x replication and H-Store is not executing the full TPC-C transaction mix. When RAMCloud over TCP is compared with H-Store optimized for latency, RAMCloud's throughput is 3.6x that of H-Store. When RAMCloud runs with kernel-bypass, its throughput is 2.8–7.6x that of H-Store. RAMCloud latency is at least 10x smaller than H-Store when H-Store is tuned for latency, and 1000x smaller when H-Store is tuned for throughput.

2.6.6 RAMCloud Throughput Limitations

The RAMCloud architecture was optimized for latency, not throughput; as a result, RIFL's throughput is lower than it could be with a different underlying architecture. In particular, RAMCloud does almost no batching of requests; this approach improves latency under low load, but hurts throughput under high load. The primary bottleneck for transaction throughput is the pipeline for replicating new data to backups. A single RAMCloud server can maintain only about 1.5 outstanding replication operations at a time, on average, where a "replication operation" consists of all of the mutations

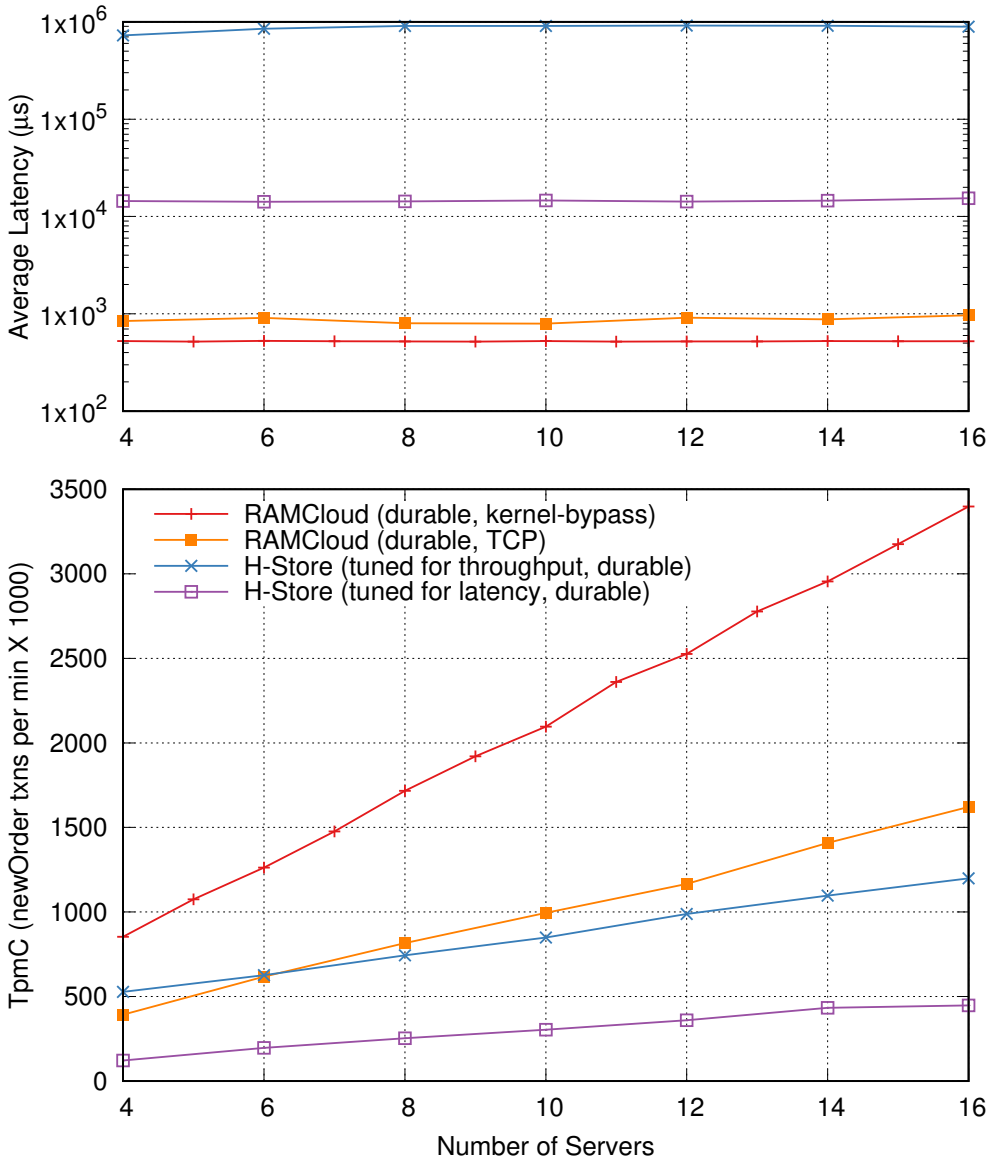

Figure 2.15: Latency and aggregate throughput of NewOrder transactions in the TPC-C benchmark, as a function of the system scale. The benchmark used the transaction mix in Table 2.2, with each warehouse stored on a single server. Increasing the number of servers also increased the number of warehouses. Roughly 90% of NewOrder transactions involved only one warehouse. RIFL-based transactions were tested on both RAMCloud's standard kernel-bypassing Infiniband transport as well as kernel TCP.

associated with one incoming RPC (such as an individual write or a `prepare` RPC describing one or more objects for a single transaction). This serialization is particularly harmful in high-latency environments such as when RAMCloud uses TCP.

In Chapter 4, we will present CURP, which solves this throughput limitation by moving replication out of the critical path and thereby enabling more aggresive batching of replication operations.

2.7 Related Work

Numerous distributed storage systems have been proposed or implemented in recent years, with a variety of approaches to consistency. Many of the early systems intentionally sacrificed consistency to enhance scalability or partition tolerance [67, 28, 23, 43]. Linearizability is impossible for these systems since they do not maintain consistency between replicas. The difficulties of programming weakly consistent systems have been widely discussed, so recent systems have tended towards stronger forms of consistency [50, 8, 22]. Some of these systems offer semantics close to linearizability, but it is difficult to tell whether any are truly linearizable. For example, Spanner [22] claims to provide consistency that is equivalent to linearizability, and it performs retries internally for certain transactions, but it does not appear to extend exactly-once semantics all the way out to clients.

Many traditional databases provide exactly-once semantics for transactions using queued transaction processing [13], but this technique is fairly heavyweight compared to RIFL. Queued transaction processing separates the execution of a single logical user defined transaction into 3 phases: request submission, request execution, and reply processing. To ensure exactly-once semantics in the face of server crashes, each phase is performed as a transaction. The overhead of turning each request into 3 transactions is especially noticeable for single-object requests.

Many systems provide strong internal consistency guarantees, but most of these systems do not provide exactly-once semantics for clients. For example, H-Store [40], Spanner [22], and FaRM [25] implement distributed transactions, but an untimely server crash can leave clients without a clear indication whether a transaction committed. A client can implement linearizability in these systems by including an additional object in the transaction with a unique identifier, which it can check later to determine if the transaction committed. However, this requires clients to create their own mechanisms for managing transaction ids.

The distributed transaction mechanism implemented by Sinfonia [6] is linearizable because clients serve as transaction coordinators: this guarantees that they see exactly-once semantics for

transaction commits. However, Sinfonia does not describe how to implement unique transaction ids, nor does it handle situations where data migrates (transactions hang until crashed servers restart).

Consensus-based systems such as ZooKeeper [35] and Raft [58] provide strongly consistent replication. ZooKeeper claims to provide "asynchronous linearizability" but does not describe what this means or how the various issues addressed by RIFL are handled. Raft outlines how to implement linearizability but does not describe how to allocate unique identifiers, and Raft does not address the migration issue, since all data is replicated on every server.

The techniques used by RIFL for ensuring exactly-once semantics share several features with network transport protocols such as TCP [36], including sequence numbers, acknowledgments, retries, and limits on outstanding operations. RIFL differs in that its protocol must survive server crashes and migration of the communication end-points.

Some Network File System (NFS) servers implement a mechanism called reply cache [37] which detects duplicate requests and replies with the response from the original execution. However, a reply cache doesn't guarantee linearizability since it may lose responses in case of server crashes or cache overflows. Highly Available NFS [14] persists the reply cache entries, but it uses the reply cache only to prevent false exceptions, not to provide full linearizability; when an exception occurs during an execution (such as trying to create a directory that already exists), the server checks if the request was previously successful and ignores the exception.

2.8 Client Failures and the Ultimate Client

RIFL only guarantees exactly-once semantics for an operation if the client is reliable. If a client crashes and loses its state, then its operations in progress may or may not complete. We assume this behavior is acceptable, since the timing of the crash could already cause some operations not to complete.

However, the client may itself be a server for some higher-level client. For example, in a large-scale Web application the lowest-level client is typically a front-end server that responds to HTTP requests from Web browsers; it issues requests to other servers in the datacenter and generates HTTP responses. Even if the interactions between the front-end server and back-end servers are implemented in an exactly-once fashion, the front-end server could crash before responding to the Web browser. If this happens, the browser may retry the operation, resulting in the same problems that RIFL was designed to eliminate.

One way to handle these multi-layer situations is to implement RIFL at each layer. For example,

browsers could generate a unique identifier for each HTTP request, which can be recorded on front-end servers and used to ensure that requests are carried out exactly once.

However, the ultimate client is a human, and humans are not likely to implement RIFL. For example, what happens if a user invokes an operation in his/her browser and the browser (or the machine running it) crashes during the operation? Once again, the operation may or may not complete.

The only way to ensure true end-to-end exactly-once semantics is for systems to provide users with clues of potential duplicates, so that users can avoid requesting already completed requests again. For example, when an Amazon.com user clicks "Place your order," the Amazon server checks if the user previously ordered the same item. If so, instead of completing the order right away, the server redirects the user to a confirmation page to verify if it's a valid new order.

Even though final responsibility for exactly-once semantics must lie with the user, it is still important for the underlying system to support linearizability in its implementation. This allows the system to be constructed in layers while avoiding internal duplication of operations, so that a user-recognizable operation occurs either exactly as specified or not at all. As a counter-example, if an order were partially filled, it might be difficult for the user to recognize that it needs to be (partially) reissued.

2.9 Linearizability and Transactions

One interesting question for system design is the relationship between linearizability and transactions. Ultimately, distributed systems often must provide both transactions and exactly-once semantics. In the traditional approach, transactions are implemented first (a relatively thick layer). Exactly-once behavior is then implemented on top of transactions using one of the approaches discussed in Section 2.7, such as queued transactions. In RIFL we first implemented linearizability as a distinct layer, then built transactions on top of it.

The RIFL approach has two advantages. First, the linearizability layer can also be used for simple operations that do not require distributed transactions, such as writes and increments. This results in lower latency and higher throughput for these operations.

The second advantage of a separate linearizability layer is that it provides a better modular decomposition. It encapsulates a significant fraction of the complexity of distributed transactions into a separate layer, which can be implemented and validated independently. The existence of a linearizability layer then makes transactions easier to implement, and it eliminates the need for an exactly-once layer on top of transactions.

2.10 Summary

This chapter has described RIFL, a mechanism for achieving exactly-once RPC semantics for large-scale distributed systems. RIFL is a general-purpose mechanism, independent of any particular operation or system. We have implemented RIFL in the RAMCloud storage system to demonstrate its versatility and performance. RIFL allowed us to make basic operations such as writes and atomic increments linearizable with only a few additional lines of code, and it significantly simplified the implementation of a high-performance distributed transaction mechanism.

Chapter 3

Implementing RIFL on Other Systems

Chapter 2 discussed how to implement RIFL on a large-scale distributed storage system, RAMCloud. The generality and complexity of RAMCloud made RIFL's design a little more complicated than it needs to be when implemented on some other systems.

Implementing RIFL is challenging particularly because of these two key requirements (out of 4 requirements in §2.2):

- *Atomic durability of the execution results with the mutated objects.* To implement this, RIFL needs to directly use the persistence layer deep in each RPC's execution code, so RIFL cannot be deployed as a feature of an RPC package such as gRPC [27] or Apache Thrift [77].
- *Finding completion records after reconfiguration (retry rendezvous).* To implement this, RIFL needs to modify a system's migration and recovery implementation.

The rest of this chapter discusses a situation when the two features can be simplified and how to implement them easily.

3.1 RIFL on systems that replicate client requests

There are two ways to make an operation durable: the first way is saving the outcome of executions such as the new value of mutated objects (used by RAMCloud), and the second way is saving the client requests for replay during recovery (used by Redis, MySQL). Systems using the second mechanism can recover from a server crash by replaying the saved client requests (they assume operations are deterministic). Both mechanisms to ensure the durability of operations are widely used, and some systems such as MySQL support both mechanisms.

Implementing RIFL can be much simpler for systems that save the client requests for durability. The key difference is that such systems don't need to save completion records durably but only need to keep execution results in memory. When a server crashes and a new server reconstructs the pre-crash state by replaying the saved client requests, RIFL can naturally reobtain the execution results without the help of completion records. The metadata for RIFL such as the RPC ID and acknowledgment ID are already part of client requests themselves, so replaying client requests also automatically reconstructs the pre-crash state of ResultTracker as well. For proper crash recovery, RIFL merely has to ensure that the replays of client requests go through the normal execution paths so that RIFL methods can be invoked as in the original executions. In conclusion, RIFL only needs to save the execution results in memory; for convenience, we will assume execution results are saved inside ResultTracker.

3.1.1 Benefits

Keeping execution results only in memory simplifies the RIFL architecture in many ways. First, RIFL no longer needs to be aware of the durability/replication system. One of the prerequisites for implementing RIFL on a system is that its durability/replication mechanism should support atomic commitment of a completion record along with the effects of its execution. Without the need for durable completion records, implementing RIFL doesn't need to touch the durability/replication layer at all.

Second, RIFL no longer has to worry about retry rendezvous. If an object is recovered at a different server after a crash, the object must be reconstructed by replaying client requests. The execution results of replays can be saved in ResultTracker. Without durable completion records, RIFL no longer needs to associate each RPC with a specific object or modify migration code for handling completion records; RIFL can just rely on the underlying system to migrate client requests accordingly.

Third, RIFL's performance overhead is reduced as well. Since a system no longer has to save the execution results as durable completion records, a system can save disk bandwidth (durability) or network bandwidth (replication). Thus, RIFL can stay efficient even for operations with large responses such as compare-and-swap or read-and-replace.

3.1.2 Accommodating snapshots

Systems that save client requests for durability use snapshotting to limit the size of redo logs. After snapshotting the current system state, systems drop all client requests before the time of the snapshot. When a system crashes, it can recover first from the snapshot and then replay the client requests that were executed after the snapshot.

The removal of client requests can be a problem since RIFL relies on client requests for recovering RIFL's internal data structure, ResultTracker (which now incorporates execution results). To accommodate snapshotting, the state of ResultTracker must be snapshotted as well. This can be done by scanning the table in ResultTracker and saving RPC IDs and execution results.

If a system blocks normal operations while snapshotting, snapshotting its data and ResultTracker is trivial. However, practical systems must not stop normal operation during snapshotting. They usually achieve the snapshot in background with two approaches: storing data in immutable data structures and relying on the operating system's copy-on-write such as fork.

For systems that fork their processors for snapshotting, RIFL adds no complexity. A fork makes a copy of the server's entire virtual address space including ResultTracker and in-memory completion records. Thus, the child process can write out both the server's state and RIFL's data, while the parent process continues servicing normal operations.

For systems using immutable data structures, a snapshotting task takes a reference to the immutable data structure. All client requests up until this time point should be discarded after the snapshot task is completed. In this case, RIFL can achieve the immutable-during-snapshot effect by using two lookup tables for ResultTracker. When a snapshot is taken, RIFL can freeze the currently used lookup table so that it can be scanned and snapshotted consistently. At the same time, the data in the immutable table can be copied to the new table. While copying the lookup table, RIFL checks incoming requests against both the new table and the old table. Once the new table incorporates all completion records of the old table, the old table resets for the next snapshotting.

Chapter 4

Removing Overheads of Consistent Replication

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 [73], file writes to network replicated drives [4], or scattered replication [59]). 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 [59]).

To show its performance benefits and applicability, I implemented CURP in two NoSQL storage systems: Redis [72] and RAMCloud [62]. Redis is generally used as a non-durable cache due to its very expensive durability mechanism. When CURP was applied to Redis, it provided 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 [49, 68, 54, 26, 9], 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).

4.1 Separating Durability from Ordering

Replication protocols supporting concurrent clients have traditionally 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 one or more backups in parallel.

Consensus protocols with strong leaders (e.g. Multi-Paxos [44] or Raft [58]) 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 [46] and Generalized Paxos [45] 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 §A.1 for a detailed explanation on why they cannot achieve 1 RTT.)

Network-Ordered Paxos [49] and Speculative Paxos [68] 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.

To achieve durability in 1 RTT, CURP clients directly record their requests in temporary storage,

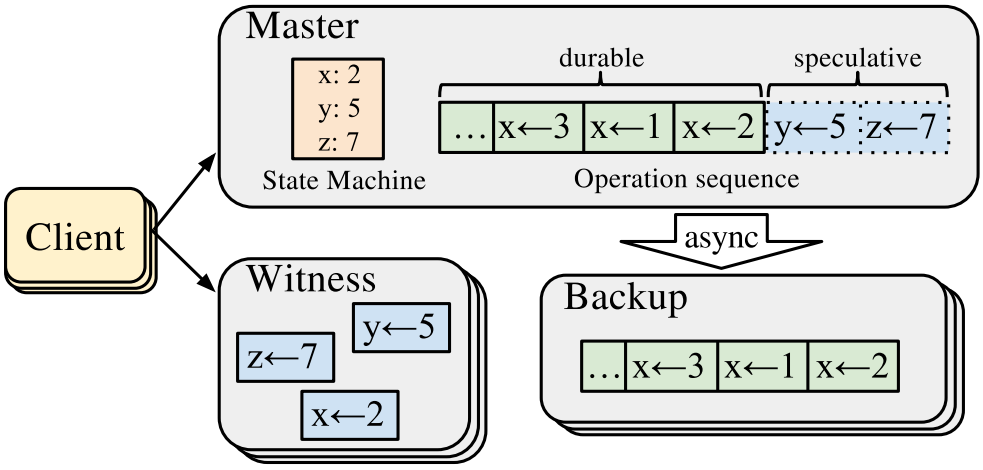

Figure 4.1: CURP clients directly replicate to witnesses. Witnesses only guarantee durability without ordering. Backups hold data that includes ordering information. Witnesses are temporary storage to ensure durability until operations are replicated to backups.

called a *witness*, without serializing them through masters. As shown in Figure 4.1, 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. Overall, a client can 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 Chapter 2 presented a general solution, RIFL. CURP reuses RIFL to avoid duplicate executions during recovery.

We can apply the idea of separating ordering and durability to both consensus-based replicated state machines (RSM) and primary-backup, but this chapter 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 [17], ZooKeeper [35], Raft [58] are used for cluster coordinators in GFS [30], HDFS [75], and RAMCloud [62]). 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 chapter will focus on the CURP protocol for primary-backup, which is the main replication technique for data servers. In §5.2, we sketch how the same technique can be used for consensus.

4.2 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.

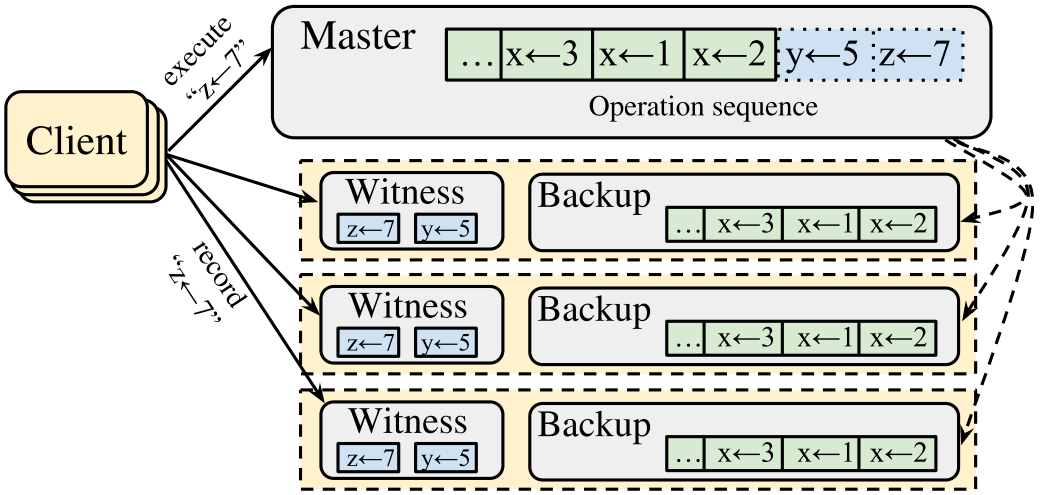

Figure 4.2: CURP architecture for $f = 3$ fault tolerance.

4.2.1 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 4.2, 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 this chapter, we assume that witnesses are separate from backups. This allows CURP to be applied to a wide range of existing replicated systems without modifying 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 §5.1 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 [68] and Network-Ordered Paxos [49] require special networking hardware and cannot be used for geo-replication.)

4.2.2 Normal Operation

4.2.2.1 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 (potentially new) master and tries to record the RPC request in witnesses of the new master.

4.2.2.2 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 [62].

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 a 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 orders, which can cause witnesses to accept and reject different sets of operations. However, this does not endanger consistency. First, as mentioned in §4.2.2.1, 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 §4.2.3).

4.2.2.3 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.

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 4.3, 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 operation. If a master responds for a noncommutative 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

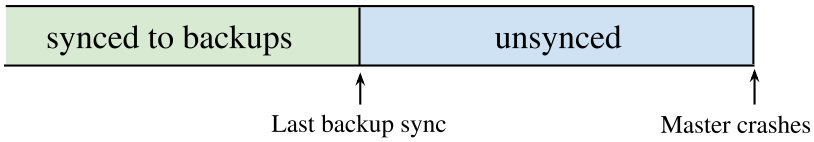

Figure 4.3: Sequence of executed operations in the crashed master.

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. In the read vs. write case above, the master will sync the write before returning from the read.

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.

4.2.3 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 [47].

To avoid duplicate executions of the requests that are already replicated to backups, CURP relies on exactly-once semantics provided by RIFL [47], 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 [47], 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.)

4.3 Informal Proof of Correctness

With the normal operation behaviors described in §4.2.2, the recovery protocol in §4.2.3 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 §4.2.2.1, 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 §4.2.2.3; 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 ordered list of all operations that were executed at the master before α was executed, 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 4.3: ⟨synced⟩ followed by ⟨unsynced⟩. The 1st phase of recovery will recover the exactly same execution history for the ⟨synced⟩ part. By (Rule 2), we know that losing any ⟨unsynced⟩ 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. It might seem that CURP could 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 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. □

4.3.1 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 [47]).

4.3.2 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.

4.3.3 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 §4.2.2.3. If the read operation conflicts with some unsynced update operations, the master must sync the unsynced updates to backups before responding for the read.

4.3.4 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 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

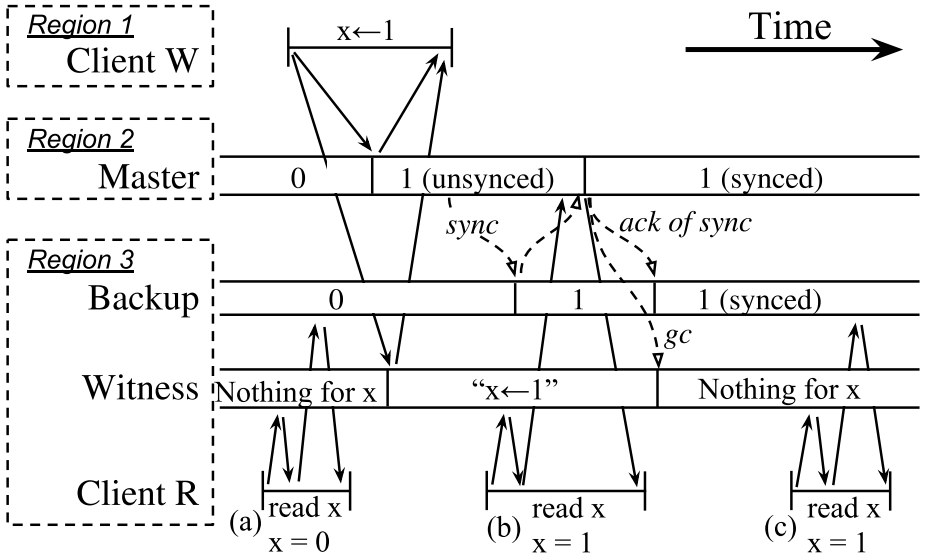

Figure 4.4: Three cases of reading the value of x from a backup replica while another client is changing the value of x from 0 to 1: (a) client R first confirms that a nearby witness has no request that is not commutative with "read x," so the client directly reads the value of x from a nearby backup. (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.

currently saved in the witness (as shown in Figure 4.4). 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.

4.4 Implementation on NoSQL Storage

This section describes how to implement CURP on low-latency NoSQL storage systems that use primary-backup replication. With the emergence of large-scale Web services, NoSQL storage systems became very popular (e.g. Redis [72], RAMCloud [62], DynamoDB [76] and MongoDB [2]), 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

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 4.5: The APIs of witnesses.

problem with a broad impact.

The most important piece missing from §4.2 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.

4.4.1 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 4.5. 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.

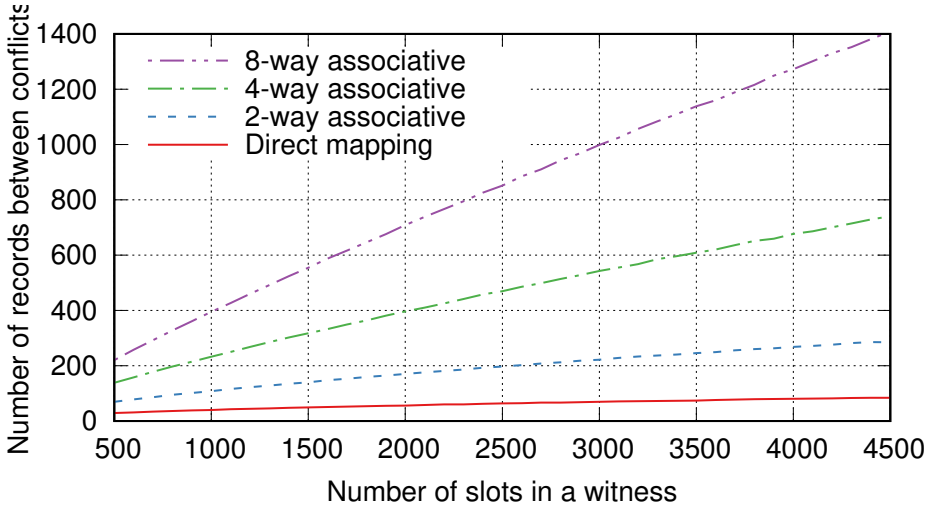

Figure 4.6: Simulation results for the expected number of recordings before a collision occurs in a witness' cache, assuming a random distribution of keys. Each data point is the average of 10000 simulations. Introducing associativity reduces the chance of collisions significantly.

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 sends **end** to free up the resources, so that the witness server can start another life for a different master.

4.4.2 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.

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 4.6 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.)

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.

4.4.3 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 the last backup sync started. By comparing the timestamp of an object against the start time of the last successfully completed backup sync, a master can tell whether the value of the object has been synced to backups.

4.4.4 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 and batch syncs 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 [62] 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.

4.4.5 Garbage Collection

As discussed in §4.3.1, masters send garbage collection RPCs for synced updates to their witnesses. Right after syncing to backups, masters send **gc** RPCs (in Figure 4.5), 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 [47]. 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 $\langle$*keyHash*, *rpcIds*$\rangle$ 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 a witness rejects a **record** RPC due to a conflict, it marks the existing noncommutative request as uncollected garbage if there have been many garbage collections since the request was recorded (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 execute the uncollected requests (most likely filtered by RIFL) as if they were sent from clients so that the regular garbage collection mechanism will include the uncollected requests in the next **gc** request.

4.4.6 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 4.5), 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 [47], 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).

4.4.7 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 [62]). 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.

4.4.8 Modifications to RIFL

In order to work with CURP, the garbage collection mechanisms of RIFL in §2.3.2 and §2.3.3 must be modified. 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 order, acknowledgements piggybacked on later requests can make masters ignore the replay of earlier requests. Thus, clients' acknowledgments included in RPC requests must be ignored during recovery from witnesses.

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.

4.5 Evaluation

I 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.

	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 4.1: The server hardware configuration for benchmarks.

4.5.1 RAMCloud Performance Improvements

RAMCloud [62] 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) [59].

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 4.1. 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 [59], 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 RAMCloud writes.

Figure 4.7 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.

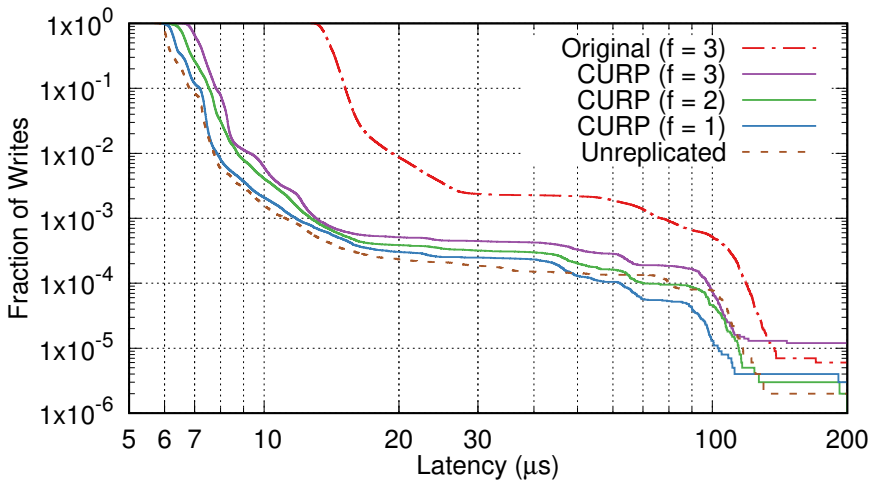

Figure 4.7: Complementary cumulative distribution of latency for 100B random RAMCloud writes with CURP. Writes were issued sequentially by a single client to a single server, which batches 50 writes between syncs. A point (x, y) indicates that y of the 1M measured writes took at least x µs to complete. 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.

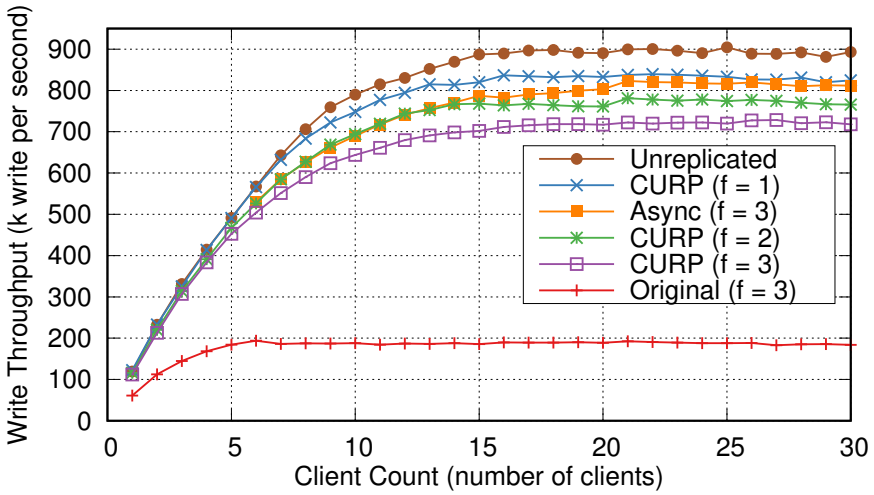

Figure 4.8: The aggregate throughput for one server serving 100B RAMCloud writes with CURP, as a function of the number of clients. Each client repeatedly issued random writes back to back to a single server, which batches 50 writes before syncs. Each experiment was run 15 times, and median values are 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.

Figure 4.8 shows the single server throughput of write operations with and without CURP as the number of clients varies. 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 $\sim$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 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.

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

4.5.2 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 (any request over 2KB are rejected and must take the slow path). 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).

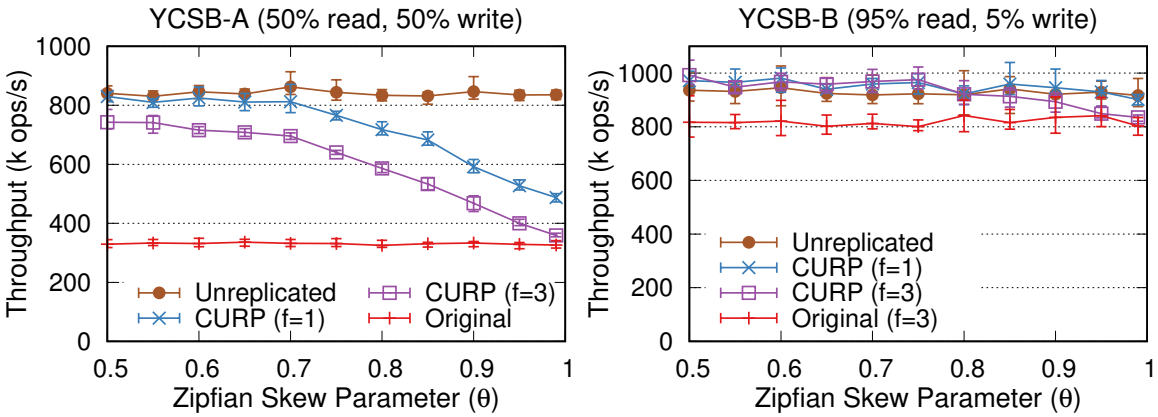

Figure 4.9: Throughput of a single RAMCloud server for YCSB-A and YCSB-B workloads with CURP at different Zipfian skewness levels. Each experiment was run 5 times, and median values are displayed with errorlines for min and max.

4.5.3 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 [33] with 1M objects. Specifically, we used two different workloads similar to YCSB-A and YCSB-B [21]; 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 4.9 shows the impact of workload skew (defined in [33]) 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 4.10 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.

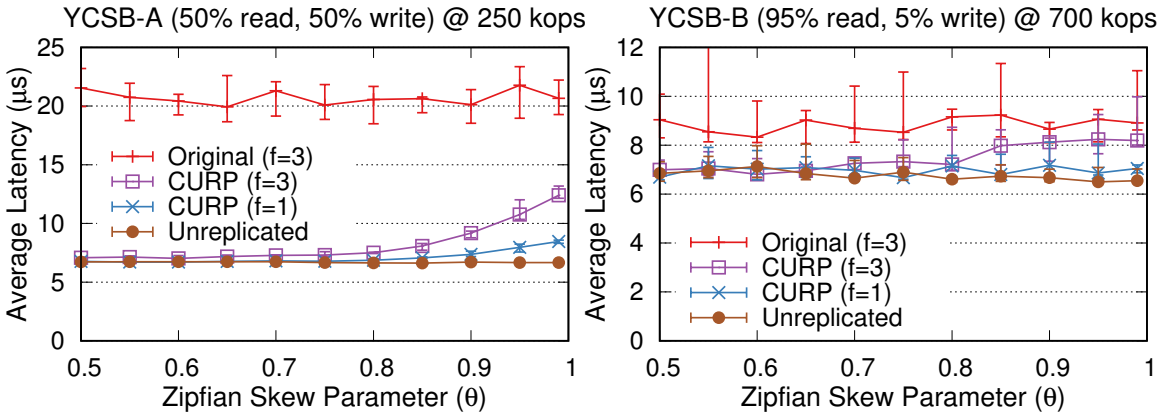

Figure 4.10: Average RAMCloud client request latency for YCSB-A and YCSB-B workloads with CURP at different Zipfian skewness levels. 10 clients issued requests to maintain a certain throughput level (250 kops for YCSB-A and 700 kops for YCSB-B). Each experiment was run 5 times, and median values are displayed with errorlines for min and max. Latency values are averaged over both read and write operations.

Figure 4.11 shows the latency distributions of reads and writes separately at $\theta = 0.95$ under the same loaded conditions as Figure 4.10. 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.

4.5.4 Making Redis Consistent and Durable

Redis [72] 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

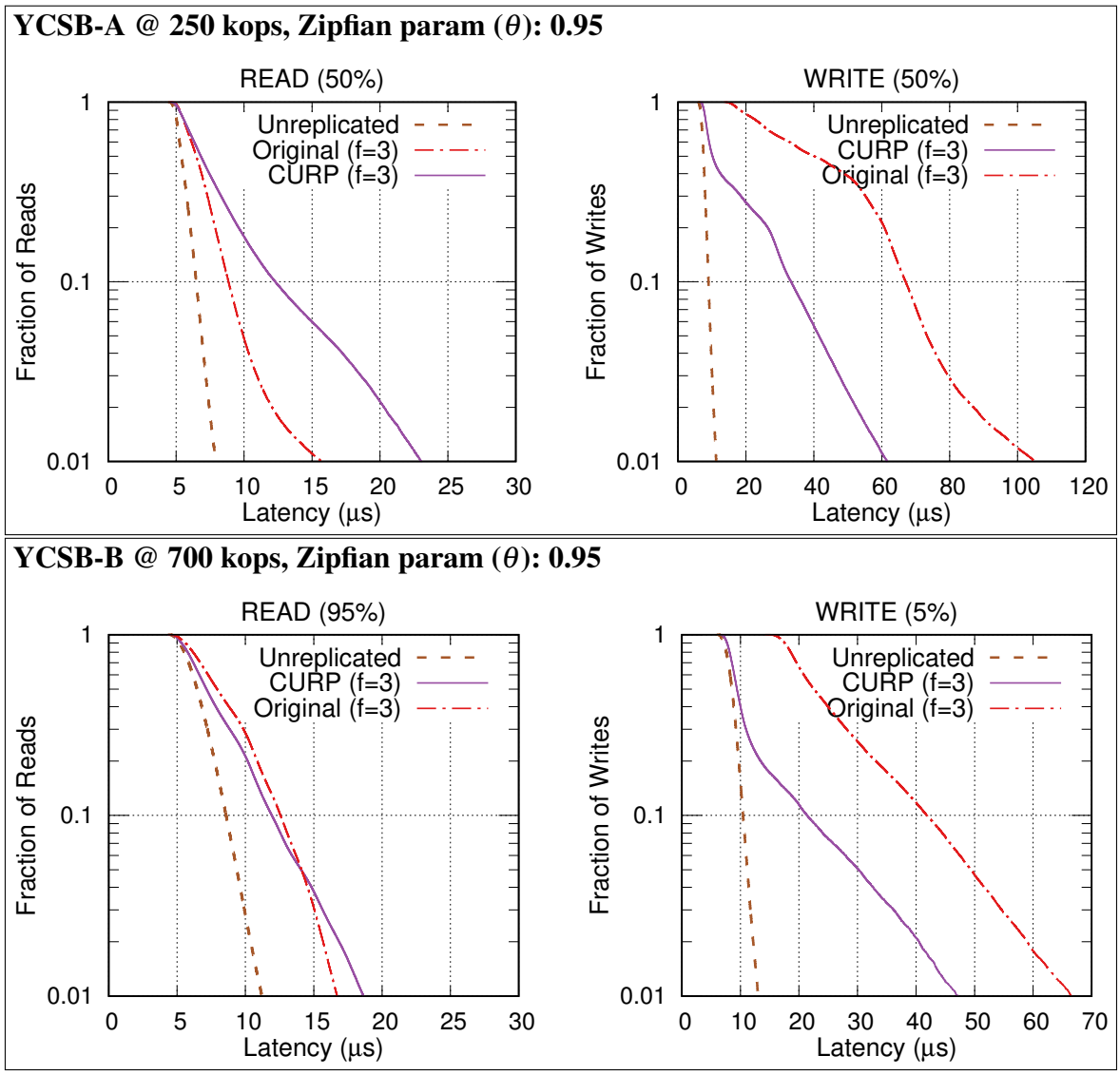

Figure 4.11: Complementary cumulative distribution of read and write latencies with CURP on a loaded server (250 kops for YCSB-A and 700 kops for YCSB-B). 10 clients issued read and write operations (using the read / write mix ratio of YCSB) for 1 min to a single server. The workloads used a Zipfian distribution with $\theta = 0.95$, which means 16% of operations are on keys that were accessed within the last 100 executed operations.

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 [71], whose specifications are in Table 4.1. 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" [78] for clients. We modified "C++ Client" to construct Redis requests more quickly.

Figure 4.12 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 4.12, 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 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 4.13 shows the throughput of Redis SET operations for a single Redis server with varying

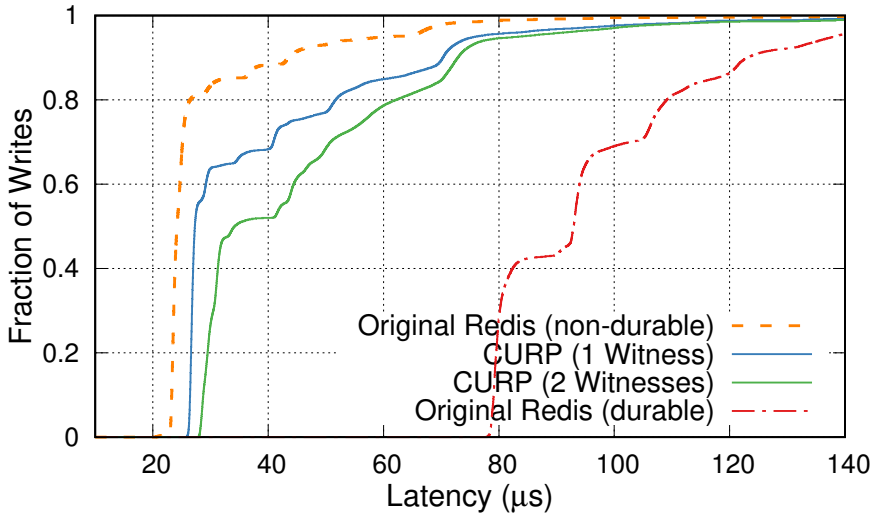

Figure 4.12: Cumulative distribution of latency for 100B random Redis SET requests with CURP. Writes were issued sequentially by a single client to a single Redis server. CURP used one or two additional Redis servers as witnesses. "Original Redis (durable)" refers to the base Redis without CURP, configured to invoke fsync on a backup file before replying to clients.

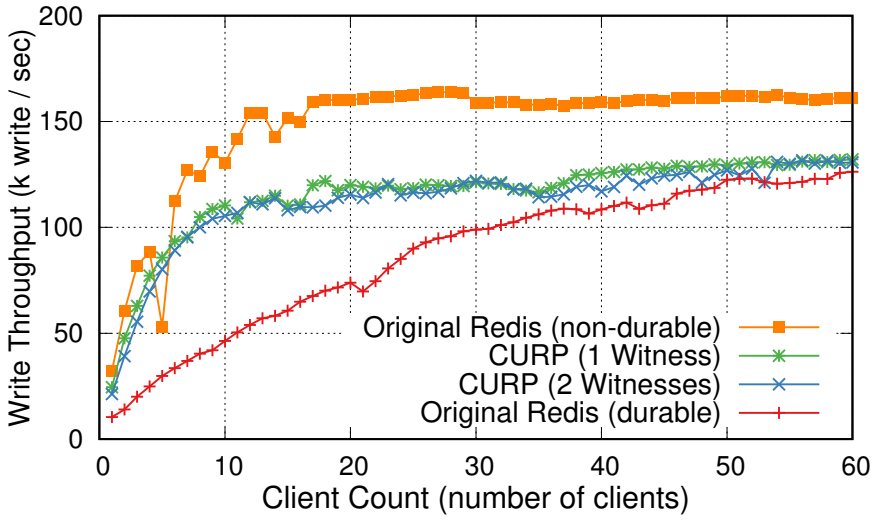

Figure 4.13: The aggregate throughput for one server serving 100B Redis SET operations with CURP, as a function of the number of clients. Each client repeatedly issued random writes back to back to a single server. "Original Redis (durable)" refers to the base Redis without CURP, but configured to invoke fsync before replying to clients.

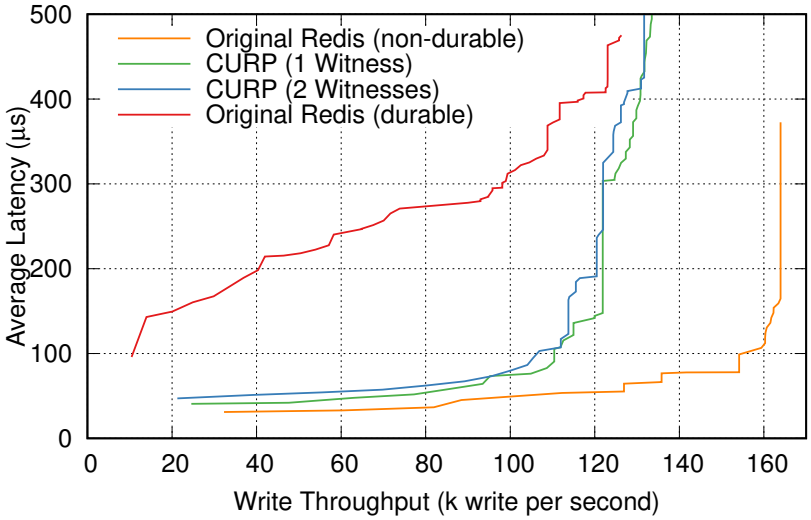

Figure 4.14: Observed latency at a specific throughput level for one server serving 100B Redis SET operations with CURP. "Original Redis (durable)" refers to the base Redis without CURP, but configured to invoke fsync before replying to clients. Original Redis processes requests from multiple clients, fsyncs once per eventloop, and replies to all clients.

numbers of clients. Applying CURP reduced the throughput of Redis with 60 clients about 17%. 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.

Figure 4.14 shows the observed latency during the throughput benchmark. Both CURP and non-durable Redis maintain latency low until it reaches 80% of max throughput. The latency of durable Redis increases almost linearly due to batching mentioned in the previous paragraph. The 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 delays before responding back to clients, so latency increases almost linearly.

4.5.5 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

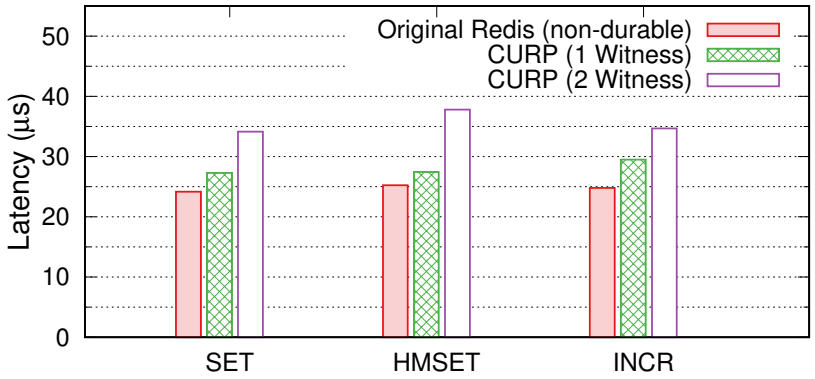

Figure 4.15: Median latencies before and after applying CURP on various Redis commands. All experiments select a random 30B key over 2M unique keys. SET used 100B random values, and each HMSET operation sets 1 member with a 100B value. The member key was 1B. Commands were issued sequentially by a single client to a single Redis server, with one or two additional Redis witness servers in CURP.

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 4.15 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.

4.6 Related work

Table 4.2 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 [45] allows clients to complete operations (i.e. receive execution results) in 1.5 RTTs and supersedes *Fast Paxos* [46]. 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

			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	$\sim$0 WAN RTT	1.5 WAN RTTs	$\sim$1 WAN RTT	Not Avail.
		write	1 WAN RTT	1.5 WAN RTTs	$\sim$1 WAN RTT	Not Avail.
load on leader		read	$<$1 RPC	$\sim n$ RPCs	$\sim$2 RPCs	1 RPC
		write	1 RPC	$\sim n$ RPCs	$\sim$2 RPCs	1 RPC

Table 4.2: 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". In a geo-replicated setting, CURP clients can read from local replicas, requiring only one local RTT which is much smaller than WAN RTT. NOPaxos's RTT is longer than usual since network packets must detour through a sequencer; "α" denotes the detour delay. 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.

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

Egalitarian Paxos (EPaxos) [54] 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 [58]. 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 [68] and *Network-Ordered Paxos* (NOPaxos) [49] reduce latency almost to 1 RTT by serializing client requests within network so that all of the consensus servers receive requests in the same order. Both protocols use SDNs to detour requests from all clients through a single

network device (a root layer switch or middlebox); thus, 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 ($\sim$25 µs) or NOPaxos($\sim$16 µs) have higher latency overhead compared to CURP ($\sim$1 µs).

TAPIR [83], Janus [55], and MDCC [41] 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* [81] has been widely adopted in industry [23, 20, 16]. Systems using eventual consistency return from updates before replication is complete, and replications happen asynchronously; since nearby replicas may miss up-to-date updates, clients must read from far-away masters for consistency. Pileus [79] and Tuba [7] allow applications to declare their consistency and latency priorities, and they dynamically select replicas to read from.

Broadcast-broadcast (BB) protocols [15, 9, 26, 38] for total order broadcasts [24] 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 [9, 26] 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.

4.7 CURP limitations for geo-replication

Altough CURP can be used for geo-replication since it doesn't rely on any special networking, the benefits of CURP may diminish with high message delays in a WAN. Most fast replication protocols (including CURP, Generalized Paxos, Egalitarian Paxos) rely on commutativity, and they all suffer

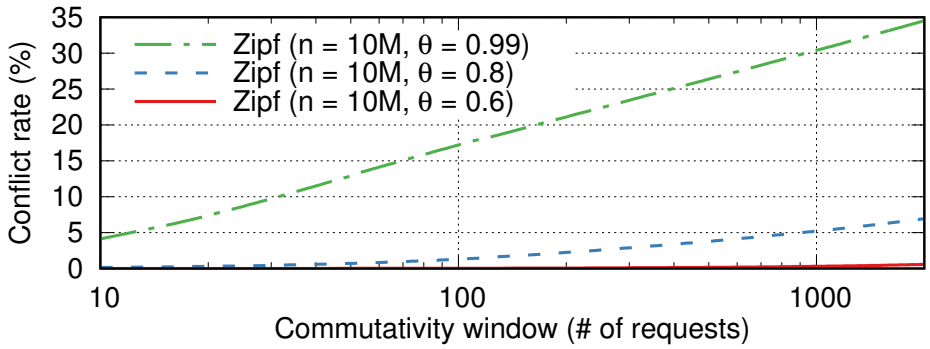

Figure 4.16: Simulated conflict rate as a function of workload skew and the number of requests that need to be commutative (commutativity window). A Zipfian distribution was used to pick a random key among 10M unique keys, and check whether the key was picked within last X times, where X is the commutativity window.

the same problem.

With much higher message delays in a WAN, CURP requires many more requests to be commutative for 1 RTT completion of a client operation; this increases the chance of commutativity conflicts. We can quantify how many more requests need to be commutative; a new request should be commutative with any requests in witnesses and any unsynced operation in the master. A request stays in a witness for 1.5 RTT (explanation in Figure 5.3), and it takes 1 RTT for the operation to get synced in the master; the greater time is 1.5 RTT. During the 1.5 RTT, $1.5 \times \text{RTT} \times \text{Throughput}$ requests are processed by the system, and they must be checked for commutativity (it may be higher with batched backup syncs). We will call $1.5 \times \text{RTT} \times \text{Throughput}$ the commutativity window. When used for geo-replication, CURP will suffer from a large commutativity window, which linearly increases with the message delay.

When combined with a skewed workload (generally considered realistic), the increased commutativity window dramatically raises the chance of commutativity conflicts, possibly enough to nullify the benefits of CURP. Figure 4.16 shows that simulated conflict rate increases in sublinear fashion as the commutativity window increases. For example, with a Zipfian distribution of $\theta = 0.99$, a 100x increase of commutativity window ($20 \rightarrow 2000$) incurs a 4.7x increase of the conflict rate ($7.4\% \rightarrow 34.5\%$); a commutativity window value of 20 corresponds to a typical Redis server in a datacenter with 100 µs RTT and 133 kops throughput, and a commutativity window value of 2000 corresponds to a situation when we geo-replicate the same Redis server in 10 ms RTT network.

4.8 Summary

In this chapter, 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.

Chapter 5

Alternative Designs for CURP

Chapter 4 presented a design of CURP that is appropriate for augmenting existing primary-backup replication systems. In this chapter, we represent two slightly different designs of CURP: one suitable for building systems from scratch, and one for augmenting quorum-based consensus systems. These are paper designs and not implemented or tested.

5.1 CURP-H: Combining Witnesses with Backups into Hybrids

The CURP design in Chapter 4 has witnesses separated from backups. Thanks to this design decision, CURP requires fewer changes to existing systems and is more applicable to many wildly different backup mechanisms. Both of the implementations in Chapter 4 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.

However, for systems whose clients can be aware of the backup servers for each master, combining witnesses and backups can bring extra performance benefits such as lower network bandwidth use. When witnesses and backups are combined, clients directly send requests to a master and backups. In each backup, there is a "witness module" which saves requests sent from clients (see Figure 5.1).

Combining witnesses and backups enables a performance optimization. The key change is that

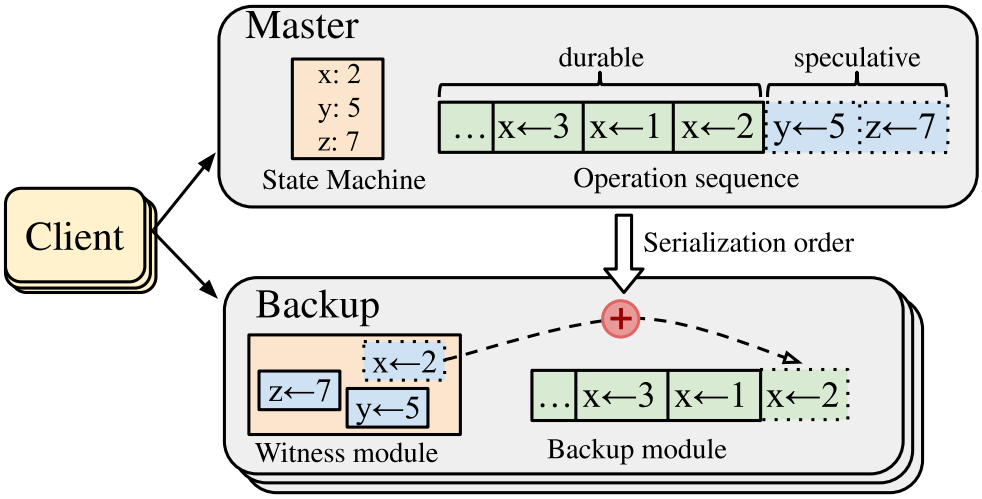

Figure 5.1: CURP-H clients directly replicate requests to backups, and the requests are saved in the "witness module" of each backup. Masters send only ordering information to backups instead of full client requests. The "backup module" retrieves full client requests from the witness module in the same backup. By serializing client requests from witness modules according to the ordering information from master, backup modules can replicate the data in a master.

masters now sync with backups by sending operation orders (by listing RPC IDs as in witness **gc** RPCs) instead of full client requests; then backups lookup the matching requests from their witness modules and move them to backup modules. This approach will lower network bandwidth consumption since RPC IDs are much smaller than full client requests.

To clarify the benefit of this design change, this section presents CURP-H, which is a modified design of CURP. The rest of this section discusses how CURP-H is different from CURP in Chapter 4.

5.1.1 Architecture and Failure Model

As shown in Figure 5.1, CURP-H uses one master and f backups to tolerate f failures. In this section, we assume a fail-stop model where either a master or a backup fails entirely (i.e., when a backup fails, both witness module and backup module fail and stop). There are no other changes except those mentioned above.

serialize(list of {*keyHash, rpcId, seqNum*}) → OK or REJECTED, list of *request*
Syncs a backup with the state of master by applying client requests with the given *keyHash*es and *rpcId*s as the *seqNum*'th operation. Returns two values: whether the sync was successful, and stale requests that haven't been garbage collected for a long time.

Figure 5.2: A possible request signature of the CURP-H's serialize RPC (for an implementation on NoSQL storage as in §4.4).

5.1.2 Normal Operation

Clients in CURP-H directly send their requests to backups in parallel with sending the requests to a master. To do that, clients must know which backups are used by each master; this could potentially be implemented in the same way that CURP clients track witnesses.

Witnesses no longer exist as separate servers; rather, they are incorporated in backups as "witness modules". The role of witness modules is the same as the witness servers in Chapter 4. They handle all record RPCs arriving at backups.

As the first optimization, masters now sync with backups by sending **serialize** requests that convey the serialization orders of client requests, instead of full client requests (an example of a serialize request signature is in Figure 5.2). For serialize requests, backups retrieve the full client requests from their witness modules. This change saves network bandwidth since RPC IDs are much smaller than full client requests. Under certain unfortunate circumstances, a witness module may not have the client request whose serialization order is sent from the master; this can happen if the client is slow at sending the request to the backup or if the witness module previously rejected the request. For such requests, backups should reject the serialize request and ask the master to retry by performing regular syncs with full requests. To avoid this extra step, backups may save the client requests that are rejected from witness modules in some temporary cache.

As the second optimization, masters no longer send **gc** RPCs to witnesses. While handling the **serialize** requests from masters, backups move the requests in the witness module to the backup module. Essentially, sync requests from masters also function as a garbage collection of witness modules. This saving of gc RPCs will improve masters' throughput. As shown in Figure 5.3, the changed garbage collection also reduces the retention period of client requests in witness modules by 1 RTT; this helps to reduce commutativity conflicts.

There may be uncollected records in some occasions; for example, they can be created when a record RPC is delayed significantly and arrives after a backup had already processed the **serialize** requests for them. CURP-H can remove the uncollected records in a way similar to how CURP

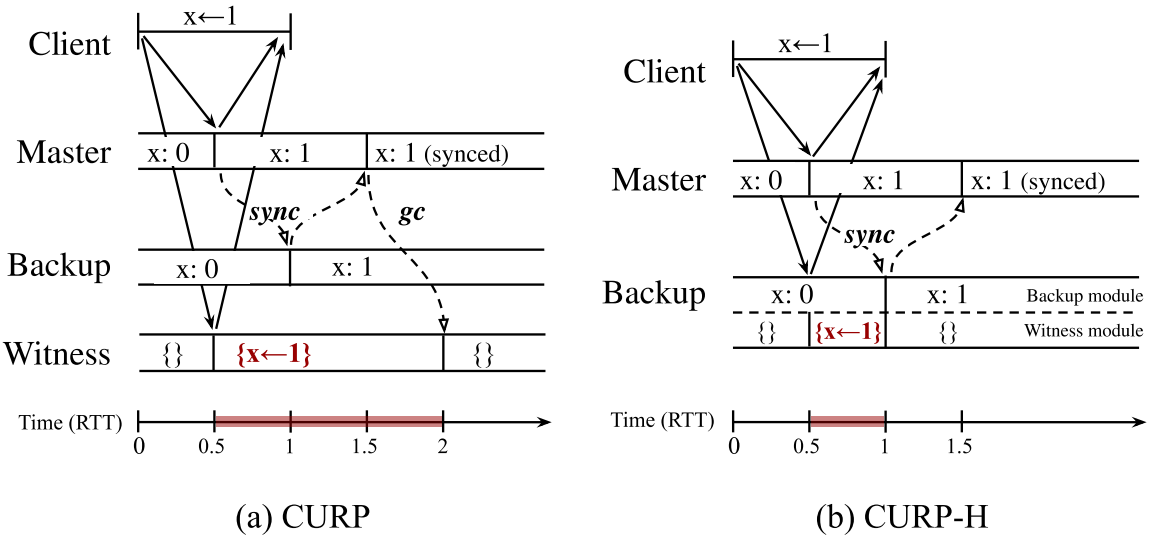

Figure 5.3: Comparison of how long client requests stay in witnesses before being garbage collected. Text in boxes indicates the state of servers. (a) shows that the original CURP takes 1.5 RTT to garbage collect. On the other hand, (b) shows that CURP-H takes only 0.5 RTT to garbage collect. CURP-H cut 1 RTT from CURP by removing the one-way delay for notifying master after syncing and another one-way delay for sending **gc** RPCs.

removes them (described in §4.4.5). The change is that the uncollected requests are reported back to masters through the response messages of serialize RPCs instead of gc RPCs since CURP-H uses serialize RPCs for regular garbage collection. As masters are notified of uncollected requests, the masters retry the requests (most likely filtered by RIFL), and include them in the next serialize requests.

5.1.3 Master Crash Recovery

The recovery process is very similar to the process in Chapter 4. The only difference is that a new master has to pick a single backup and use it for both backup data recovery and witness replay. If we mix a backup module and a witness module from two different backups, recovery may lose some operations. This can occur because witness modules are now allowed to garbage collect before all backup modules are synced (each backup individually move client requests from its witness module to its backup module).

5.1.4 Summary

In this section, we presented how to combine witnesses with backups, which is appropriate if clients can be aware of the backups for each master. This approach can bring many performance benefits: reduced network bandwidth use and fewer commutativity conflicts. Unfortunately, masters in RAMCloud keep changes their backups, so RAMCloud cannot use CURP-H.

5.2 CURP-Q: Making Quorum-based Consensus Protocols Faster

This section introduces CURP-Q, which is an extension of CURP in Chapter 4 to reduce the latency of quorum-based consensus protocols. CURP-Q can be integrated into most consensus protocols with strong leaders such as Raft [58] or Viewstamped Replication [57]. 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-Q can reduce it to 1 RTT.

Similarly to CURP, CURP-Q 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 committing 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.

CURP-Q must deal with two additional problems not present in CURP because of the difference between quorum-based consensus protocols and primary-backup protocols.

- CURP-Q must be available for normal operation even before the recovery of failed replicas. This requirement is a key property/benefit of quorum-based consensus over primary-backup protocols which block normal operation during recovery. §5.2.1 defines the precise guarantee.

- CURP-Q must clean up the speculatively executed operations at decommissioned leaders since they may diverge from the committed operation sequences. This was not a problem in CURP since primary-backup replication does not reuse crashed masters, and new masters are always reconstructed from the data in backups.

The rest of this section illustrates how to modify CURP in Chapter 4 to deal with the two problems introduced by quorum-based consensus protocols.

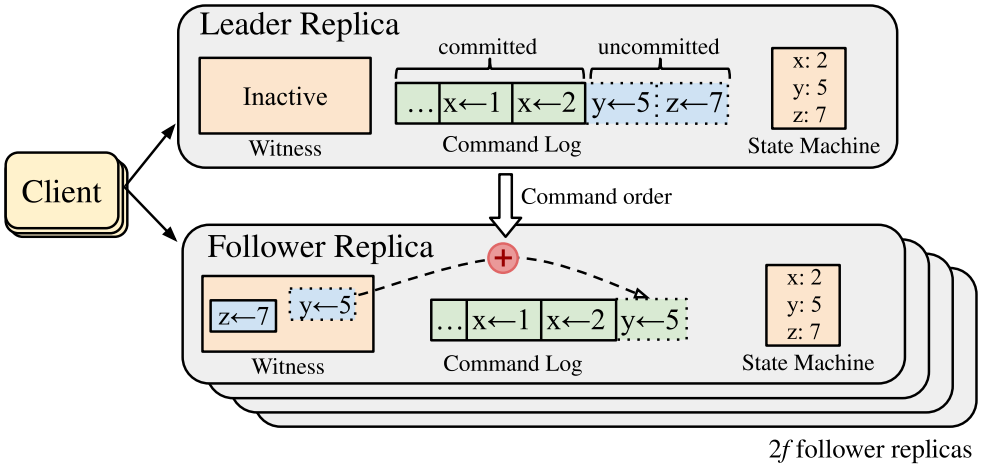

Figure 5.4: In CURP-Q, there are $2f+1$ replicas to tolerate f failures. Each replica has a witness component, a command log, and a state machine. The leader replica determines the order of commands and syncs the order to followers. As in CURP, followers first try to retrieve full client commands from witnesses. If this fails, followers fetch the full requests from the leader.

5.2.1 Architecture and Model

To mask f failures, consensus protocols use $2f+1$ replicas, and systems stay available with up to f failed replicas. For the same guarantee, CURP-Q also uses $2f+1$ replicas, but each replica also has a witness component in addition to existing components for consensus (see Figure 5.4). Although CURP can proceed with $f+1$ available replicas, it needs $f+\lceil f/2 \rceil+1$ replicas (a *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 results (2 RTTs).

5.2.2 Normal Operation

5.2.2.1 Client

Like clients in CURP, clients in CURP-Q multicast their requests to all replicas in parallel. Leader replicas return execution results in 1 RTT, and follower replicas respond to indicate whether their witnesses accepted or rejected the requests.

With acceptances from $f+\lceil f/2 \rceil$ or more witnesses, clients can return the execution results from leaders to applications without waiting for them to be committed. Thus, client operations can complete in 1 RTT. §5.2.3.1 will explain why clients need acceptances from $f+\lceil f/2 \rceil$ witnesses.

Without enough acceptances, clients must send explicit commit requests (similar to sync in CURP) to the leader, which returns after committing the operations to a majority (1 leader and f followers) of replicas.

5.2.2.2 Leader Replica

Like masters in 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.

Witnesses in leaders are inactive, and all client requests are serialized and logged in their command logs directly.

We can apply the optimizations discussed in §5.1 since witnesses are components of each replica. The optimizations help to minimize network bandwidth use and commutativity conflicts. We omit how to apply them here since it's no different from the description in §5.1.

5.2.2.3 Follower Replica

In addition to the standard role of follower replicas in consensus, followers in CURP-Q incorporate witnesses. For requests multicasted by clients, followers serve as witnesses. Witnesses in followers generally behave the same as in CURP (§4.2.2.2). A witness is assigned to a particular leadership, and it rejects all recording requests targeted to prior leaderships. Witnesses in CURP-Q also become immutable during recovery and can be reset to new leaderships.

5.2.3 Recovery (Leader Change)

The recovery of CURP-Q is different from that of CURP in that CURP-Q's recovery considers the information in multiple witnesses, not just one witness. This change allows clients to complete operations after recording to a *superquorum* of witnesses instead of all witnesses in CURP.

5.2.3.1 Why Superquorum?

For a client to complete an operation in 1 RTT, it must be recorded in a *superquorum* of replicas (1 leader and $f + \lceil f/2 \rceil$ witnesses in followers).

First, let's reason about why it's okay to complete an operation before it is recorded to all replicas as in CURP. The reason is that CURP-Q uses more servers ($2f + 1$ for tolerating f failures) than CURP uses ($f + 1$). Thus, there are still $f + 1$ witnesses available even after f failures, and some

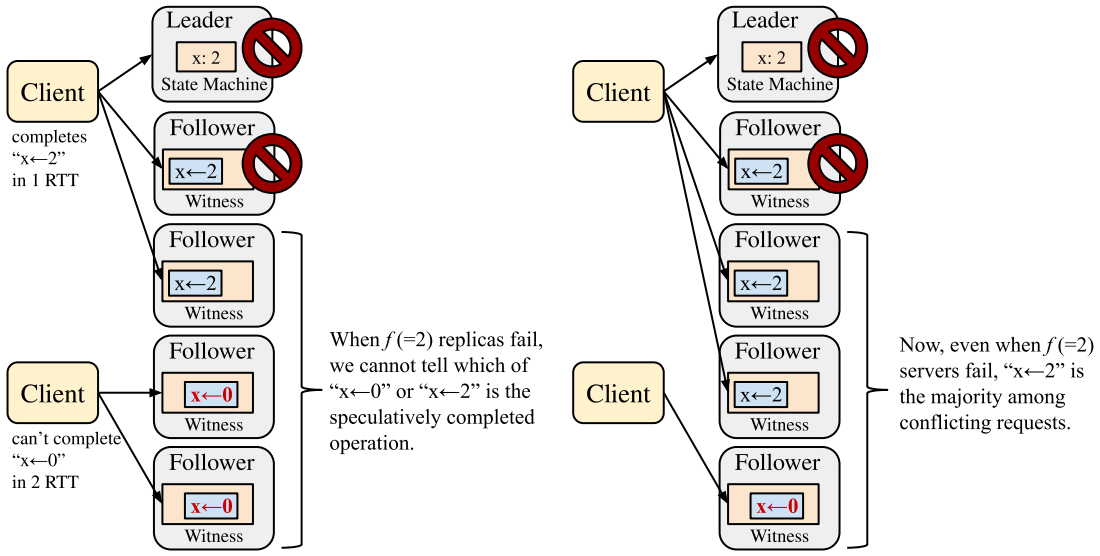

Figure 5.5: Illustration of why clients must wait for a superquorum of replicas. This figure shows the case of 5 replicas ($f = 2$), and it omits irrelevant components in each replica. A superquorum of 5 replicas is 4 of them. Two replicas (the leader and a follower) became unavailable after the top client completed "$x \leftarrow 2$". In (a), the top client completed "$x \leftarrow 2$" merely after recording to a majority of replicas (2 witnesses in addition to the leader). In (b), the top client completed "$x \leftarrow 2$" after recording to a superquorum of replicas (3 witnesses in addition to the leader).

of those available witnesses will have the completed operation; on the other hand, CURP needs to recover even when there are only 1 backup and 1 witness available.

Now, let's reason about why a completed operation must be recorded to a superquorum (a.k.a. fast quorum) of replicas, not just $f + 1$ which is enough to ensure that at least one of the available servers has the completed operation. The reason is that there are many clients concurrently competing to record requests, and these requests may not commute with each other. During recovery, it must be possible to recognize 1 correct (completed speculatively) record out of many non-commutative records. This is similar to other client-driven consensus protocols such as Fast Paxos [46] or Speculative Paxos [68]; they also require clients to wait for a superquorum of replicas so that the future recovery can figure out which command is committed for a particular command slot.

Figure 5.5 illustrates why $f + \lceil f/2 \rceil$ witnesses are the minimum to distinguish the completed request from other requests that don't commute with it. The key idea is ensuring that any completed operation will always outnumber other non-commutative operations during a leadership change.

To ensure that, a client must guarantee the completed request will always appear in a majority of available witnesses during recovery; since each witness doesn't accept multiple non-commutative operations, other non-commutative operations must appear in less than the majority of available witnesses. In the worst case when a leader and $f-1$ witnesses fail, $f+1$ witnesses will be available. The minimum number to become a majority of $f+1$ witnesses is $\lceil f/2 \rceil + 1$. Considering that $f-1$ witnesses may be unavailable during a leadership change, a client must wait for acceptances from $(\lceil f/2 \rceil + 1) + (f-1) = f + \lceil f/2 \rceil$ witnesses.

5.2.3.2 Preparing New Leader

When a leader becomes unavailable, consensus protocols perform leadership changes to recover availability (e.g., leader election in Raft [58] or view change in Viewstamped Replication [57]). With CURP, leadership changes happen in the following steps: (1) a new leader is elected by a standard leadership change process, (2) the leader collects all saved requests from all witnesses, and (3) the new leader selectively picks among the saved requests and replays the selected ones.

When the underlying consensus protocol detects the unavailability of a leader, it should initiate the leadership change protocol. CURP doesn't interfere with this process. However, even after the standard leadership change process, the new leader is not ready to resume servicing normal requests. Since the new leader's command log may miss the last few operations that the previous leader executed speculatively, the new leader must replay them from witnesses before serving normal requests.

For recovery from witnesses, new leaders must first collect saved requests from at least $f+1$ witnesses. Before returning records, each witness must be set to immutable state, just as in CURP; this prevents clients from completing an operation by recording it in a witness whose data won't be checked during future recoveries.

After collecting from witnesses, a new leader counts the appearance of each client request and replays it if it appears in more than $\lceil f/2 \rceil + 1$ witnesses. This selective replay is required to ensure commutativity among replayed requests as mentioned in §5.2.3.1.

Once replaying from witnesses is completed, the leadership change process is completed, and the new leader can resume servicing normal requests. As follower replicas notice the completion of the leadership change, they should reset their witnesses for the new leadership. For tracking leaderships, witnesses and clients may use the existing identifiers such as a term number in Raft or a view-number in Viewstamped Replication.

5.2.3.3 Cleaning Up Old Leader

After the leadership change, the state machine of the decommissioned leader could have diverged from other replicas since the leader could have speculatively executed some operations that couldn't be recorded to enough number of witnesses or committed to a majority of replicas.

To fix this, a leader replica has to periodically make a checkpoint by snapshotting its state machine (this is a common practice in consensus), and the decommissoned leader must reload from a checkpoint that does not have any speculative executions (i.e., any operations executed for the checkpoint were committed eventually). However, reloading from a checkpoint is an expensive operation. We can avoid reloading from checkpoints if the leadership change was not caused by a crash or disconnect of the old leader; Instead of requiring the old leader to reload from a checkpoint, the new leader can fetch and commit all uncommitted operations in the old leader's command log to prevent the divergence of state machines.

5.2.3.4 Safety

For correctness, the client requests replayed from witnesses during recovery must

- include all completed operations (i.e., operations whose executions are observed by any client) that are not yet committed in a majority of replicas' command logs. Otherwise, recovery may lose the completed operations.

- commute with each other. Otherwise, recovery may result in an inconsistent state.

Since clients complete operations with speculative results only after recording to $f + \lceil f/2 \rceil$ witnesses, all completed-but-not-yet-committed operations are guaranteed to exist in a majority ($\lceil f/2 \rceil + 1$) of any combination of $f + 1$ witnesses. Thus, if a client could complete an operation, it must be replayed during recovery since it is guaranteed to appear more than $\lceil f/2 \rceil + 1$ available witnesses.

Also, any client requests that don't commute with the completed operations cannot exist in more than $\lfloor f/2 \rfloor$ (less than the majority of any quorum). The reason is that a completed operation is already in $f + \lceil f/2 \rceil$ witnesses (out of $2f$ witnesses), and those witnesses must have rejected the non-commutative request. Thus, during recovery, all requests that appear in a majority ($\lceil f/2 \rceil + 1$) from any combination of $f + 1$ witnesses are guaranteed to be commutative.

The two properties can replace (Rule 1) and (Rule 2) in the informal proof of CURP in §4.3.

5.2.4 Zombies

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 applying CURP-Q since an operation must be committed in a majority before being executed and at least one replica would know the leadership has been changed. To prevent clients from completing operations with a zombie 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.

5.2.5 Read Optimization

CURP-Q 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-Q does not interfere with this read lease mechanism.

Chapter 6

Future Work

6.1 Replication on Programmable Data-plane

New networking hardware opens up opportunities for improving the performance of distributed systems. In the last few years, researchers have started putting more functionality in networking devices, most notably in programmable NICs (a.k.a. SmartNICs).

I suspect that SmartNICs can be used to offload the job of replication. As a preliminary investigation, Sean Choi and I implemented CURP's witness on programmable NICs to offload CPU cores and avoid costly PCI-E crossings [19]. Our prototype on ASIC-based SmartNICs from Netronome [5] showed a performance comparable to running witnesses on CPU (Figure 6.1). Although the throughput on SmartNIC isn't as good as the witness throughput on RAMCloud (which uses kernel bypass and avoids serialization), the newer SmartNICs have $\sim$2x more cores at $\sim$2x higher frequency than the one we used. Considering that the computation power of SmartNICs is improving rapidly, there is a great chance that we can completly offload the entire replication stack (including client-side and master-side) to the programmable dataplane on NICs. One concern is that we found the TCP round-trip time of SmartNICs are generally higher than that of latency-optimized NICs from Mellanox. We are not sure if the high latency is a fundamental limitation of SmartNICs or not.

6.2 Fast Geo-replication with Synchronized Clocks

Fast replication protocols can be categorized into two types: those that rely on commutativity (e.g., CURP, Generalized Paxos, Egalitarian Paxos), and those that rely on special networking to deliver request packets to replicas almost in the same order (e.g., Speculative Paxos, Network-Ordered

		On SmartNIC	On CPU	**RAMCloud** in §4.5
Latency (µs)	Avg	30.91	61.28	7
	p99	36.72	80.63	7
Throughput (kops)		758	113	1270

Table 6.1: Performance of witnesses on SmartNICs. We measured latency and throughput of witness **record** RPCs. Our test servers equipped with two Intel Xeon 5117 14-core processors operating at 2.0 GHz, 32 GB DDR4 2,666 MT/s Dual-Ranked RAM and 120 GB SATA SSD. Both client and witnesses used Netronome Agilio CX 2x10Gb SmartNICs [5] with 56 RISC cores running at 633 MHz with 2 GB of on-NIC RAM. All servers are connected via a 10 Gbps Arista DCS-7124S switch. "RAMCloud" shows the performance of the witness implementation on RAMCloud, which we already showed in §4.5; it uses low-latency NICs with kernel-bypass and binary protocols to avoid the serialization overhead.

Paxos). Unfortunately, I believe that none of them work well for geo-replication; commutativity conflicts will happen too often (as seen in §4.7) for the protocols relying on commutativity, and it's challenging to use special networking on wide-area networks for the protocols relying on in-order message deliveries [68, 49, 9, 26].

Mostly in-order message deliveries have been used for fast replication [68, 9, 26], typically with special networking hardware or settings (e.g., software-defined networking, small LANs with symmetric message delays). They assume that broadcasts of requests from clients are delivered mostly in-order and let state machines optimistically handle requests without waiting for the ordering information from the leader. Clients check if their requests are delivered in-order by comparing the hash values of the ordered lists of all operations executed at each server; if a superquorum of replicas return the same hash values, the client can complete the operation by returning the execution result to the application. However, if the suspected order turns out to be different from the order determined by the leader, the state machine must rollback to correct the inconsistency. Unfortunately, on WANs, there will be more out-of-order deliveries since message reordering and asymmetric message delays are common, and it is difficult, on WANs, to install special networking devices that enforce in-order deliveries. Thus, most client requests will not be able to complete in 1 RTT.

Recently, accurately synchronizing clocks of network-connected machines became possible [29]. I suspect that synchronized clocks can be used for enabling mostly-in-order message deliveries even on WANs. With synchronized clocks, each client can now measure and track one-way latency to each replica, which may be quite different from the half of measured RTTs (which were available even before synchronized clocks) because of asymmetric message delays on WANs. More accurate one-way delay measurements can be used for atomic broadcast; when a client broadcasts its request,

it asks replicas to delay the delivery of the request to state machines until the time when all replicas are expected to receive the request, which can be calculated by the client's broadcast time plus the maximum of measured one-way delays from the client to any replica. Thus, we can achieve mostly-in-order delivery to all replicas, even on WANs.

There are some drawbacks and challenges of this approach. The first challenge is the message delay may change continuously. If the variation of message delay within 1 RTT is larger than the gap between request arrivals at each replica, the requests may get delivered out of order. We can mitigate this problem in two ways: (1) adding a delay in addition to the maximum of measured one-way delays to each replica (2) relying on commutativity to decrease the effective request arrival rate (now, delivery order only matters among noncommutative requests). The second challenge is avoiding state machine rollback, which is known to be very expensive. Instead of letting each replica to execute client requests immediately, we can leverage the idea of witnesses in CURP, saving client request without executing until the serialization order arrives from the leader.

In summary, synchronized clocks enable cherry-picking benefits of the two types of fast replication: no rollback from those relying on commutativity and tolerance to skewed workloads from those relying on in-order message deliveries. I suspect that the hybrid of these technologies can birth a fast replication protocol which works well with any network and workloads.

Chapter 7

Conclusion

In this dissertation, I presented three consistency mechanisms for large-scale and low-latency distributed systems. All of them provide the strongest form of consistency, called linearizability, with a minimal impact on either latency or scalability. RIFL provides a general-purpose mechanism for converting at-least-once RPC semantics to exactly-once semantics. RIFL-TX is a distributed transaction mechanism with linearizability (a.k.a. strict serializability in the database community). CURP allows clients to complete update operations on replicated data in 1 RTT instead of the traditional 2 RTTs. With these results, we could bring consistency to systems that demand large scale ($\sim$1 million nodes) and low latency ($\sim$10 µs).

In addition to scalability and latency, avoiding complexity in systems often becomes a reason for settling with weak consistency. However, the burden of achieving consistency doesn't disappear; rather, it will be amplified and given to application developers. For example, Facebook's social graph database doesn't support distributed transactions; when a person (say, Alice) accepts a friend request from Bob, two edges are inserted (an edge from Bob to Alice, and another edge from Alice to Bob). Without distributed transaction support, one of the insertions may fail, and one-sided friendship is possible. To fix this anomaly, Facebook created a separate infrastructure that scans all friendship data and fixes any inconsistencies. In this example, a simple operation (becoming friends) ended up with an additional complex mechanism to deal with the weakly-consistent system. As a general principle, system designers should try to encapsulate complexity within lower levels of system software, so that application developers are not exposed to it [61]. Providing consistency should be a job of systems.

Appendix A

Additional Materials

A.1 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. [49, 68, 84]

Bibliography

[1] Low Latency 10 and 40 Gigabit Ethernet End-to-End Solutions. Tech. rep., Mellanox Technologies, Sunnyvale, CA, 2011. 4

[2] MongoDB, Apr. 2014. `http://www.mongodb.org/`. 63

[3] RAMCloud Git Repository, June 2015. `https://github.com/PlatformLab/RAMCloud.git`. 22

[4] GlusterFS. `https://www.gluster.org`, 2017. Accessed: 2017-09-22. 50

[5] Agilio SmartNIC. `https://www.netronome.com/products/`, 2018. 95, 96

[6] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 159–174. 5, 23, 42

[7] 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. 81

[8] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234. 3, 11, 42

[9] 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. 50, 81, 96

[10] BALLMER, S. Keynote speech at Worldwide Partner Conference, July 2013. 1

[11] BARROSO, L., MARTY, M., PATTERSON, D., AND RANGANATHAN, P. Attack of the Killer Microseconds. *Commun. ACM 60*, 4 (Mar. 2017), 48–54. 4

[12] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 49–65. 4

[13] BERNSTEIN, P. A., AND NEWCOMER, E. *Principles of transaction processing*. Morgan Kaufmann, 2009. 42

[14] BHIDE, A., ELNOZAHY, E. N., AND MORGAN, S. P. A highly available network file server. In *Proc. Winter 1991 USENIX Conference* (1991), vol. 91, pp. 199–205. 43

[15] BIRMAN, K., SCHIPER, A., AND STEPHENSON, P. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst. 9*, 3 (Aug. 1991), 272–314. 81

[16] 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. 3, 81

[17] 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. 53

[18] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26. 3

[19] CHOI, S., PARK, S. J., SHAHBAZ, M., PRABHAKAR, B., AND ROSENBLUM, M. Toward Scalable Replication Systems with Predictable Tails Using Programmable Data Planes. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019* (Beijing, China, 2019), APNet '19, ACM, pp. 78–84. 95

[20] 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. 81

[21] 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. 73

[22] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google&Rsquo;s Globally Distributed Database. *ACM Trans. Comput. Syst. 31*, 3 (Aug. 2013), 8:1–8:22. 11, 42

[23] 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* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220. 3, 11, 42, 81

[24] 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. 81

[25] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414. 1, 4, 10, 12, 13, 42

[26] 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. 50, 81, 96

[27] FOUNDATION, C. N. C. gRPC: A high performance, open-source universal RPC framework. `https://grpc.io`. 46

[28] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-based Scalable Network Services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 78–91. 11, 42

[29] GENG, Y., LIU, S., YIN, Z., NAIK, A., PRABHAKAR, B., ROSENBLUM, M., AND VAHDAT, A. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 81–94. 96

[30] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proc. 19th ACM symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43. 53

[31] GRAY, C., AND CHERITON, D. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1989), SOSP '89, ACM, pp. 202–210. 15

[32] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. 12

[33] 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. 73

[34] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492. 2, 9, 11

[35] 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), USENIX ATC '10, USENIX Association, pp. 11–11. 20, 43, 53

[36] INFORMATION SCIENCES INSTITUTE. RFC 793: Transmission Control Protocol, 1981. Edited by Jon Postel. Available at `https://www.ietf.org/rfc/rfc793.txt`. 43

[37] JUSZCZAK, C. Improving the Performance and Correctness of an NFS Server. In *Proc. Winter 1989 USENIX Conference* (1990), pp. 53–63. 43

[38] 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. 81

[39] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb 2019), USENIX Association, pp. 1–16. 4

[40] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow. 1*, 2 (2008), 1496–1499. 10, 12, 37, 42

[41] KRASKA, T., PANG, G., FRANKLIN, M. J., MADDEN, S., AND FEKETE, A. MDCC: Multi-data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 113–126. 81

[42] KUNG, H.-T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS) 6*, 2 (1981), 213–226. 24

[43] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40. 3, 11, 42

[44] LAMPORT, L. The Part-Time Parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169. 12, 51

[45] LAMPORT, L. Generalized Consensus and Paxos. Tech. rep., March 2005. 51, 79

[46] LAMPORT, L. Fast Paxos. *Distributed Computing 19* (October 2006), 79–103. 51, 79, 91

[47] 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. 5, 6, 58, 60, 67, 68

[48] LI, B., CUI, T., WANG, Z., BAI, W., AND ZHANG, L. Socksdirect: Datacenter Sockets Can Be Fast and Compatible. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China, 2019), SIGCOMM '19, ACM, pp. 90–103. 4

[49] 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. 50, 51, 54, 80, 96, 99

[50] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11, ACM, pp. 401–416. 11, 42

[51] LU, H., VEERARAGHAVAN, K., AJOUX, P., HUNT, J., SONG, Y. J., TOBAGUS, W., KUMAR, S., AND LLOYD, W. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, CA, 2015), SOSP '15, ACM, pp. 295–310. 2

[52] memcached: a Distributed Memory Object Caching System, Jan. 2011. `http://www.memcached.org/`. 1

[53] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proc. 2018 ACM SIGCOMM Conference* (Budapest, Hungary, 2018), SIGCOMM '18, ACM. 1

[54] 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. 50, 80

[55] 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. 81

[56] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398. 1

[57] 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. 88, 92

[58] 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. 43, 51, 53, 80, 88, 92

[59] 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. 50, 70

[60] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 361–378. 1

[61] OUSTERHOUT, J. *A Philosophy of Software Design*, 1st ed. Yaknyam Press, 2018. 98

[62] 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. 1, 4, 5, 10, 13, 22, 32, 50, 53, 55, 63, 67, 68, 70

[63] PARK, S. J., AND OUSTERHOUT, J. Exploiting Commutativity For Practical Fast Replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), USENIX Association, pp. 47–64. 6

[64] PAVLO, A. Personal communication, March 22 2015. 38

[65] PAZ, O. InfiniBand Essentials Every HPC Expert Must Know. `http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1/1_Mellanox.pdf`, April 2014. 4

[66] PESSACH, Y. *Distributed Storage: Concepts, Algorithms, and Implementations*. Amazon, 2013. 2

[67] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 288–301. 3, 11, 42

[68] 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. 50, 51, 54, 80, 91, 96, 99

[69] PREKAS, G., KOGIAS, M., AND BUGNION, E. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 325–341. 4

[70] QIN, H., LI, Q., SPEISER, J., KRAFT, P., AND OUSTERHOUT, J. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carslbad, CA, 2018 (to appear)), USENIX Association. 1

[71] 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. 76

[72] SANFILIPPO, S., ET AL. Redis. `https://redis.io/`, 2015. Accessed: 2017-04-18. 3, 50, 63, 74

[73] SCHNEIDER, F. B. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319. 50

[74] SETHI, R. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM) 29*, 2 (1982), 394–403. 24

[75] 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. 53

[76] 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. 63

[77] SLEE, M., AGARWAL, A., AND KWIATKOWSKI, M. Thrift: Scalable cross-language services implementation. 46

[78] SPRENKER, L., AND HAMMOND, B. Redis C++ Client. `https://github.com/mrpi/redis-cplusplus-client`, 2011. Accessed: 2017-04-20. 76

[79] 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. 81

[80] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC-C Benchmark (Revision 5.11). `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, 2010. 37

[81] VOGELS, W. Eventually Consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44. 81

[82] ZENG, J. H. The Computer Networks behind the Social Network. Invited talk at 3rd Asia-Pacific Workshop on Networking, August 2019. 1

[83] 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. 81

[84] ZHAO, W. Fast Paxos Made Easy: Theory and Implementation. *International Journal of Distributed Systems and Technologies (IJDST) 6*, 1 (2015), 15–33. 99