Implementing Linearizability at Large Scale and Low Latency

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

Stanford University, [†]NEC

Abstract

Linearizability is the strongest form of consistency for concurrent systems, but most large-scale storage systems settle for weaker forms of consistency. 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. RIFL handles data migration by associating linearizability metadata with objects in the underlying store and migrating metadata with the corresponding objects. It uses a lease mechanism to implement garbage collection for metadata. We have implemented RIFL in the RAMCloud storage system and used it to make basic operations such as writes and atomic increments linearizable; RIFL adds only 530 ns to the 13.5 μs base latency for durable writes. We also used RIFL to construct a new multi-object transaction mechanism 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.

1 Introduction

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 [12]. 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.

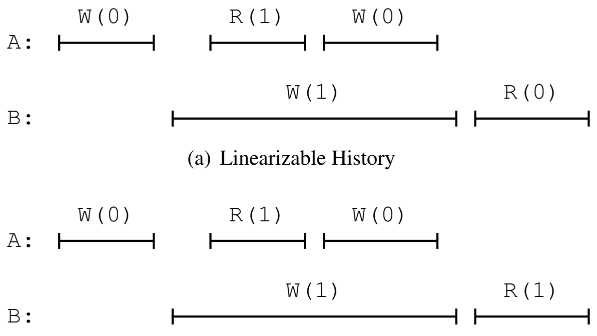
In this paper we describe RIFL (Reusable Infrastructure for Linearizability), which is a mechanism for ensuring "exactly-once semantics" in large-scale systems. RIFL records the results of completed remote procedure calls (RPCs) durably; if an RPC is retried after it has completed, RIFL ensures that the correct 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 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 [21] and

[*] These authors contributed equally to this work.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815416



(a) Linearizable History

(b) Non-Linearizable History

Figure 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 [12]. 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. Horizontal bars indicate the time duration of each operation.

FaRM [8], 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 [15] on the TPC-C benchmark, providing at least 10x lower latency and 1.35x–7x as much throughput.

2 Background and Goals

Linearizability is a safety property concerning the behavior of operations in a concurrent system. A collection of operations is *linearizable* if each operation 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 1 shows examples of linearizable and non-linearizable operation histories. Linearizability is the strongest form of consistency for concurrent systems.

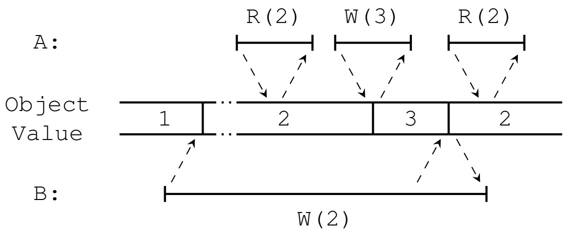


Figure 2: Non-linearizable behavior caused by crash recovery. In this example, the server completes a write from Client B but crashes before responding. After the server restarts, Client B reissues the write, but meanwhile Client A has written a different value. As a result, Client A observes the value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

Early large-scale storage systems settled for weak consistency models in order to focus on scalability or partition-tolerance [7, 9, 17, 23], but newer systems have begun providing stronger forms of consistency [3, 6, 19]. 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.
- Two-phase commit and/or consensus protocols [11, 18], 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 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 [15] and FaRM [8], 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 8).

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 [21] and FaRM [8], 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 9) and servers can store RIFL's metadata reliably.

3 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 4 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 delete completion records. Completion records can also be garbage collected when a client crashes. In this situation the client will not provide acknowl-

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

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 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.

Figure 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.

edgments, 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 [10]: 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. The lease approach can still result in ambiguous situations where a client cannot tell whether an RPC completed, but it allows these situations to be detected and reported.

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

4 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 5.

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 3). The second module, LeaseManager, runs on both clients and servers to manage client leases (Figure 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 5).

4.1 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.

4.2 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.

4.3 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 [13] so it can be recovered after crashes. When a client issues its first RPC and invokes the `getClientId` method of LeaseManager (Figure 4), LeaseManager contacts the lease server to allocate a new lease. LeaseManager automatically renews the lease and deletes it when the client exits. 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 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 lease is near to, or past, expiration. 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. The lease server returns its current cluster clock value whenever it responds to a client request for lease creation or renewal, and the client includes this time in RPC requests. Servers use this information for two purposes. First, it allows them to compute a lower bound on the lease server's value of the cluster clock. Second, it allows servers to estimate when the client's lease will expire, given knowledge of the lease interval (which is a fixed system-wide parameter). Using this approach, a server can almost always determine trivially that a client's lease is still valid. If a lease is near to expiration, then the server must contact the lease manager to validate the lease; this also returns the current cluster clock value. A server only expires a lease if it has confirmed the expiration with the lease server. The cluster clock mechanism includes several other details, which we omit here because of space limitations.

4.4 Migration

As discussed in Section 3, 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 RIFL's ResultTracker module so that it can reconstruct its in-memory metadata that maps from RPC identifiers to completion records.

5 Implementation

In order to evaluate RIFL, we have implemented it in RAMCloud [21], 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 [1].

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 [21], 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 stores an entry for the completion record's RPC id.

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, 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 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 4.

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.

6 Implementing Transactions with RIFL

This section describes how we used RIFL to implement a new multi-object distributed transaction mechanism in RAMCloud. Transactions are a more complex use case for RIFL, since they 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 RAMCloud transactions is based on Sinfonia [2]; 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; we have omitted a few details and corner cases because of space limitations.

6.1 APIs

The application-visible API for RAMCloud transactions 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. 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 [24] using optimistic concurrency control [16].

6.2 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. 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, the commit aborts and no updates occur.

6.3 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 Figure 6). 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;

prepare(*tableId*, *key*, *version*, *operation(READ, WRITE, DELETE)*, *newValue*, *rpcId*, *firstIncomplete*, *leaseInfo*, *allObjects*) → {PREPARED, ABORT},

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. *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 6: The APIs for the RPCs used to implement the commit protocol for RAMCloud transactions.

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 tombstone record to the RAMCloud log to nullify the lock record.

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. This is 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. The participant 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` 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 7).

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.

6.4 Client crashes

If the client crashes before completing all of the `decision` RPCs, then a server must complete the process on its behalf. RAMCloud transactions use 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`.

6.5 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.

6.6 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 RAMCloud transactions 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 6.4. 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.

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 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.

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.

6.7 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.

7 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 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 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 [21] for details).

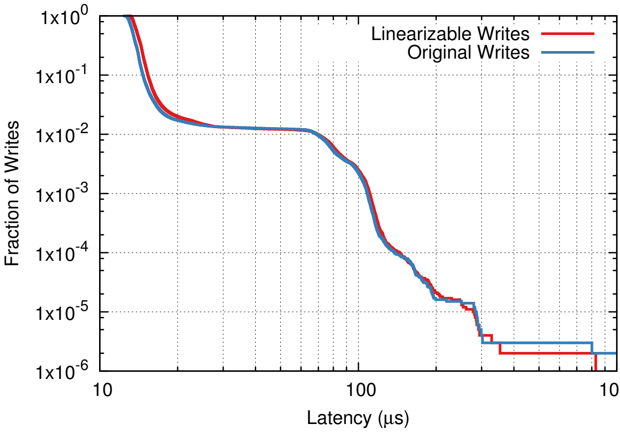


Figure 7: Reverse cumulative distribution of latency for 100B random RAMCloud write operations with and without RIFL. Writes were issued sequentially by a single client to a single server in a 4-server cluster. A point (x, y) indicates that y of the 1M measured writes took at least x µs to complete. "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.

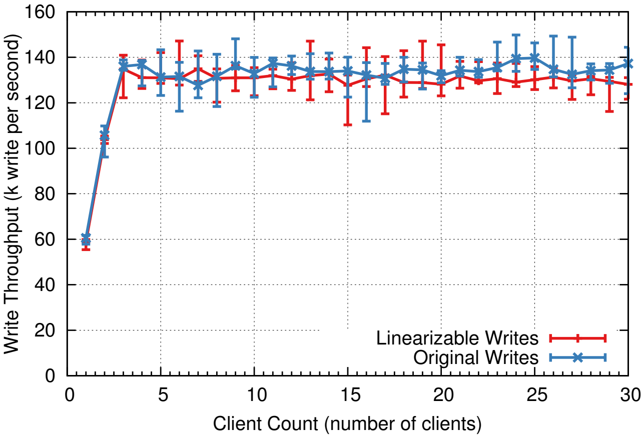


Figure 8: The aggregate throughput for one server serving 100B RAMCloud writes with and without RIFL, as a function of the number of clients. Each client repeatedly issued random writes back to back to a single server in a 4-server cluster. "Original Write" refers to the base RAMCloud system before adding RIFL. Each experiment was run 5 times. The data points show the median values; error bars show min and max.

7.1 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 7 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 gen-

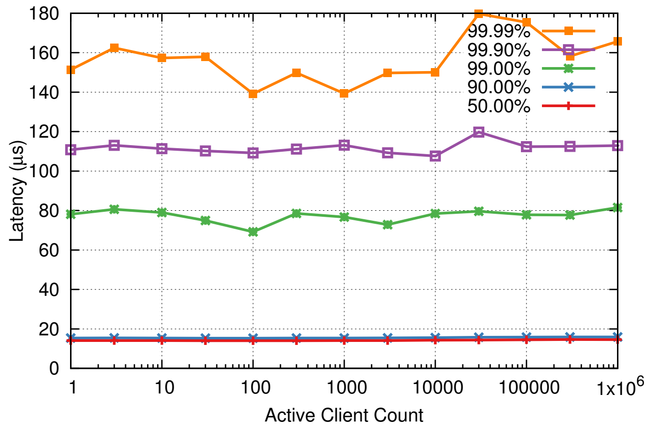


Figure 9: Latency of linearizable writes as a function of the amount of client state managed by the server. A single client application simulated many virtual clients, each with its own RequestTracker and client id. The client issued a series of random 100B write requests, with each write request using 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.

erally, the overall shape of the latency distribution did not change with the addition of linearizability.

Figure 8 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).

7.2 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 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 9). 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.

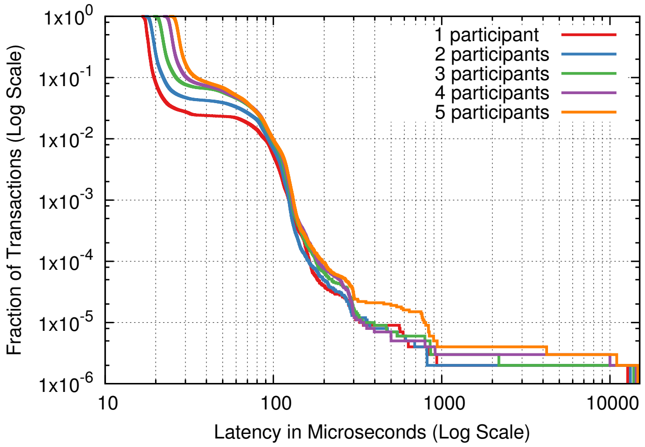


Figure 10: Reverse cumulative distribution of transaction commit latency measured from the beginning of the commit process to the completion of all `prepare` RPCs. Each transaction reads and writes one randomly chosen object per participant; the cluster contained 10 servers. A point (x, y) indicates 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.

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.

7.3 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 wrapper

RIFL also significantly simplified the implementation of distributed transactions in RAMCloud. This was discussed in detail in section 6.5.

7.4 Evaluating RIFL-based Transactions

Figure 10 shows the 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 while a transaction with 5 objects commits in 27.3 µs. Fig-

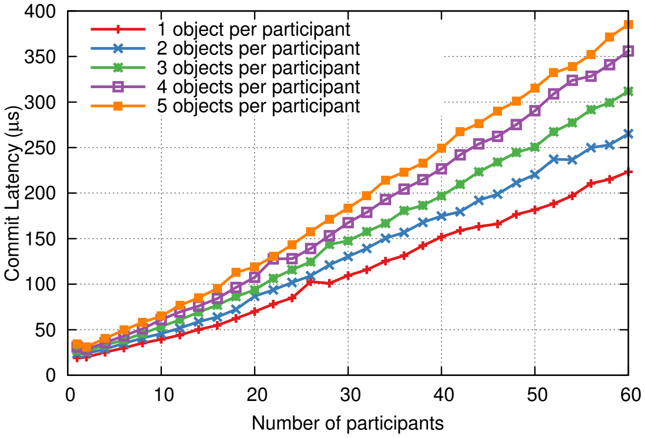


Figure 11: Median transaction commit latency for a single client measured from the beginning of the commit process to the completion of all `prepare` RPCs, as a function of the number of participants. For instance, with 5 objects per participant and 60 participants the transaction is committing 300 objects spanning 60 participants. All experiments used a cluster with 60 servers.

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: 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.

ure 11 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 12 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 12 also shows that the single-server optimization described in Section 6 improves throughput by about 40%.

7.5 Comparing Transaction Performance with H-Store

We implemented the TPC-C benchmark [25] with RAMCloud transactions and used it to compare performance between RAMCloud and H-Store [15], 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). 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 *NewOrder* transactions per minute (TpmC).

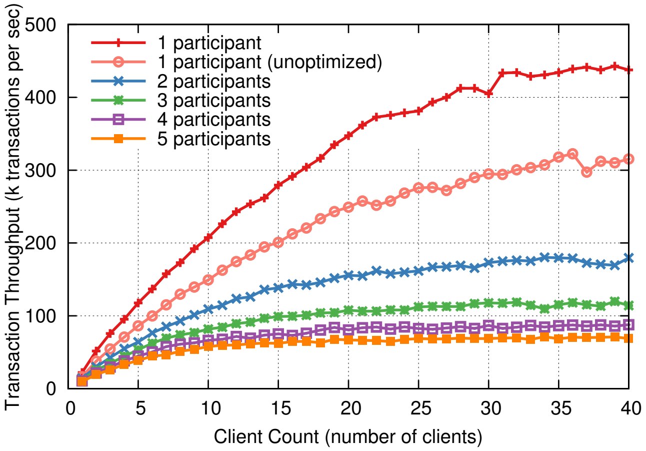


Figure 12: Aggregate transaction throughput for a cluster of 10 servers as a function of the number of clients. Each client repeatedly creates and commits transactions back to back where each transaction reads and writes one 100-byte object per participant; participants were chosen at random for each 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.

Our implementation of the TPC-C benchmark for RIFL 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 [22] 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 13 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 13. In this configuration H-Store's latency is more than 10x higher than RAMCloud's when durability is enabled in H-Store, and 20% 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

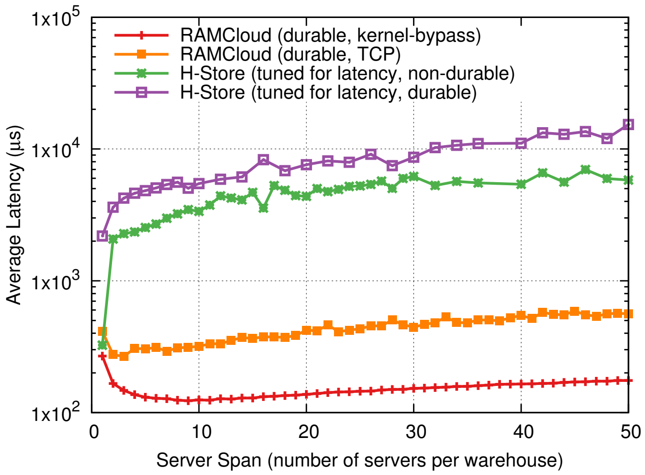


Figure 13: Average latency of TPC-C NewOrder transactions with a single warehouse, as a function of the degree of sharding for the warehouse. "Server Span" of 10 indicates that the data of a single warehouse is sharded across 10 servers. H-Store was tuned for low latency and measured with durability both enabled and disabled. RIFL-based transactions were measured on both RAMCloud's standard kernel-bypassing transport as well as over kernel TCP.

efficient for single-server transactions because it uses predefined 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 13 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 13 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 if it used kernel-bypass.

Figure 14 shows latency and throughput for the TPC-C benchmark with the standard mix detailed in Table 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.

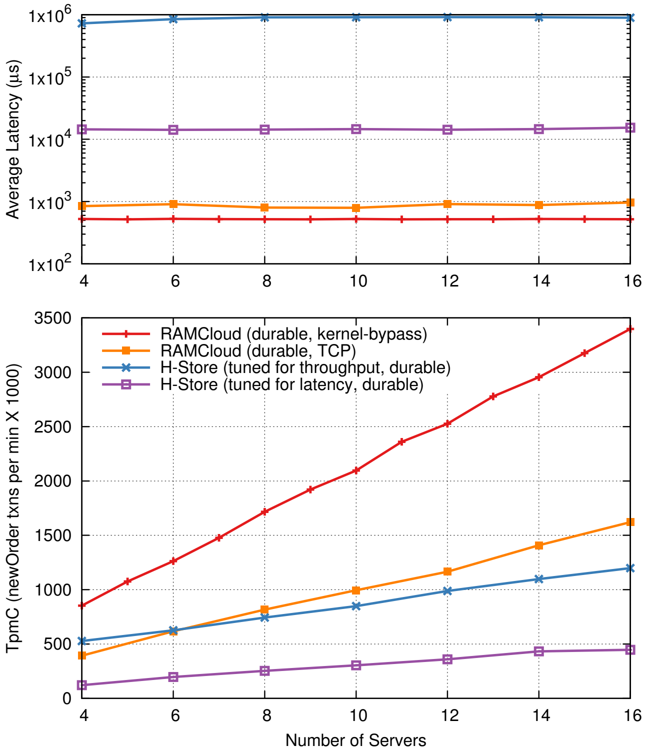


Figure 14: Latency and aggregate throughput of NewOrder transactions in the TPC-C benchmark, as a function of the system scale. The benchmark used the transaction mix in Table 2, with each warehouse stored on a single server. Increasing the number of servers also increased the number of warehouses. Roughly 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.

RIFL outperforms H-Store for TPC-C both in latency and throughput. RIFL's advantage is smallest when comparing throughput between RAMCloud over TCP and H-Store optimized for throughput; even so, RIFL'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.

7.6 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 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. We believe that adaptive batching techniques such as those in the IX system [4] could improve RAMCloud's (and RIFL's) throughput with little impact on latency; we leave this to future work.

8 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 [7, 9, 17, 23]. 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 [3, 6, 19]. Some of these systems offer semantics close to linearizability, but it is difficult to tell whether any are truly linearizable. For example, Spanner [6] 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 [5], 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 [15], Spanner [6], and FaRM [8] 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 [2] 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 [13] and Raft [20] 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 [14], 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.

9 Client Failures and the Ultimate Client

RIFL only guarantees exactly-once semantics for an operation if its 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 the side effects of operations to be recognizable to humans, so that users can determine on their own whether each operation completed. For example, if a user doesn't receive an acknowledgment that a request completed, he/she can check to see whether the desired reservation exists or the desired order is present in his/her order history. If not, then the user re-invokes the operation.

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 avoid 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.

10 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 8, 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.

11 Conclusion

In this paper we have 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.

Acknowledgments

This work was supported by C-FAR (one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA), and by grants from Emulex, Facebook, Google, Huawei, Inventec, NEC, NetApp, Samsung, and VMware. Seo Jin Park was supported by a Samsung Scholarship. The presentation in the paper benefited from comments by the anonymous conference reviewers and our shepherd, Steven Hand. Diego Ongaro introduced us to the idea that idempotent operations are unsafe in the presence of concurrency.

References

[1] RAMCloud Git Repository. https://github.com/PlatformLab/RAMCloud.git.

[2] Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. *ACM Transactions on Computer Systems 27*, 3 (Nov. 2009), 5:1–5:48.

[3] 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.

[4] 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.

[5] Bernstein, P. A., and Newcomer, E. *Principles of transaction processing*. Morgan Kaufmann, 2009.

[6] 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's Globally Distributed Database. *ACM Trans. Comput. Syst. 31*, 3 (Aug. 2013), 8:1–8:22.

[7] 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 *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 205–220.

[8] 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.

[9] 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.

[10] Gray, C., and Cheriton, D. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1989), SOSP '89, ACM, pp. 202–210.

[11] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers

Inc., San Francisco, CA, USA, 1992.

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

[13] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference* (Berkeley, CA, USA, 2010), USENIX ATC '10, USENIX Association, pp. 11–11.

[14] INFORMATION SCIENCES INSTITUTE. RFC 793: Transmission control protocol, 1981. Edited by Jon Postel. Available at `https://www.ietf.org/rfc/rfc793.txt`.

[15] 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 (Aug. 2008), 1496–1499.

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

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

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

[19] 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* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[20] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 305–319.

[21] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., ET AL. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 7.

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

[23] 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.

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

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