The RAMCloud Storage System

JOHN OUSTERHOUT, ARJUN GOPALAN, ASHISH GUPTA, ANKITA KEJRIWAL, COLLIN LEE, BEHNAM MONTAZERI, DIEGO ONGARO, SEO JIN PARK, HENRY QIN, MENDEL ROSENBLUM, STEPHEN RUMBLE, and RYAN STUTSMAN, Stanford University

RAMCloud is a storage system that provides low-latency access to large-scale datasets. To achieve low latency, RAMCloud stores all data in DRAM at all times. To support large capacities (1 PB or more), it aggregates the memories of thousands of servers into a single coherent key-value store. RAMCloud ensures the durability of DRAM-based data by keeping backup copies on secondary storage. It uses a uniform log-structured mechanism to manage both DRAM and secondary storage, which results in high performance and efficient memory usage. RAMCloud uses a polling-based approach to communication, bypassing the kernel to communicate directly with NICs; with this approach, client applications can read small objects from any RAMCloud storage server in less than 5 µs; durable writes of small objects take about 15 µs. RAMCloud does not keep multiple copies of data online; instead, it provides high availability by recovering from crashes very quickly (1–2 seconds). RAMCloud's crash recovery mechanism harnesses the resources of the entire cluster working concurrently, so that its performance scales with cluster size.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Distributed*; D.4.2 [**Operating Systems**]: StorageManagement—*Main memory*; *Secondary storage*; *Distributed memories*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

General Terms: Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: Datacenters, large-scale systems, low latency, storage systems

1. INTRODUCTION

DRAM and its predecessor, core memory, have played an important role in storage systems since the earliest days of operating systems. For example, early versions of UNIX in the 1970s used a cache of buffers in memory to improve file system performance [Ritchie and Thompson 1974]. Over the last 15 years the use of DRAM in storage systems has accelerated, driven by the needs of large-scale Web applications. These applications manipulate very large datasets with an intensity that cannot be satisfied by disk and flash alone. As a result, applications are keeping more and more of their long-term data in DRAM. By 2005 all of the major Web search engines kept their search indexes entirely in DRAM, and large-scale caching systems such as memcached [mem 2011] have become widely used for applications such as Facebook, Twitter, Wikipedia, and YouTube.

Although DRAM's role is increasing, it is still difficult for application developers to capture the full performance potential of DRAM-based storage. In many cases DRAM is used as a cache for some other storage system such as a database; this approach forces developers to manage consistency between the cache and the backing store, and its performance is limited by cache misses and backing store overheads. In other cases, DRAM is managed in an application-specific fashion, which provides high performance

This work was supported by the Gigascale Systems Research Center and the Multiscale Systems Center (two of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program), by C-FAR (one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA), by the National Science Foundation under grant No. 096385, and by Stanford Experimental Data Center Laboratory affiliates Cisco, Emulex, Facebook, Google, Inventec, Mellanox, NEC, NetApp, Samsung, SAP, and VMware. Stephen Rumble was supported by a Natural Sciences and Engineering Research Council of Canada Postgraduate Scholarship. Diego Ongaro was supported by The Junglee Corporation Stanford Graduate Fellowship
Author's addresses: TBD.
This document is currently under submission for publication. It can be cited as "Stanford Technical Report, October 2014."

but at a high complexity cost for developers. A few recent systems such as Redis [red 2014] and Cassandra [cas 2014] have begun to provide general-purpose facilities for accessing data in DRAM, but their performance does not approach the full potential of DRAM-based storage.

This paper describes RAMCloud, a general-purpose storage system that keeps all data in DRAM at all times. RAMCloud combines three overall attributes: low latency, large scale, and durability. When used with leading-edge networking, RAMCloud offers exceptionally low latency for remote access. In our 80-node development cluster, a client can read any 100-byte object in less than 5 μs, and durable writes take about 15 μs. In a large datacenter with 100,000 nodes, we expect small reads to complete in less than 10 μs, which is 50-1000x faster than existing storage systems.

RAMCloud's second attribute is large scale. In order to support future Web applications, we designed RAMCloud to allow clusters to grow to at least 10,000 servers. RAMCloud aggregates all of their memories into a single coherent key-value store. This allows storage capacities of 1 PB or more.

The third attribute of RAMCloud is durability. Although RAMCloud keeps all data in DRAM, it also maintains backup copies of data on secondary storage to ensure a high level of durability and availability. This frees application developers from the need to manage a separate durable storage system, or to maintain consistency between in-memory and durable storage.

We hope that low-latency storage systems such as RAMCloud will stimulate the development of a new class of applications that manipulate large-scale datasets more intensively than is currently possible. Section 2 motivates RAMCloud by showing how the high latency of current storage systems limits large-scale applications, and it speculates about new applications that might be enabled by RAMCloud.

Sections 3–9 present the RAMCloud architecture from three different angles that address the issues of latency, scale, and durability:

Storage management. RAMCloud uses a unified log-structured approach for managing data both in memory and on secondary storage. This allows backup copies to be made efficiently, so that RAMCloud can provide the durability of replicated disk and the low latency of DRAM. The log-structured approach also simplifies crash recovery and utilizes DRAM twice as efficiently as traditional storage allocators such as malloc. RAMCloud uses a unique two-level approach to log cleaning, which maximizes DRAM space utilization while minimizing I/O bandwidth requirements for secondary storage.

Latency. RAMCloud avoids the overheads associated with kernel calls and interrupts by communicating directly with the NIC to send and receive packets, and by using a polling approach to wait for incoming packets. Our greatest challenge in achieving low latency has been finding a suitable threading architecture; our current implementation pays a significant latency penalty in order to provide an acceptable level of flexibility.

Crash recovery. RAMCloud takes advantage of the system's scale to recover quickly from server crashes. It does this by scattering backup data across the entire cluster and using hundreds of servers working concurrently to recover data from backups after crashes. Crash recovery is fast enough (typically 1-2 seconds) to provide a high degree of availability without keeping redundant copies of data online in DRAM.

We have implemented all of the features described in this paper in a working system, which we hope is of high enough quality to be used for real applications. The RAMCloud source code is freely available. This paper corresponds to RAMCloud 1.0 as of September 2014. Table I summarizes a few key performance measurements; these are discussed in more detail in the rest of the paper.

Table I. Selected performance metrics for RAMCloud.

Read latency (100-byte objects, one client, unloaded server)	4.7 µs
Read bandwidth (1 MB objects, one client, unloaded server)	2.7 GB/sec
Write latency (100-byte objects, one client, unloaded server)	15.0 µs
Write bandwidth (1 MB objects, one client, unloaded server)	430 MB/sec
Read throughput (100-byte objects, many clients, single server)	900 Kobjects/sec
Multi-read throughput (100-byte objects, many clients, one server)	6 Mobjects/s
Multi-write throughput (100-byte objects, many clients, one server)	450 Kobjects/s
Crash recovery throughput (per server, unloaded)	800 MB/s or 2.3 Mobjects/s
Crash recovery time (40 GB data, 80 servers)	1.9 s

A few themes appear repeatedly in our presentation of RAMCloud. The first theme is the use of randomization. In order for RAMCloud to be scalable, it must avoid centralized functionality wherever possible, and we have found randomization to be a powerful tool for creating simple yet effective distributed algorithms. The second theme is that we have attempted throughout the system to minimize the number of distinct error cases that must be handled, in order to reduce the complexity of fault tolerance. Section 6 will discuss how this often means handling errors at a very high level or a very low level. Third, the design of the system has been influenced in several ways by scaling in underlying technologies such as memory capacity and network speed. The impact of technology is particulary severe when technologies evolve at different rates. Section 2 discusses how uneven scaling motivated the creation of RAMCloud, and Section 10 describes how it also limits the system.

2. WHY DOES LOW LATENCY MATTER?

There are several motivations for RAMCloud [Ousterhout et al. 2011], but the most important one is to enable a new class of applications by creating a storage system with dramatically lower latency than existing systems. Figure 1 illustrates why storage latency is an important issue for large-scale Web applications. Before the rise of the Web, applications were typically run by loading the application code and all of its data into the memory of a single machine (see Figure 1(a)). This allows the application to access its data at main memory speeds (typically 50-100 ns); as a result, applications using this approach can perform intensive data manipulation while still providing interactive response to users. However, this approach limits application throughput to the capacity of a single machine.

The Web has led to the creation of new applications that support 1M–1B users; these applications cannot possibly use the single-machine approach of Figure 1(a). Instead, Web applications run on hundreds or thousands of servers in a datacenter, as shown in Figure 1(b). The servers are typically divided into two groups: one group services incoming HTTP requests from browsers, while the other group stores the application's data. Web applications typically use a *stateless* approach where the application servers do not retain data between browser requests: each request fetches the data that it needs from storage servers and discards that data once a response has been returned to the browser. The latency for each fetch varies from a few hundred microseconds to 10ms or more, depending on the network speed and whether the data is stored in memory, flash, or disk on the storage server.

Unfortunately, the environment for Web applications has not scaled uniformly compared to the single-machine environment. Total CPU power available to a Web application has improved by a factor of 1000x or more in comparison to single-server applications, and total storage capacity has also improved by a factor of 1000x or more,

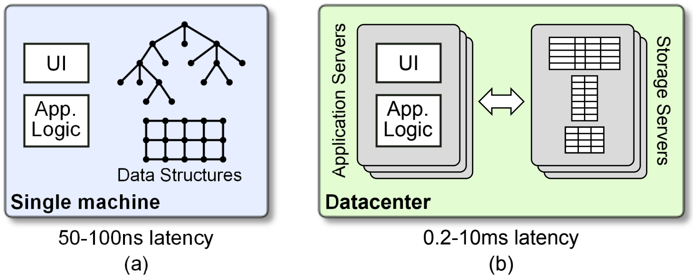


Fig. 1. In a traditional application (a) the application's data structures reside in memory on the same machine containing the application logic and user interface code; the latency for an application to access its data is determined by the last-level cache miss time (50-100 ns). In a scalable Web application (b) the data is stored on separate servers from the application logic and user interface code; the latency for an application to access data over the network ranges from 200-300 µs (if data is cached in the storage server's DRAM) to 10 ms or more (if data is on disk).

but the latency for an application to access its own data has *degraded* by 3-5 orders of magnitude. In addition, throughput has not scaled: if an application makes small random read requests, the total throughput of a few thousand storage servers in the configuration of Figure 1(b) is not much more than that of a single server in the configuration of Figure 1(a)! As a result, Web applications can serve large user communities, and they can store large amounts of data, but they cannot use very much data when processing a given browser request.

When we began the RAMCloud project in 2009, Facebook used a server structure similar to that in Figure 1(b) and it was experiencing the problems associated with high latency [Johnson and Rothschild 2009]. Facebook used MySQL database servers as the primary repository for its user data. However, these servers could not meet the needs of the application servers in terms of either latency or throughput, so they had been supplemented with memcached servers that cached recent query results in DRAM. By 2009, Facebook had approximately 4000 MySQL servers and 2000 memcached servers. The latency for memcached requests was around 300 µs, and the overall hit rate for data in memcached was about 96.5%.

Even so, the high latency of data access limited the functionality of Facebook applications and created complexity for developers. In order to provide acceptable response times for users, a Facebook application server could only make 100-150 sequential requests for data (either memcached or MySQL) while servicing a given browser request. Unfortunately, this limited the functionality that could be provided to users. To get past this limitation, Facebook applications made concurrent requests whenever possible. In addition, Facebook created materialized views that aggregated larger amounts of data in each memcached object, in the hopes of retrieving more useful data with each request. However, these optimizations added considerable complexity to application development. For example, the materialized views introduced consistency problems: it was difficult to identify all of the memcached objects to invalidate when data was changed in a MySQL server. Even with these optimizations, applications were still limited in the amount of data they can access.

There do exist scalable frameworks that can manipulate large amounts of data, such as MapReduce [Dean and Ghemawat 2008] and Spark [Zaharia et al. 2012]. However, these frameworks require data to be accessed in large sequential blocks in order to hide latency. As a result, these frameworks are typically used for batch jobs that run for minutes or hours; they are not suitable for online use in large-scale Web applications or for applications that require random access.

Our goal for RAMCloud is to achieve the lowest possible latency for small random accesses in large-scale applications; today this is around 5 µs for small clusters and 10 µs in a large datacenter. This represents an improvement of 50-1000x over typical storage systems used by Web applications today.

We hypothesize that low latencies will simplify the development of data-intensive applications like Facebook and enable a new class of applications that manipulate large data sets even more intensively. The new applications cannot exist today, since no existing storage system could meet their needs, so we can only speculate about their nature. We believe they will have two overall characteristics: (a) they will access large amounts of data in an irregular fashion (applications such as graph processing or large-scale machine learning could be candidates), and (b) they will operate at interactive timescales (tens to hundreds of milliseconds).

One possible application area for a system such as RAMCloud is collaboration at large scale. As a baseline, Facebook offers collaboration at small scale. It creates a "region of consciousness" for each user of a few dozen up to a few hundred friends: each user finds out instantly about status changes for any of his or her friends. In the future, applications may enable collaboration at a much larger scale. For example, consider the morning commute in a major metropolitan area in the year 2025. All of the cars will be self-driving, moving at high speed in tightly-packed caravans. In a single metropolitan area there may be a million or more cars on the road at once; each car's behavior will be affected by thousands of cars in its vicinity, and the region of consciousness for one car could include 50,000 or more other cars over the duration of a commute. A transportation system like this is likely to be controlled by a large-scale datacenter application, and the application is likely to need a storage system with extraordinarily low latency to disseminate large amounts of information in an irregular fashion among agents for the various cars.

3. RAMCLOUD ARCHITECTURE

In order to foster a new breed of data-intensive applications, RAMCloud implements a new class of storage that provides uniform low-latency access to very large datasets, and it ensures data durability so that developers do not have to manage a separate backing store. This section describes the overall architecture of the RAMCloud system, including the key-value data model offered to applications and the server-based organization of the system.

3.1. Data model

RAMCloud's data model is a key-value store, with a few extensions. We chose this data model because it is general-purpose enough to support a variety of applications, yet simple enough to yield a low latency implementation. We tried to avoid features that limit the system's scalability. For example, if RAMCloud were to assign a unique sequential key to each new object in a table, it would require all insertions for the table to pass through a single server; this feature is not scalable because the overall write throughput for the table could not be increased by adding servers. Thus, RAMCloud does not assign unique sequential keys.

Data in RAMCloud is divided into *tables*, each of which is identified by a unique textual name and a unique 64-bit identifier. A table contains any number of *objects*, each of which contains the following information:

— A variable-length *key*, up to 64 KB, which must be unique within its table. We initially used fixed-length 64-bit values for keys, but found that most applications need

TABLE OPERATIONS

createTable(*name*) → *id*

Creates a table with the given name (if it doesn't already exist) and returns the identifier for the table.

getTableId(*name*) → *id*

Returns the identifier for the table indicated by *name*, if such a table exists.

dropTable(*name*)

Deletes the table indicated by *name*, if it exists.

BASIC OPERATIONS

read(*tableId*, *key*) → *value*, *version*

Returns the value of the object given by *tableId* and *key*, along with its version number.

write(*tableId*, *key*, *value*) → *version*

Writes the object given by *tableId* and *key*, either creating a new object or replacing an existing object. The new object will have a higher version number than any previous object with the same *tableId* and *key*. Returns the new version number.

delete(*tableId*, *key*)

Deletes the object given by *tableId* and *key*, if it exists.

BULK OPERATIONS

multiRead([*tableId*, *key*]) → [*value*, *version*]

Returns values and version numbers for a collection of objects specified by table identifier and key.

multiWrite([*tableId*, *key*, *value*]) → [*version*]

Writes one or more objects and returns their new version numbers.

multiDelete([*tableId*, *key*])

Deletes one or more objects specified by table identifier and key.

enumerateTable(*tableId*) → [*key*, *value*, *version*]

Returns (in streaming fashion and unspecified order) all of the objects in the table given by *tableId*.

ATOMIC OPERATIONS

conditionalWrite(*tableId*, *key*, *value*, *condition*) → *version*

Writes the object given by *tableId* and *key*, but only if the object satisfies the constraints given by *condition*. The condition may require the object to have a particular version number, or it may require that the object not previously exist. Fails if the condition was not met, otherwise returns the object's new version.

increment(*tableId*, *key*, *amount*) → *value*, *version*

Treats the value of the object given by *tableId* and *key* as a number (either integer or floating-point) and atomically adds *amount* to that value. If the object doesn't already exist, sets it to *amount*. Returns the object's new value and version.

MANAGEMENT OPERATIONS

splitTablet(*tableId*, *keyHash*)

If *keyHash* is not already the smallest key hash in one of the tablets of the table given by *tableId*, splits the tablet containing *keyHash*, so that *keyHash* is now the smallest key hash in its tablet.

migrateTablet(*tableId*, *keyHash*, *newMaster*)

Move the tablet that contains *keyHash* in *tableId* so that it is now stored on the server given by *newMaster*.

Fig. 2. A summary of the API provided by RAMCloud 1.0. Some of these operations, such as **read** and **write**, map directly onto a single remote procedure call (RPC) from a client to a single server. Other operations, such as **multiRead** and **enumerateTable**, are implemented by the RAMCloud client library using multiple RPCs.

to look up some values using variable-length strings; in order to support these applications in the absence of secondary indexes, we switched to variable-length keys.
— A variable-length *value*, up to 1 MB.
— A 64-bit *version number*. When an object is written, RAMCloud guarantees that its new version number will be higher than any previous version number used for the same object (this property holds even if an object is deleted and then recreated).

An object is named uniquely by its key and the identifier for its table. RAMCloud does not assume any particular structure for either keys or values. Objects must be read and written in their entirety.

Figure 2 summarizes the most important operations provided by RAMCloud 1.0. They fall into the following categories:

— Operations for creating and deleting tables.
— Operations that read, write, and delete individual objects.
— Operations that manipulate objects in bulk, including multi-object forms of read, write, and delete, and an operation to iterate over all the objects in a table. These operations provide higher throughput by batching information about multiple objects in each server request and also by issuing requests to different servers in parallel.
— Two atomic operations, **conditionalWrite** and **increment**, which can be used to synchronize concurrent accesses to data. For example, a single object can be read and updated atomically by reading the object (which returns its current version number), computing a new value for the object, and invoking **conditionalWrite** to overwrite the object, with a condition specifying that the object must still have the same version returned by the read.
— Operations to split tablets and move them between masters (these operations are not typically used by normal clients).

We recognize that applications would benefit from higher level features such as secondary indexes and transactions, but decided to omit them from the initial implementation of RAMCloud. We are currently experimenting with these features to see if RAMCloud can support them without sacrificing its latency or scalability (see Section 10).

The consistency guarantees made (or not made) by a storage system can also have a large impact on the ease of developing applications. Many recent large-scale storage systems have accepted weaker consistency models in order to enhance scalability, but this has made them more difficult to program and sometimes exposes users of the applications to unexpected behaviors. For example, if a value is read shortly after being modified, a storage system with weaker consistency may return the old value. We have designed RAMCloud to provide strong consistency. Specifically, our goal for RAMCloud is linearizability [Herlihy and Wing 1990], which means that the system behaves as if each operation executes exactly once, atomically, at some point between when the client initiates the operation and when it receives a response. The architecture described in this paper contains much of the infrastructure needed for linearizability, but a few features are still missing (see Section 10.3).

3.2. Server architecture

RAMCloud is a software package that runs on a collection of commodity servers (see Figure 3). A RAMCloud cluster consists of a collection of *storage servers* managed by a single *coordinator*; client applications access RAMCloud data over a datacenter network using a thin library layer. We designed RAMCloud to support clusters as small as a few tens of servers, and as large as 10,000 or more servers.

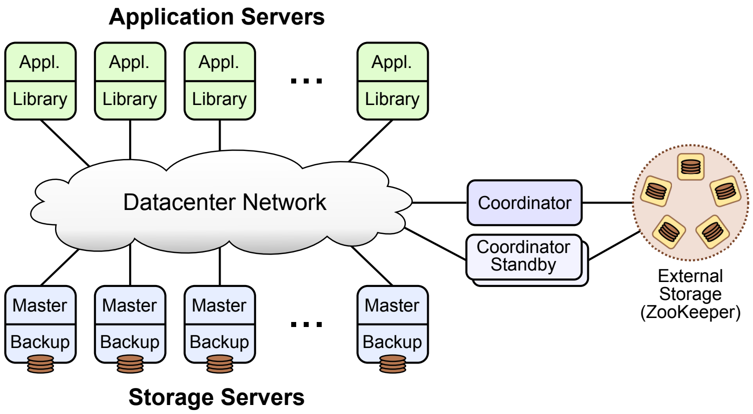


Fig. 3. The RAMCloud cluster architecture.

Each storage server contains two components. A *master* module manages the DRAM of the server to store RAMCloud data, and it handles read and write requests from clients. A *backup* module uses local disk or flash memory to store copies of data owned by masters on other servers. We expect storage servers to be configured with as much DRAM as is cost-effective, which is about 64-256 GB in 2014 (to store more data it is cheaper to use additional servers).

The information in tables is divided among masters in units of *tablets*. If a table is small, it consists of a single tablet and the entire table will be stored on one master. Large tables are divided into multiple tablets on different masters using hash partitioning: each key is hashed into a 64-bit value, and a single tablet contains the objects in one table whose key hashes fall in a given range. This approach tends to distribute the objects in a given table uniformly and randomly across its tablets.

The coordinator manages the cluster configuration, which consists primarily of metadata describing the current servers in the cluster, the current tables, and the assignment of tablets to servers. The coordinator is also responsible for managing recovery of crashed storage servers. At any given time there is a single active coordinator, but there may be multiple standby coordinators, each of which is prepared to take over if the active coordinator crashes. The active coordinator stores the cluster configuration information on an external storage system that is slower than RAMCloud but highly fault-tolerant (currently ZooKeeper [Hunt et al. 2010]). The standby coordinators use the external storage system to detect failures of the active coordinator, choose a new active coordinator, and recover the configuration information (this process is described in more detail in Section 9).

In order for a single coordinator to manage a large cluster without becoming a performance bottleneck, it must not be involved in high-frequency operations such as those that read and write RAMCloud objects. Each client library maintains a cache of configuration information, which allows it to identify the appropriate server for a read or write request without involving the coordinator. Clients only contact the coordinator to load the cache. If a client's cached configuration information becomes stale because data has moved, the client library discovers this when it makes a request to a server that no longer stores the desired information, at which point it flushes the stale data from its cache and fetches up-to-date information from the coordinator.

3.3. Networking substrate

In order for RAMCloud to achieve its latency goals, it requires a high-performance networking substrate with the following properties:

Low latency. Small packets can be delivered round-trip in less than 10 µs between arbitrary machines in a datacenter containing at least 100,000 servers.
High bandwidth. Each server has a network connection that runs at 10 Gb/s or higher.
Full bisection bandwidth. The network has sufficient bandwidth at all levels to support continuous transmission by all of the machines simultaneously without internal congestion of the network.

This kind of networking was not widely available in 2009 when we started the RAMCloud project, but it is becoming available today and we expect it to become commonplace in the future (see Section 5). In our development cluster we use Infiniband networking, which offers round-trip latency around 3.5 µs for a small cluster and bandwidth per machine of 24Gb/s.

4. LOG-STRUCTURED STORAGE

This section describes how RAMCloud uses DRAM and secondary storage to implement the key-value store (see [Rumble et al. 2014] and [Rumble 2014] for additional details not covered here). Three requirements drove the design of the storage mechanism. First, it must provide high performance, including low latency and high throughput. In particular, the latency of the operations in Figure 2 must not be limited by the speed of secondary storage. Second, the storage system must provide a high level of durability and availability, at least equivalent to replicated disks. Third, the storage system must be scalable, meaning that both overall system capacity and throughput can be increased by adding storage servers. In order to achieve scalability, servers must act independently as much as possible; any centralized functionality represents a potential scalability bottleneck.

RAMCloud provides durability and availability using a primary-backup approach to replication. It keeps a single (primary) copy of each object in DRAM, with multiple backup copies on secondary storage.

We considered keeping additional copies of data in DRAM, but this would be very expensive, since DRAM accounts for at least half of total system cost even without replication. In addition, replication in DRAM would not solve the durability problem, since all of the DRAM copies could be lost in a datacenter power outage. Replication in DRAM could improve throughput for frequently accessed objects, but it would require additional mechanisms to keep the replicas consistent, especially if writes can be processed at any replica. We expect that RAMCloud's throughput for a single copy will be high enough to make replication in DRAM unnecessary except in a small number of cases, and we leave it up to applications to handle these situations.

RAMCloud stores data using a log-structured approach that is similar in many ways to a log-structured file system (LFS) [Rosenblum and Ousterhout 1992]. Each master manages an append-only log in which it stores all of the objects in its assigned tablets. The log is the only storage for object data; a single log structure is used both for primary copies in memory and backup copies on secondary storage.

Log-structured storage provides four attractive properties, which have been instrumental in meeting the requirements of performance, durability, and scalability:

— **High throughput**: updates can be batched together in large blocks for efficient writing to secondary storage.

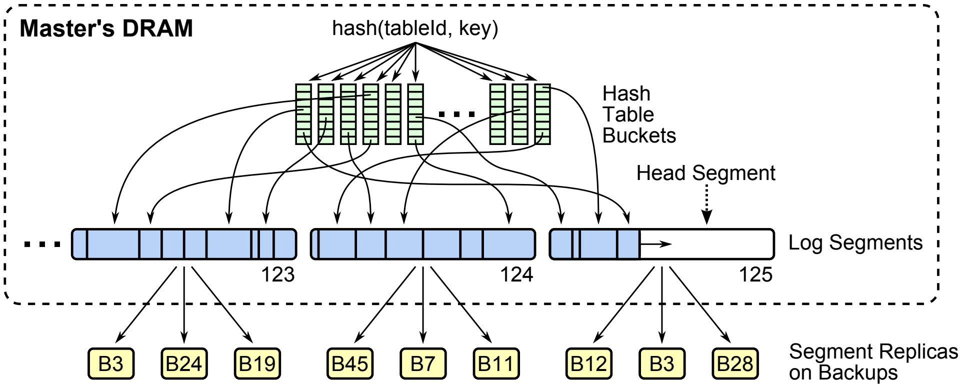


Fig. 4. Each master organizes its main memory as a log, which is divided into 8 MB segments. Each segment is replicated on the secondary storage of several backups (for example, segment 124 is replicated on backups 45, 7, and 11). The master maintains a hash table to locate live objects quickly. To look up an object, a master selects a hash table bucket using a hash of the object's table identifier and key. A bucket occupies one cache line (64 bytes) and contains 8 entries, each holding a pointer to an object in the log and 16 bits of the object's key hash. For each bucket entry that matches the desired key hash, the full key must be compared against the key stored in the log entry. Small objects can typically be retrieved with two last-level cache misses: one for the hash table bucket and one for the object in the log. If a hash bucket fills, its last entry is used as a pointer to an overflow bucket.

— **Crash recovery**: if a master crashes, its log can be replayed to reconstruct the information that was in the master's DRAM.
— **Efficient memory utilization**: the log serves as the storage allocator for most of a master's DRAM, and it does this more efficiently than a traditional malloc-style allocator or garbage collector.
— **Consistency**: the log provides a simple way of serializing operations. We have made only limited use of this feature so far, but expect it to become more important as we implement higher-level features such as multi-object transactions.

We will discuss these properties in more detail over the rest of the paper.

4.1. Log basics

The log for each master is divided into 8 MB *segments* as shown in Figure 4; log segments occupy almost all of the master's memory. New information is appended to the *head segment*; segments other than the head are immutable. Figure 5 summarizes the types of entries that are stored in the log.

In addition to the log, the only other major data structure on a master is a hash table, which contains one entry for each live object stored on the master. During read requests, the hash table allows the master to determine quickly whether there exists an object corresponding to a particular table identifier and key and, if so, find its entry in the log (see Figure 4).

Each log segment is replicated in secondary storage on a configurable number of backups (typically three). The master chooses a different set of backups at random for each segment; over time, its replicas tend to spread across all of the backups in the cluster. Segment replicas are never read during normal operation; they are only read if the master that wrote them crashes, at which time they are read in their entirety as described in Section 7. RAMCloud never makes random accesses to individual objects on secondary storage.

Object Describes a single object, including table identifier, key, value, version number, and coarse-grain timestamp for last modification (for cleaning). §4.2

Tombstone Indicates that an object has been deleted or overwritten. Contains the table identifier, key, and version number of the deleted object, as well as the identifier of the segment containing the object. §4.4

Segment header This is the first entry in each segment; it contains an identifier for the log's master and the identifier of this segment within the master's log. §4.5

Log digest Contains the identifers of all the segments that were part of the log when this entry was written. §4.3, §4.5, §7.4

Safe version Contains a version number larger than the version of any object ever managed by this master; ensures monotonicity of version numbers across deletes when a master's tablets are transferred to other masters during crash recovery.

Tablet statistics Compressed representation of the number of log entries and total log bytes consumed by each tablet stored on this master. §7.4

Fig. 5. The different types of entries stored in the RAMCloud log. Each entry also contains a checksum used to detect corruption. Log digests, safe versions, and table statistics are present only in segments containing newly written data, and they follow immediately after the segment header; they are not present in other segments, such as those generated by the cleaner or during recovery. The section numbers indicate where each entry type is discussed.

The segment size was chosen to make disk I/O efficient: with an 8 MB segment size, disk latency accounts for only about 10% of the time to read or write a full segment. Flash memory could support smaller segments efficiently, but RAMCloud requires each object to be stored in a single segment, so the segment size must be at least as large as the largest possible object (1 MB).

4.2. Durable writes

When a master receives a write request from a client, it appends a new entry for the object to its head log segment, creates a hash table entry for the object (or updates an existing entry), and then replicates the log entry synchronously in parallel to the backups storing the head segment. During replication, each backup appends the entry to a replica of the head segment buffered in its memory and responds to the master without waiting for I/O to secondary storage. When the master has received replies from all the backups, it responds to the client. The backups write the buffered segments to secondary storage asynchronously. The buffer space is freed once the segment has been closed (meaning a new head segment has been chosen and this segment is now immutable) and the buffer contents have been written to secondary storage.

This approach has two attractive properties: writes complete without waiting for I/O to secondary storage, and backups use secondary storage bandwidth efficiently by performing I/O in large blocks, even if objects are small.

However, the buffers create potential durability problems. RAMCloud promises clients that objects are durable at the time a write returns. In order to honor this promise, the data buffered in backups' main memories must survive power failures; otherwise a datacenter power failure could destory all copies of a newly written object. RAMCloud currently assumes that servers can continue operating for a short period after an impending power failure is detected, so that buffered data can be flushed to secondary storage. The amount of data buffered on each backup is small (not more than a few tens of megabytes), so only a few hundred millseconds are needed to write it safely to secondary storage. An alternative approach is for backups to store buffered head segments in nonvolatile memory that can survive power failures, such as flash-backed DIMM modules [nvd 2013].

4.3. Two-level cleaning

Over time, free space will accumulate in the logs as objects are deleted or overwritten. In order to reclaim the free space for new log segments, each RAMCloud master runs a *log cleaner*. The cleaner uses a mechanism similar to that of LFS [Rosenblum and Ousterhout 1992]:

— The cleaner runs when the number of free segments drops below a threshold value. In general it is better to delay cleaning until memory is low, since that will allow more free space to accumulate, which makes cleaning more efficient.
— In each pass, the cleaner selects several segments to clean, using the cost-benefit approach developed for LFS. The best segments to clean are those with large amounts of free space and those in which free space is accumulating slowly (i.e. the remaining objects are unlikely to be deleted soon). We found and corrected an error in the original LFS formula; see [Rumble et al. 2014] for details.
— For each of the selected segments, the cleaner scans the segment stored in memory and copies any live objects to new segments. Liveness is determined by checking for a reference to the object in the hash table. The live objects are sorted to separate old and new objects into different segments, which improves the efficiency of cleaning in the future.
— The cleaner makes the old segments' memory available for new segments, and it notifies the backups for those segments that they can reclaim storage for the replicas.

The cleaner does not write live data to the head segment, since this would require synchronization with normal write operations. Instead, the cleaner uses separate *survivor segments* for its output; once a cleaning pass has finished and the survivor segments have been replicated to backups, the cleaner adds the survivor segments to the log using the log digest (see Section 4.5). This approach allows the cleaner to run concurrently with normal writes, thereby hiding most of the cost of cleaning.

Cleaning introduces a tradeoff between memory utilization (the fraction of memory used for live data) and the cost of cleaning (CPU time and memory/network/disk bandwidth). As memory utilization increases, there will be less free space in segments, so the cleaner will spend more time copying live data and get back less free space. For example, if segments are cleaned when 80% of their data are still live, the cleaner must copy 8 bytes of live data for every 2 bytes it frees. At 90% utilization, the cleaner must copy 9 bytes of live data for every 1 byte freed. As memory utilization approaches 100%, the system will eventually run out of bandwidth for cleaning and write throughput will be limited by the rate at which the cleaner can create free segments. Techniques like LFS' cost-benefit segment selection improve cleaner performance by skewing the distribution of free space, so that segments chosen for cleaning have lower utilization than the overall average. However, they cannot eliminate the fundamental tradeoff between utilization and cleaning cost.

As described above, disk and memory cleaning are tied together: cleaning is first performed on segments in memory, then the results are reflected to backup copies on disk. This is the way that RAMCloud was initially implemented, but with this approach it was impossible to achieve both high memory utilization and high write throughput. If we used memory at high utilization (80-90%), write throughput would be limited by the cleaner's consumption of backup disk bandwidth. The only way to reduce disk bandwidth requirements was to allocate more space for the disk log, thereby reducing its utilization. For example, at 50% disk utilization we could achieve high write throughput. Furthermore, disks are cheap enough that the cost of the extra space is not significant. However, disk and memory were fundamentally tied together: in order to allocate more space for the disk log, we would also have had to allocate more space for the in-memory log, which would have reduced its utilization too. That was unacceptable.

The solution is to decouple the cleaning of the disk and memory logs so that the disk log is cleaned less frequently than the memory log — we call this *two-level cleaning*. The first level of cleaning, called *segment compaction*, operates only on the in-memory segments of masters and consumes no network or disk I/O. It compacts a single segment at a time, copying its live data into a smaller region of memory and freeing the original storage for new segments. Segment compaction maintains the same logical log in memory and on disk: each segment in memory still has a corresponding segment on disk. However, the segment in memory takes less space because defunct log entries have been removed. The second level of cleaning is just the mechanism described at the beginning of this subsection. We call this *combined cleaning* because it cleans both disk and memory together. Two-level cleaning introduces additional issues such as how to manage variable-size segments in DRAM and when to run each cleaner; see [Rumble et al. 2014] and [Rumble 2014] for details.

With two-level cleaning, memory can be cleaned without reflecting the updates on backups. As a result, memory can have higher utilization than disk. The cleaning cost for memory will be high, but DRAM has enough bandwidth to clean at 90% utilization or higher. Combined cleaning happens less often. The disk log becomes larger than the in-memory log, so it has lower overall utilization, and this reduces the bandwidth required for cleaning.

Two-level cleaning leverages the strengths of memory and disk to compensate for their weaknesses. For memory, space is precious but bandwidth for cleaning is plentiful, so RAMCloud uses extra bandwidth to enable higher utilization. For disk, space is cheap but bandwidth is precious, so RAMCloud uses extra space to save bandwidth. One disadvantage of two-level cleaning is that the larger on-disk log takes more time to read during crash recovery, but this overhead can be offset by using additional backups during crash recovery (see Section 7).

4.4. Tombstones

Whenever a master deletes or modifies an object, it appends a *tombstone* record to the log, which indicates that the previous version of the object is now defunct. Tombstones are ignored during normal operation, but they are needed during crash recovery to distinguish live objects from dead ones. Without tombstones, deleted objects would come back to life when a master's log is replayed during crash recovery.

Tombstones have proven to be a mixed blessing in RAMCloud: they prevent object resurrection, but they introduce problems of their own. One problem is tombstone garbage collection. Tombstones must eventually be removed from the log, but this is only safe if the corresponding objects have been cleaned (so they will never be seen during crash recovery). To enable tombstone deletion, each tombstone includes the identifier of the segment containing the obsolete object. When the cleaner encounters a tombstone in the log, it checks the segment referenced in the tombstone. If that segment is no longer part of the log, then it must have been cleaned, so the old object no longer exists and the tombstone can be deleted. If the segment is still in the log, then the tombstone must be preserved.

A second problem with tombstones is that they complicate two-level cleaning. Memory compaction removes objects from memory, but the objects remain on secondary storage. Thus, it is not safe to delete a tombstone, even during compaction, until the combined cleaner has deleted all of the replicas of segment containing the corresponding object (otherwise, the compacted segment for the tombstone could undergo combined cleaning and be reflected on disk before the object's segment has been deleted). If memory compaction runs for an extended period without combined cleaning, tombstones will accumulate in memory. For example, if a single object is overwritten repeatedly, memory compaction can eliminate all but one version of the object, but it must retain all of the tombstones. The accumulated tombstones make compaction less and less efficient; the result is that the combined cleaner must run more frequently than would be required without tombstones.

Given the issues with tombstones, we have wondered whether some other approach would provide a better mechanism for keeping track of object liveness. LFS used the metadata in its log to determine liveness: for example, a file block was still live if there was pointer to it in an inode. We considered approaches for RAMCloud that use explicit metadata to keep track of live objects, such as persisting the hash table into the log, but these were complex and created their own performance issues. Tombstones appear to be the best of the alternatives we have seen, but we consider this an area ripe for new ideas.

4.5. Log reconstruction

In order to make RAMCloud as scalable as possible, log management is completely decentralized: each master manages its own log independently, allocating new segments without contacting the coordinator. There is no central repository containing the locations of segment replicas; information is distributed among masters, which know the locations of their own replicas, and backups, which know the source for each replica they store.

When a master crashes, the coordinator must determine the locations of all of the master's segment replicas, for use in replaying the log. The coordinator does this by querying each of the backups in the cluster to find out which segments it stores for the crashed master. However, it is possible that some of the backups have also crashed, in which case they cannot respond to this query. The coordinator must be able to determine unambiguously whether the responses that it received constitute a complete and up-to-date copy of the crashed master's log.

RAMCloud takes two steps to ensure accurate log reconstruction. First, each new head segment includes a *log digest* entry, which lists the identifiers for all segments in the log at the time the digest was written. If the coordinator can find the latest digest, it can use it to ensure that all of the other segments of the log are available.

However, it is possible that all replicas for the latest head segment may be unavailable because of multiple crashes; the coordinator must be able to detect this situation and delay crash recovery until at least one copy of the head segment is available. To do this, RAMCloud enforces an ordering on log updates during segment transitions. When a master creates a new head segment, it tells each of the backups for that segment that the segment is *open*; when it has filled the segment and created a new head segment, it informs the backups for the old head that it is now *closed*. In the transition to a new head segment, a master must open the new segment and write a digest into it before closing the old head segment. Furthermore, the old head must be closed before any objects or tombstones are written to the new head segment. This guarantees two properties: (a) there is always at least one open segment for each master's log, and (b) if the coordinator finds an open segment, it can safely use that segment's digest to verify log completeness (if there are two open segments, the newer one must be empty, so it is safe to use the digest from either segment).

When the coordinator queries backups for replica information during crash recovery, backups return the log digest(s) for any open segment(s) to the coordinator. If at least one digest is returned, and if replicas are available for all of the segments named in that digest, then the coordinator knows that a complete copy of the log is available.

Table II. The hardware configuration of the 80-node cluster used for benchmarking. All nodes ran Linux 2.6.32 and were connected to a two-level Infiniband fabric with full bisection bandwidth. The Infiniband fabric supports 40 Gbps bandwidth, but PCI Express limited the nodes to about 24 Gbps.

CPU	Xeon X3470 (4x2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
Flash Disks	2 Crucial M4 SSDs CT128M4SSD2 (128 GB)
NIC	Mellanox ConnectX-2 Infiniband HCA
Switches	Mellanox MSX6036 (4X FDR) and Infiniscale IV (4X QDR)

4.6. Cleaner performance evaluation

The goal for the log cleaner is to provide high throughput for client writes, even at high memory utilization. Ideally, the cleaner will run concurrently with normal writes and create free space as quickly as it is needed, so that cleaning has no impact on write performance. However, as described in Section 4.3, the cost of cleaning rises non-linearly as memory utilization approaches 100%; at some utilization level the cleaner must inevitably limit write throughput.

We used the test cluster described in Table II to evaluate the effectiveness of the log cleaner, with the following overall results:

- RAMCloud can support memory utilizations up to 80-90% without significant impact on write throughput.
- The two-level approach to cleaning improves write throughput by as much as 6x, compared to the one-level approach.
- The log-structured approach to memory management allows significantly higher memory utilization than traditional memory allocators.

Figure 6 displays the results of a benchmark that measured the write throughput of a single master under an intensive workload of multi-write requests; the lines labeled “Two-level” show the performance of the cleaner configured for normal production use. We varied the workload in several ways to get a deeper understanding of cleaner performance:

Memory utilization. The percentage of the master’s log memory used for holding live data (not including tombstones) was fixed in each run, but varied from 30% to 90% in different runs. For example, at 50% utilization the master stored 8 GB of live data in 16 GB of total log space. As expected, throughput drops as utilization increases.

Object size. Figure 6 shows throughput with three different object sizes: 100, 1000, and 10000 bytes (we also ran experiments with 100KB objects; the results were nearly identical to those with 10KB objects). For small objects, cleaner performance is limited by per-object overheads such as updating the hash table. For large objects, cleaner performance is limited by the bandwidth available for writing replicas to flash disk. Workloads with small objects are more sensitive to memory utilization because tombstones are large compared to the objects they delete: at high utilization, the total memory utilization was significantly higher than the listed number due to an accumulation of tombstones.

Locality. We ran experiments with both uniform random overwrites of objects and a Zipfian distribution in which 90% of writes were made to 15% of the objects. The uniform random case represents a workload with no locality; Zipfian represents locality similar to what has been observed in memcached deployments [Atikoglu et al. 2012]. At high memory utilization, the cleaner operates more efficiently when

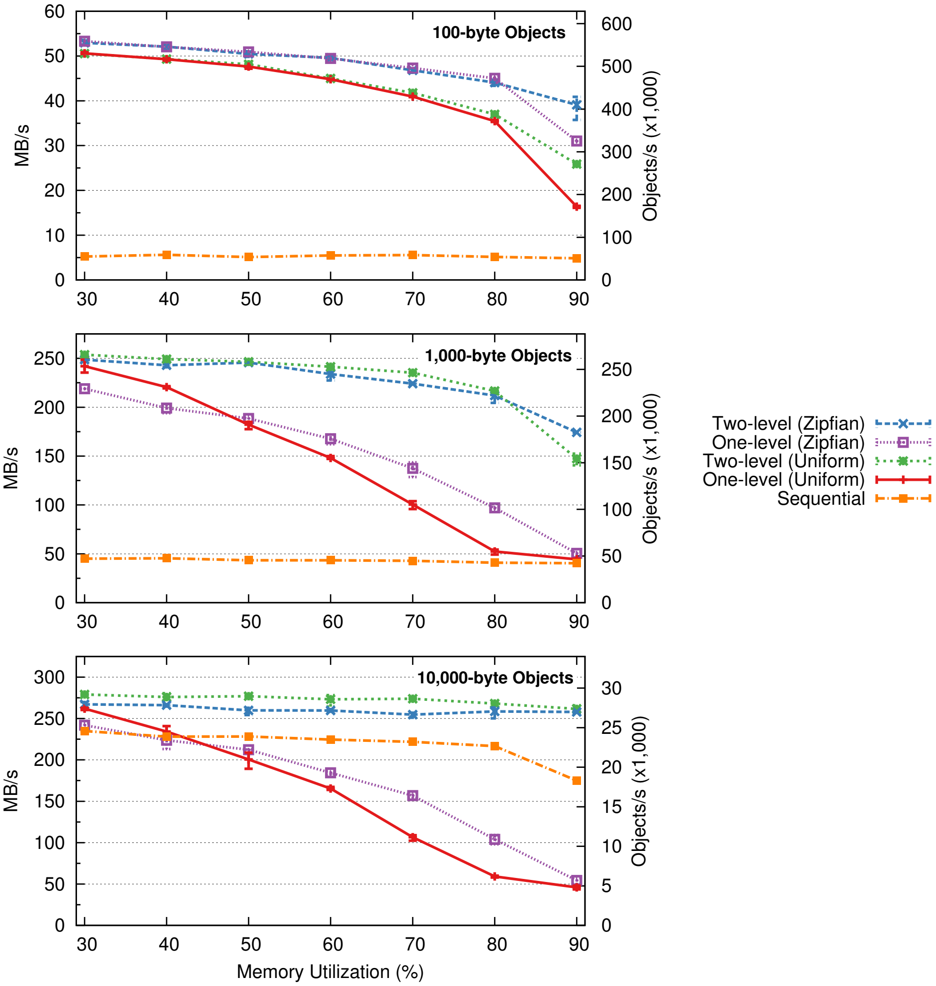


Fig. 6. Write throughput for a single master as a function of object size, memory utilization, and access locality. All curves except "Sequential" used concurrent multi-write requests to stress the cleaner to its limit: the client maintained 10 outstanding requests at any given time, with 75 individual writes in each request. The "Sequential" curve is similar to "Two-level (Uniform)" except that it used only a single outstanding write request at a time. In the "One-level" curves two-level cleaning was disabled, so only the combined cleaner was used. Each measurement used five nodes from the cluster described in Table II: one node ran a master, three nodes were used for backups (two backups with separate flash disks ran on each node); and one node ran the coordinator and the benchmark application. The master was given 16GB of log space and used two cores for cleaning. Each segment was replicated on three backups; in total, the backups provided 700 MB/s of write bandwidth for replication. The client created objects with sequential keys until the master reached a target memory utilization; then the client overwrote objects (maintaining a fixed amount of live data) until the overhead for cleaning converged to the stable value shown.

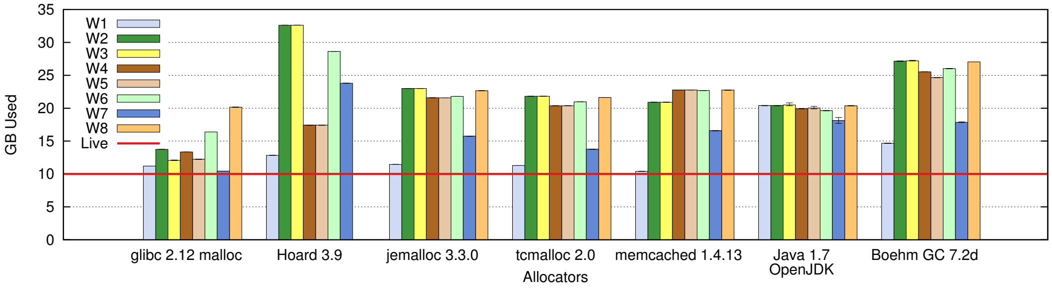


Fig. 7. Total memory needed by memory allocators under a collection of synthetic workloads (up is worse). Each workload maintained 10 GB of live data but changed the size distribution of its objects part-way through the experiment (see [Rumble et al. 2014] for details). "Live" indicates the amount of live data, and represents an optimal result. "glibc" is the allocator typically used by C and C++ applications on Linux. "Hoard" [Berger et al. 2000], "jemalloc" [Evans 2006], and "tcmalloc" [tcm 2013] are non-copying allocators designed for speed and multiprocessor scalability. "Memcached" is the slab-based allocator used in the memcached [mem 2011] object caching system. "Java" is the JVM's default parallel scavenging collector with no maximum heap size restriction (it ran out of memory if given less than 16 GB of total space). "Boehm GC" is a non-copying garbage collector for C and C++. Hoard could not complete the W8 workload (it overburdened the kernel by *mmap*ing each large allocation separately). Each data point is the average of 5 runs.

the workload has locality; this indicates that the cost-benefit selection mechanism is effective at separating hot and cold data.

The most important result from Figure 6 is that RAMCloud can support high memory utilization without sacrificing performance: write throughput degraded less than 20% at 80% memory utilization for all of the workloads except small objects with no locality. For large objects, even 90% memory utilization can be supported with low cleaning overhead.

Results in practice are likely to be even better than suggested by Figure 6. All of the measurements in Figure 6 except the curves labeled "Sequential" used the most intensive write workload we could generate (concurrent multi-writes), in order to create the greatest possible stress on the cleaner. However, actual workloads are likely to be less intensive than this. If a client issues individual write requests, the server will spend much of its time in basic request processing; as a result, objects will be written at a lower rate, and it will be easier for the cleaner to keep up. The "Sequential" curves in Figure 6 show performance under these conditions: if actual RAMCloud workloads are similar, it should be reasonable to run RAMCloud clusters at 90% memory utilization. For workloads with many bulk writes, it makes more sense to run at 80% utilization: the higher throughput will more than offset the 12.5% additional cost for memory.

Figure 6 also demonstrates the benefits of two-level cleaning. Each graph contains additional measurements in which segment compaction was disabled ("One-level"); in these experiments, the system used RAMCloud's original one-level approach where only the combined cleaner ran. The two-level cleaning approach provides a considerable performance improvement: at 90% utilization, client throughput is up to 6x higher with two-level cleaning than single-level cleaning. Two-level cleaning has the greatest benefit for large objects at high memory utilization: these workloads are limited by disk bandwidth, which the two-level approach optimizes.

4.7. Memory utilization

When we chose a log-structured approach for managing memory, rather than an off-the-shelf memory allocator such as the C library's malloc function, our original motivation was to improve throughput for writes. However, the logging approach has the additional advantage that it uses memory more efficiently than traditional storage allocators. As discussed in the previous section, RAMCloud can run efficiently at 80-90% memory utilization. For comparison, we measured a variety of traditional allocators under synthetic workloads and found that none of them can run safely above 50% memory utilization. The results are shown in Figure 7 and discussed in the rest of this section.

Memory allocators fall into two general classes: non-copying allocators and copying allocators. *Non-copying* allocators such as malloc cannot move an object once it has been allocated, so they are vulnerable to fragmentation. Non-copying allocators work well for individual applications with a consistent distribution of object sizes, but Figure 7 shows that they can easily waste half of memory when allocation patterns change.

Changes in allocation patterns may be rare in individual applications, but they are more likely in storage systems that serve many applications over a long period of time. Such shifts can be caused by changes in the set of applications using the system (adding new ones and/or removing old ones), by changes in application phases (switching from map to reduce), or by application upgrades that increase the size of common records (to include additional fields for new features). Non-copying allocators may work well in some cases, but they are unstable: a small application change could dramatically change the efficiency of the storage system. Unless excess memory is retained to handle the worst-case change, an application could suddenly find itself unable to make progress.

The second class of memory allocators consists of those that can move objects after they have been created, such as copying garbage collectors. In principle, garbage collectors can solve the fragmentation problem by moving live data to coalesce free space. However, this comes with a trade-off: at some point all of these collectors (even those that label themselves as "incremental") must walk all live data, relocate it, and update references. This is an expensive operation that scales poorly, so garbage collectors delay global collections until a large amount of garbage has accumulated. This negates any space savings gained by defragmenting memory.

RAMCloud's log cleaner is similar in many ways to a copying garbage collector, but it has the crucial advantage that it is completely incremental: it never needs to scan all of memory. This allows it to operate more efficiently than traditional garbage collectors. In order for purely incremental garbage collection to work, it must be possible to find the pointers to an object without scanning all of memory. RAMCloud has this property because pointers exist only in the hash table, where they can be located easily. Traditional storage allocators operate in harsher environments where the allocator has no control over pointers; the log-structured approach could not work in such environments.

For additional measurements of cleaner performance, see [Rumble et al. 2014] and [Rumble 2014].

5. ACHIEVING LOW LATENCY

We started the RAMCloud project with a goal of end-to-end latency less than 5 μs in a small cluster for simple remote procedure calls such as reading a small object; in a large datacenter, simple RPCs should complete in less than 10 μs. The greatest obstacle to achieving these goals is the installed base of networking infrastructure. When we started thinking about RAMCloud in 2009, typical RPC times in large datacenters were several hundred microseconds (see Table III). Most of this latency was due to the network switches: each switch added 10-30 μs delay, and packets typically traversed five switches in each direction. In addition, our goal for total round-trip time was also exceeded by operating system overheads (kernel calls, network stacks, inter-

Table III. The components of network latency for round-trip remote procedure calls in large datacenters. “Traversals” indicates the number of times a packet passes through each component in a round-trip (e.g., five network switches must be traversed in each direction for a three-level datacenter network). “2009” estimates total round-trip latency for each component in a typical large datacenter in 2009 using 1Gb Ethernet technology. “Possible 2014” estimates best-case latencies achievable at reasonable cost in 2014 using Infiniband or 10Gb Ethernet technology. “Limit” estimates the best latencies that can be achieved in the next 5-10 years, assuming new network architectures such as [Dally 2012] and a radical integration of the NIC with the CPU. All estimates assume no contention.

Component	**Traversals**	**2009**	**Possible 2014**	**Limit**
Network switches	10	100-300 µs	3-5 µs	0.2 µs
Operating system	4	40-60 µs	0 µs	0 µs
Network interface controller (NIC)	4	8-120 µs	2-4 µs	0.2 µs
Application/server software	3	1-2 µs	1-2 µs	1 µs
Propagation delay	2	1 µs	1 µs	1 µs
Total round-trip latency		**150-400µs**	**7-12 µs**	**2.4 µs**

rupt handlers, etc.) and even by the overhead for communication between the CPU and the network interface controller (NIC). Furthermore, most datacenter networks were oversubscribed by factors of 100x or more, so congestion caused by insufficient bandwidth of top-level links added as much as tens of milliseconds of additional latency during periods of high load.

Fortunately, there were signs in 2009 that networking infrastructure would improve in the future. Our performance goals were already achievable with Infiniband networking, and new 10Gb Ethernet switching chips offered the promise of both low latency and inexpensive bandwidth. We started the RAMCloud project with the assumption that low-latency networking infrastructure would become widely deployed within 5-10 years. Such infrastructure is available at reasonable cost today for 10Gb Ethernet as well as Infiniband, though it is not yet widely deployed. In the future, significant additional improvements are possible. With new architectures for network switches and for NIC-CPU integration, we estimate that round-trip times within large datacenters could be reduced to less than 3 µs over the next decade. In addition, custom switching chips will continue to drive down the cost of bandwidth, making oversubscription unnecessary and eliminating contention in the core of datacenter networks. Thus, we designed RAMCloud for the kind of high-speed networking we expect to be commonplace in the future, and we use Infiniband in our test cluster, which gives us those capabilities today.

Although most of the improvements in round-trip latency come from the networking infrastructure, it is still challenging to create a general-purpose storage system without adding significant overheads. Most of the latency budget for an RPC will be consumed by the network or by communication with the NIC; this leaves only about 1 µs for a RAMCloud server to process an RPC once it receives a request from the NIC. Satisfying this constraint was less about what we added to RAMCloud and mostly about what we had to leave out:

Kernel calls. Servers and applications must be able to send and receive packets without passing through the kernel.

Synchronization. Synchronization operations such as acquiring locks are quite expensive: even in the absence of cache misses or contention, acquiring and releasing a single spin-lock takes about 16ns, or almost 2% of the total budget for handling an RPC.

CPU cache misses. To meet our latency goal, a server cannot incur more than about ten last-level cache misses in the handling of a single RPC.

Batching. Most networked systems optimize for throughput, not latency. As a result, they group operations into batches, which amortize overheads such as kernel crossings or thread switches across many operations. However, batching requires some operations to be delayed until a full batch has been collected, and this is not acceptable in a low-latency system such as RAMCloud.

5.1. Kernel bypass and polling

RAMCloud depends heavily on two techniques for achieving low latency: *kernel bypass* and *polling*. Kernel bypass means that an application need not issue kernel calls to send and receive packets. Instead, NIC device registers are memory-mapped into the address space of the application, so the application can communicate directly with the NIC. Different applications use different sets of memory-mapped registers. Applications communicate packet buffer addresses to the NIC using virtual addresses, so the NIC must understand virtual-to-physical address mappings; typically this requires buffer memory to be pinned in physical memory. Kernel bypass requires special features in NICs, which are not yet universally available. Fortunately, these features are becoming more common over time (similar features are needed to support I/O virtualization in virtual machine monitors). Kernel bypass explains why operating system overheads drop to zero in Table III. The Infiniband NICs in our development cluster support kernel bypass.

Our second overall technique for low latency is to use polling (busy waiting) to wait for events. For example, when a client thread is waiting for a response to an RPC request, it does not sleep; instead, it repeatedly polls the NIC to check for the arrival of the response. Blocking the thread in this situation would serve little purpose: by the time the CPU could switch to another task, the RPC will probably have completed, and the polling approach eliminates the cost of taking an interrupt and waking the blocked thread (using a condition variable to wake a thread takes about 2 μs). RAMCloud servers also use a polling approach to wait for incoming requests: even when there are no requests for it to service, a server will consume one core for polling, so that it can respond quickly when a request arrives.

5.2. Transports

Low-level networking support in RAMCloud is implemented using a collection of *transport* classes. Each transport supports a different approach to network communication, but all of the transports implement a common API for higher-level software. The transport interface plays an important role in RAMCloud because it permits experimentation with a variety of networking technologies without any changes to software above the transport level. Each server is configured when it starts up with the transport(s) that it should support for communication. Different servers in the same cluster can use different transports.

RAMCloud 1.0 contains three built-in transports:

InfRcTransport. Uses Infiniband reliably connected queue pairs, which provide reliable in-order messages. InfRcTransport takes advantage of the kernel bypass features of Infiniband NICs. It is currently RAMCloud's fastest transport and is used in most of our performance measurements.

FastTransport. Given an underlying *driver* that can send and receive unreliable datagrams, FastTransport implements a custom protocol for reliable delivery. RAMCloud currently has drivers that use kernel bypass to send and receive UDP packets, Infiniband unreliable datagrams, and raw Ethernet packets, as well as a driver that uses the kernel to send and receive UDP packets. The name for this transport is unfortunate, since it is not yet as fast as InfRcTransport.

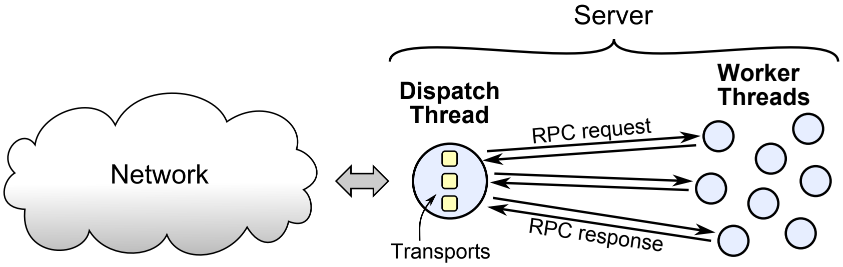


Fig. 8. The RAMCloud threading architecture. A single dispatch thread handles all network communication; it passes each incoming RPC request to a worker thread for handling. The response message is returned to the dispatch thread for transmission. Each server also contains additional threads for asynchronous tasks such as log cleaning.

TcpTransport. Uses standard TCP sockets implemented by the Linux kernel. TcpTransport does not use kernel bypass, so it has about 100 µs higher latency than InfRcTransport.

These transports range in size from about 1000 lines of C++ code for TcpTransport up to about 3000 lines of code for FastTransport. The transport API provides reliable delivery of variable-length request and response messages for remote procedure calls. The request-response nature of RPCs is reflected in the transport API; this enables internal optimizations in the transports, such as using an RPC response as the acknowledgment for the request.

5.3. Thread structure

The threading architecture used for a server has a significant impact on both latency and throughput. The best way to optimize latency is to use a single thread for handling all requests. This approach eliminates synchronization between threads, and it also eliminates cache misses required to move data between cores in a multi-threaded environment. However, the single-threaded approach limits server throughput; multi-threaded servers can handle many requests in parallel, albeit with additional overheads for synchronization and cache coherency.

Since latency was more important to us than throughput, we initially implemented RAMCloud with a single thread per server to handle all incoming RPCs: it executed in a loop, polling for an incoming request and then servicing it. However, we could not find a satisfactory way to implement fault tolerance with this approach. If an RPC takes a long time to process, the caller attempts to ping the server to make sure it is still alive. With the single-threaded approach, there was no thread to process incoming ping requests while an RPC was being processed, so long-running RPCs resulted in timeouts. Furthermore, if one machine crashed, any server communicating with it would experience long delays waiting for its RPCs to time out, during which time it could not process ping requests either. As a result, any server crash resulted in cascading timeouts that took down most or all of the cluster. We considered requiring long-running operations to check occasionally for incoming ping RPCs, but this seemed complex and error-prone.

Because of this problem, we eventually switched to a multi-threaded approach for RAMCloud servers. RPCs are handled by a single *dispatch thread* and a collection of *worker threads* as shown in Figure 8. The dispatch thread handles all network communication, including incoming requests and outgoing responses. When a complete RPC message has been received by the dispatch thread, it selects a worker thread and hands off the request for processing. The worker thread handles the request, generates a response message, and then returns the response to the dispatch thread for transmission. Transport code (including communication with the NIC) executes only in the dispatch thread, so no internal synchronization is needed for transports.

The dispatch thread implements functionality roughly equivalent to the interrupt handlers of an operating system, except that it is driven by synchronous polling rather than asynchronous interrupts. It is organized around a dispatcher that continuously polls for events and then handles them. Transports define *pollers*, which are invoked in each pass through the dispatcher's inner polling loop. For example, InfRcTransport creates a poller that checks the Infiniband NIC for incoming packets and for the return of transmit buffers. The dispatcher also allows the creation of *timers*, which will be invoked by the polling loop at specific future times, and *file handlers*, which are invoked when kernel-implemented files such as sockets become readable or writable. Timers are used by transports to trigger retransmissions and timeouts, and file handlers are used by transports such as TcpTransport that send and receive messages via the kernel. Pollers, timers, and file handlers must complete quickly without blocking, so that they do not delay the processing of other events.

Communication between the dispatch thread and worker threads is also handled by polling in order to minimize latency. When a worker thread finishes handling an RPC and becomes idle, it continously polls a private control block associated with the thread. When an RPC request becomes available, the dispatch thread selects an idle worker thread and stores a pointer to the RPC in the thread's control block; this assigns the RPC to the worker for processing. Upon completion of the RPC, the worker thread stores a pointer to its result back in the control block; a poller in the dispatch thread notices the result message and calls the appropriate transport to transmit the result.

If a worker thread polls for a long time (currently 10ms) without receiving a new RPC request, then it blocks; the dispatch thread will use a slower mechanism (a Linux futex) to wake up the worker the next time it assigns an RPC to it. The dispatch thread assigns RPCs to polling workers instead of blocked ones whenever possible; as a result, the number of polling worker threads automatically adjusts to the server's load. During long idle periods all of the worker threads will block, leaving only the dispatch thread consuming CPU time.

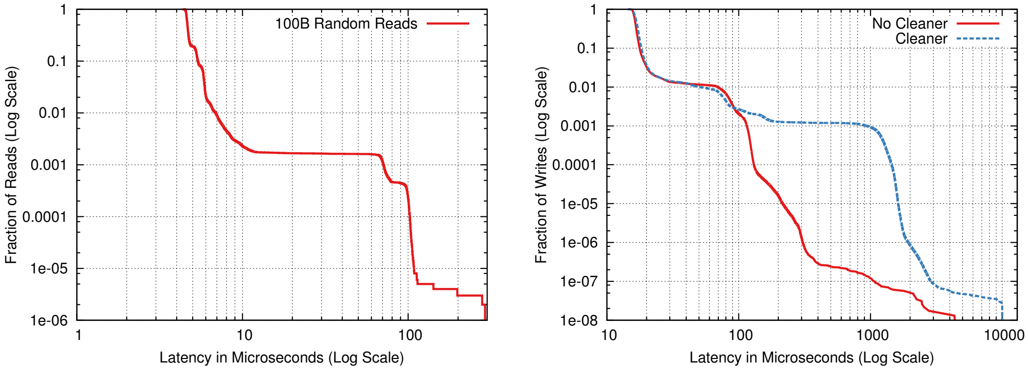
The multi-threaded approach allows multiple requests to be serviced simultaneously. This improves throughput in general, and also allows ping requests to be handled while a long-running RPC is in process.

The dispatch thread implements a reservation system, based on the opcodes of RPCs, that limits the number of threads that can be working simultaneously on any given class of RPCs. This ensures that there will always be a worker thread available to handle short-running RPCs such as ping requests. It also prevents distributed deadlocks: for example, without the reservation system, all of the threads in a group of servers could be assigned to process incoming write requests, leaving no threads to process replication requests that occur as part of the writes.

Unfortunately, the multi-threaded approach requires two thread handoffs for each request; together they add about 410 ns to the latency for simple reads in comparison to a single-threaded approach. The cost of a thread handoff takes two forms. The first is the direct cost of transferring a message pointer from one thread to another; this takes about 100 ns in each direction. In addition, there are several data structures that are accessed by both the dispatch thread and a worker thread, such as the request and response messages; thread handoffs result in extra cache misses to transfer these structures from core to core. Given the total time budget of 1 μs for a server to process a request, the overhead for thread switches is a significant issue; we are continuing to look for alternative architectures with lower overhead.

Table IV. Latency in microseconds to read or write objects of a given size, chosen at random from a large table. All writes were overwrites of existing objects (creating new objects is slightly faster), and all experiments used 30-byte keys. 99% means 99th percentile. Latency was measured end-to-end at the client, using an unloaded server.

	Reads				**Writes**			
Object Size	**Median**	**90%**	**99%**	**99.9%**	**Median**	**90%**	**99%**	**99.9%**
100 B	4.7	5.4	7.2	72.2	15.0	16.2	41.5	154
1000 B	7.0	7.7	9.4	72.5	19.4	20.8	105	176
10000 B	10.1	11.1	13.8	79.1	35.3	37.7	209	287
100000 B	42.8	44.0	45.3	109	228	311	426	489
1000000 B	358	364	381	454	2200	2300	2400	2700



(a) (b)

Fig. 9. Tail latency distributions for reads (a) and overwrites (b) when a single client issues back-to-back requests for 100-byte objects chosen at random using a uniform distribution. Each y-coordinate is the fraction of accesses that took longer than the corresponding x-coordinate. In (b) the "No cleaner" curve was measured with cleaning disabled; the "Cleaner" curve was measured at 90% memory utilization with cleaning enabled. The median write latency was 0.35 µs higher with cleaning enabled.

5.4. Latency analysis

This section presents the results of several experiments that measured the latency of basic RAMCloud read and write operations, as well as the throughput of individual RAMCloud servers. The key results are:

- The latency for simple reads and writes is dominated by the network hardware and by unavoidable cache misses on the server.
- The most significant overhead in the RAMCloud software comes from the handoffs between dispatch and worker threads; these account for about 10% of the latency for reads and about 20-30% of the latency for writes.
- The log cleaner has very little impact on the latency of writes, even at high memory utilization.
- For small reads, the dispatch thread is the performance bottleneck.

All of the experiments in this section used the cluster hardware described in Table II, with a replication factor of three for writes.

5.4.1. Basic latency for reads and writes. Table IV shows the end-to-end latency for a client to read or write randomly-chosen objects of varying size using an unloaded server. The median time to read a 100B object is 4.7 µs, and the median time to write a 100B object is 15.0 µs. As the object size increases, the write latency increases faster than the read latency; this is because the server must retransmit the object three times for replication.

5.4.2. Tail latency. Figure 9 graphs the tail latency in detail for reads and writes of 100-byte objects. About 99.8% of all 100-byte reads take less than 10 μs, and about 98% of all 100-byte writes take less than 20 μs. The most significant factor in tail latency is an additional delay of about 70 μs, which affects about 2 in 1000 read requests and 1 in 100 write requests. We have not yet isolated the cause of this delay, but it appears to occur during the handoff from the dispatch thread to a worker thread.

5.4.3. How much does the cleaner impact write latency?. Figure 9(b) shows write latency both in normal operation with the cleaner running, and also in a special setup where the cleaner was disabled. The cleaner increased the median latency for writes by only 2%, and the latency distributions with and without cleaning are similar up to about the 99.9th percentile. About 0.1% of write requests suffer an additional 1ms or greater delay when the cleaner is running. These delays appear to be caused by head-of-line blocking behind large cleaner replication requests, both in the master's NIC and on the backups.

5.4.4. Where does the time go?. Figure 10 shows a detailed timeline for a read of a small object chosen at random from a large table. Three factors account for almost all of the latency:

— **Network**: 3.2 μs out of the 4.8 μs total time was spent in the Infiniband network or communicating between CPU and NIC.
— **Cache misses**: There were a total of 9 L3 cache misses on the server for each read request; Figure 10 displays the reason for each. A normal L3 cache miss takes 86 ns, but RAMCloud issues prefetch instructions for network packets and log entries, and this reduces the cost for several of the misses.
— **Thread handoffs**: The timeline shows about 220 ns in direct costs due to thread handoffs between the dispatch and worker threads. However, the handoffs also resulted in 24 additional L2 caches misses, for a total cost of about 410 ns.

The total time spent on the server is about 1.2 μs, excluding NIC communication time, and most of this time is accounted for by cache misses and thread handoffs.

Figure 11 shows a detailed timeline for a write request that overwrites a small object chosen at random. Most of the total time for the RPC was spent replicating the new data to three backups (8.5 μs out of a total of 14.7 μs). The replication RPCs incurred high overheads on the master (about 1 μs to send each RPC and another 1 μs to process the response); some of this is due to NIC interactions, but at least half is because of inefficient interactions between the worker thread and the dispatch thread (sending RPCs and receiving the results involves transport code running in the dispatch thread).

As can be seen from Figures 10 and 11, much of the latency for reads and writes comes from the networking system (including NICs) and from cache misses. The most significant cost attributable to RAMCloud code comes from the interactions between the dispatch and worker threads: these account for about 10% of the total latency for reads and 20-30% of the latency for writes. The higher overhead for writes is due to a particularly inefficient mechanism for communicating with the dispatch thread to issue replication RPCs; we plan to refactor this mechanism.

5.4.5. How is latency impacted by server load?. The latency measurements up until now have used an unloaded server; Figure 12 shows how latency degrades if the server is loaded. This experiment used Workoad A of the YCSB benchmark [Cooper et al. 2010] to generate different loads on a single server, and it measured the latency for reads

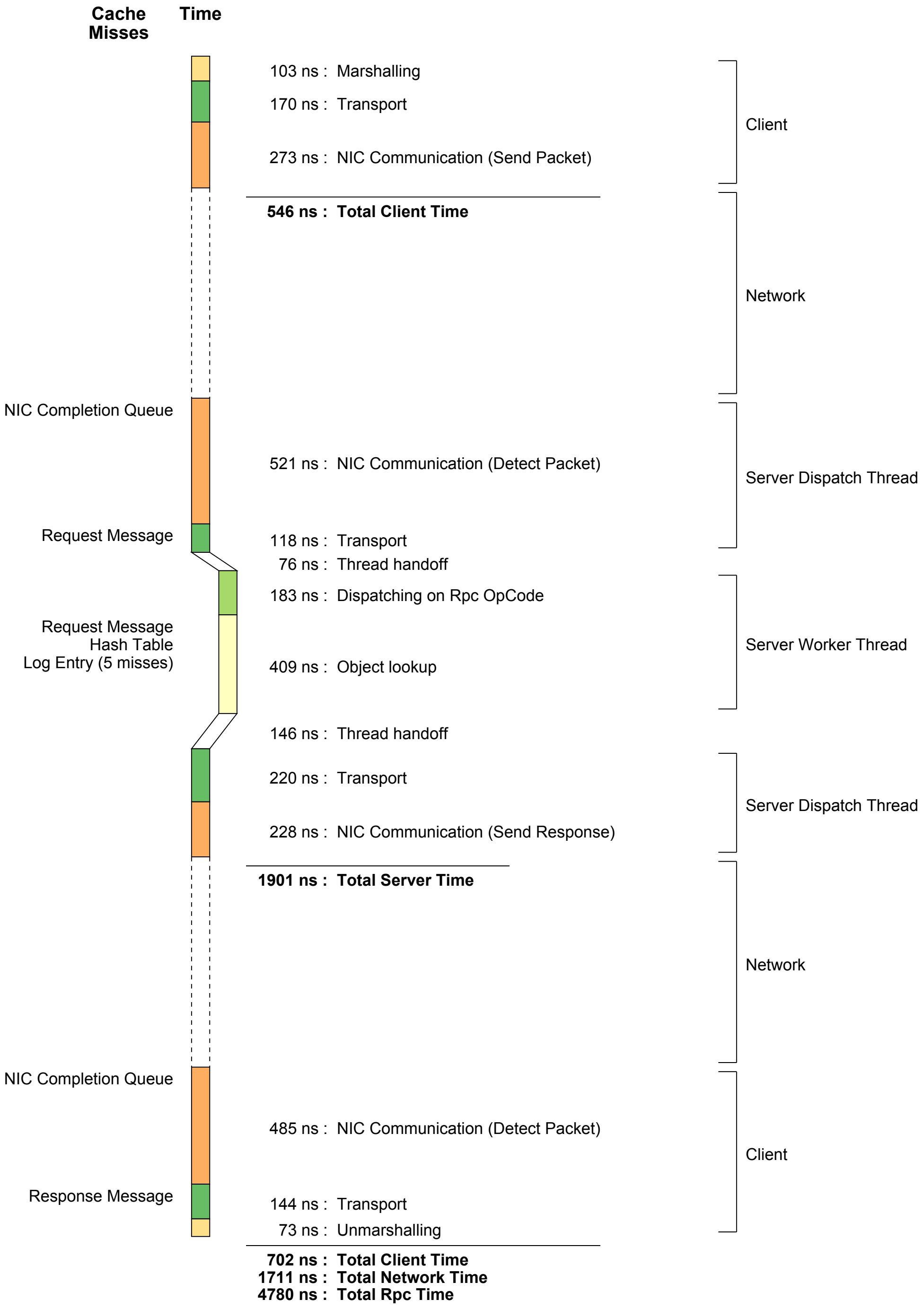


Fig. 10. Timeline to read a 100-byte object with 30-byte key chosen at random from a large table. The diagonal lines represent thread handoffs between the dispatch thread and the worker thread. There were a total of 9 cache misses on the server and 3 on the client; the text in the left column identifies the cause and approximate time of occurrence for each miss. The total network time (including both request and response) was 1711 ns (the experiment could not measure request and response times separately).

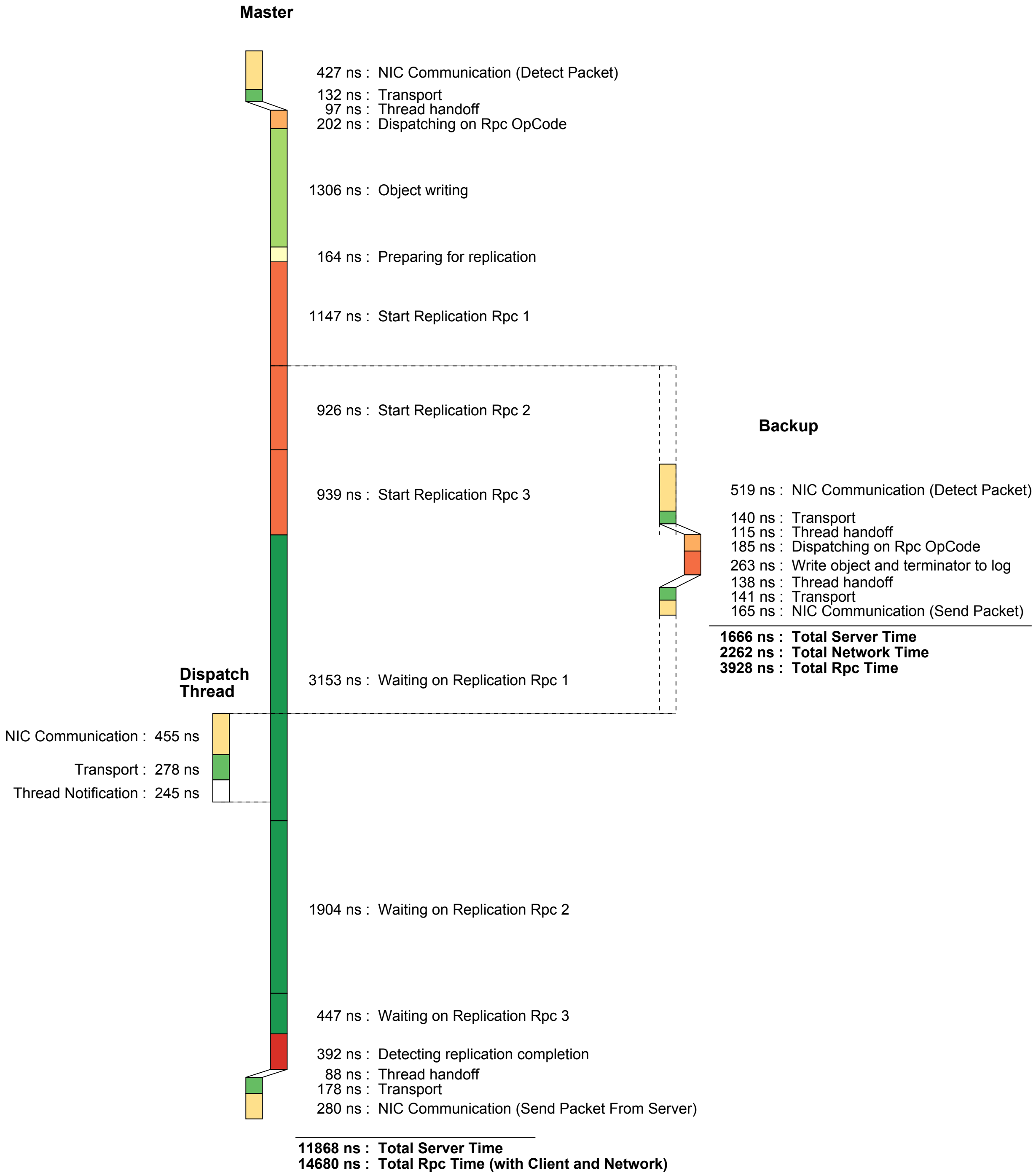
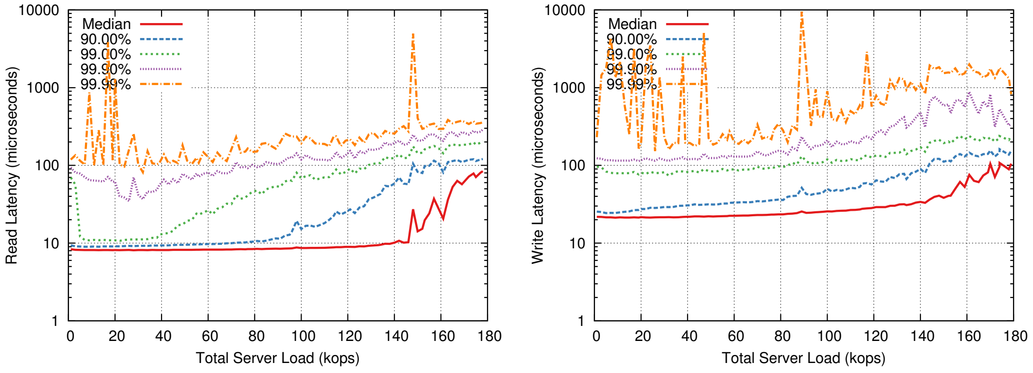

Master
427 ns : NIC Communication (Detect Packet)
132 ns : Transport
97 ns : Thread handoff
202 ns : Dispatching on Rpc OpCode
1306 ns : Object writing
164 ns : Preparing for replication
1147 ns : Start Replication Rpc 1
926 ns : Start Replication Rpc 2
939 ns : Start Replication Rpc 3
3153 ns : Waiting on Replication Rpc 1
1904 ns : Waiting on Replication Rpc 2
447 ns : Waiting on Replication Rpc 3
392 ns : Detecting replication completion
88 ns : Thread handoff
178 ns : Transport
280 ns : NIC Communication (Send Packet From Server)
11868 ns : Total Server Time
14680 ns : Total Rpc Time (with Client and Network)
Backup
519 ns : NIC Communication (Detect Packet)
140 ns : Transport
115 ns : Thread handoff
185 ns : Dispatching on Rpc OpCode
263 ns : Write object and terminator to log
138 ns : Thread handoff
141 ns : Transport
165 ns : NIC Communication (Send Packet)
1666 ns : Total Server Time
2262 ns : Total Network Time
3928 ns : Total Rpc Time
Dispatch Thread
NIC Communication : 455 ns
Transport : 278 ns
Thread Notification : 245 ns

Fig. 11. Timeline to write a 100B object with 30B key chosen at random from a large table, with a replication factor of three. The figure shows only time on the master and one of the three backups (client and network times are omitted to save space; they are similar to the times in Figure 10). The small timeline on the left shows the dispatch thread receiving the result from the first replication RPC.



(a) (b)

Fig. 12. The impact of server load on latency for reads (a) and writes (b). The workload was generated using a C++ implementation of the YCSB workload A [Cooper et al. 2010] running on 20 clients with a single server. Each client read and wrote 1000-byte objects using a Zipfian distribution for locality, with 50% reads and 50% writes. The request rate from each client was varied to produce different server loads; read and write latency were measured on one of the clients. Each graph displays the median latency at each workload plus several tail latencies ("99%" refers to 99th-percentile latency). The maximum load that the server could sustain was about 180 kops/sec.

and writes on one of the YCSB clients. The median latency for reads and writes did not increase significantly until the server was loaded at 70-80% of its capacity. However, the tail latency was more sensitive to load, particularly for reads: 90th-percentile read latency begins to increase when the server is about 40% loaded, and 99th-percentile read latency increases once the server reaches a load of about 20% of its capacity.

5.4.6. What is the throughput of a single server?. The final experiment in this section measures total server throughput for read requests when multiple clients access small objects. Figure 13 shows the results. If clients issue individual read requests (Figure 13(a)), a single server can handle about 900,000 requests per second. If clients use multi-read requests to fetch objects in large batches (Figure 13(b)), a single server can return about 6 million objects per second.

Figure 13 also shows the utilization of worker threads during the experiments. For individual reads, the maximum worker utilization is only about 0.8 (the server cannot keep a single worker thread completely busy); this indicates that the dispatch thread is the bottleneck for throughput. Most of the dispatch thread's time is spent communicating with the NIC and interacting with worker threads. We have not optimized RAMCloud for throughput, so the dispatch thread currently performs these operations independently for each RPC; batching techniques could be used to make the dispatch thread more efficient under high load. When clients issue multi-read operations (Figure 13(b)), it takes longer for workers to process each request, so the dispatch thread can keep several workers busy.

6. FAULT TOLERANCE INTRODUCTION

Fault tolerance has proven to be the most complex and difficult part of the RAMCloud implementation; we have spent considerably more effort on it than on achieving low latency. RAMCloud must recover from many different kinds of failures:

— Low-level networking failures, such as packet loss.
— Crashes of individual masters and backups.
— Coordinator crashes.
— Corruption of segments, either in DRAM or on secondary storage.

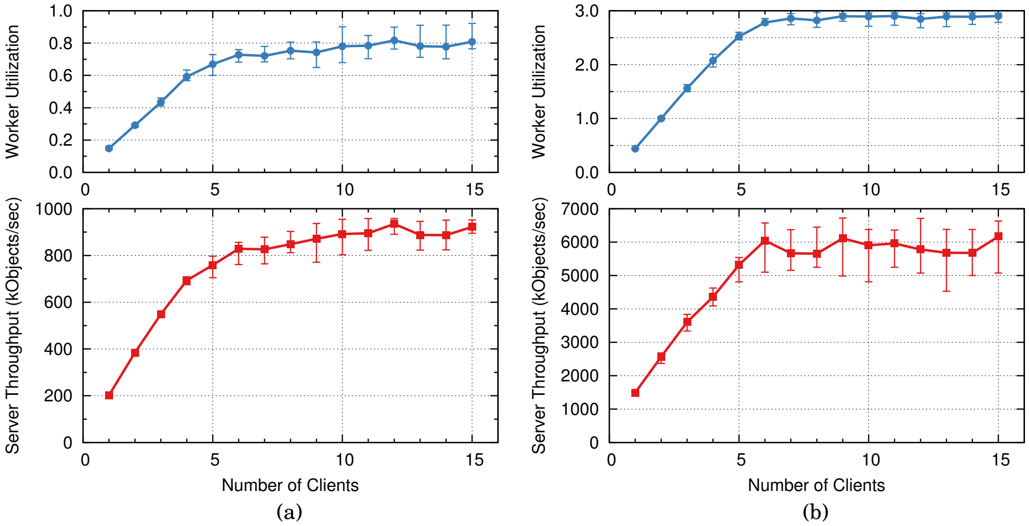


Fig. 13. Throughput of a single server for reads of small objects. Each client generated a continuous stream of back-to-back requests containing either a single read request (a) or a multi-read request for 70 objects (b). All objects had 30-byte keys and 100-byte values, and were chosen at random from a single table with 2M objects. The top graphs show the resources consumed by worker threads (for example, a utilization of 1.5 means that, on average, 1.5 worker threads were occupied servicing requests over the measurement interval). The number of concurrent worker threads was limited to 3 in this experiment (servers had 4 cores). Each data point displays the average, minimum, and maximum values over five one-second runs.

Multiple failures can occur simultaneously; for example, all of the backups storing replicas for a particular segment might crash at the same time, or a datacenter power failure could take down the entire cluster. In addition, RAMCloud may decide that a server has crashed when it is merely slow or disconnected, and the server could continue operating after the system has reassigned its responsibilities; RAMCloud must neutralize these *zombie* servers so that they don't return stale data or produce other undesirable behaviors.

Our overall goal for RAMCloud fault tolerance is for the system to deliver normal service even in the presence of individual server failures. This means that the system should provide near-continuous availability, high performance, and correct operation with no loss of data. RAMCloud should also provide normal service in the face of multiple failures, as long as the failures are randomly distributed and small in number compared to the cluster size. If a large-scale outage occurs, such as a network partition or a power failure, the system may become partially or completely unavailable until the problem has been corrected and servers have restarted. No data should ever be lost unless all of the replicas of a particular segment are destroyed; we expect the replication factor to be chosen in a way that makes this extremely unlikely.

We assume a fail-stop model for failures, in which the only way servers fail is by crashing. If a server has not crashed, then we assume it is functioning correctly. We have not attempted to handle Byzantine failures, in which servers deliberately misbehave. When a server crashes and restarts, we assume that data on its secondary storage will survive the crash with high probability. We assume an asynchronous network in which packets may be lost, delayed, duplicated, or reordered. Communication with a host may be disrupted temporarily, such that the host appears to have crashed, and then resume, without the host actually crashing. We expect network partitions inside a datacenter to be rare, so RAMCloud assumes full network connectivity among all of the servers in the cluster. If a network partition occurs, only servers in the partition containing the current coordinator will continue to provide service.

Error handling is a significant source of complexity in large-scale systems like RAMCloud. Furthermore, it is difficult to test and rarely exercised, so it may not work when needed. Because of these problems, we designed RAMCloud to minimize the visibility of failures, both in terms of the number of different failure conditions that must be handled and the number of places where they must be handled. We used two specific techniques: *masking* and *failure promotion*. Masking means that error recovery is implemented at a low level so that higher levels of software need not be aware of the problems. For example, we used masking in the RAMCloud client library. All internal RAMCloud failures, such as server crashes, are handled internally by the client library. No failure conditions are returned by any of the client library methods; in the worst case, the methods will delay until cluster functionality has been restored and the operation can complete.

We used the second technique, failure promotion, to handle failures within the storage servers. If a server detects an internal error such as a data structure inconsistency, it does not usually attempt to handle that problem in a problem-specific fashion. Instead, in most cases it "promotes" the error to a server crash by logging a message and exiting. Thus, instead of writing many different error handlers for each of the individual problems, we only had to write handlers for server crashes, which were unavoidable. For example, if a master detects corruption in an object in memory, it could potentially restore the object by reading one of the backup replicas. However, this special case would have added complexity (there is currently no backup operation to read an object from a replica), so we chose instead to crash the server and invoke normal master recovery code. In addition to reducing the complexity of failure handling code, failure promotion also has the advantage that the remaining fault handlers are invoked more frequently, so bugs are more likely to be detected and fixed.

Promoting a failure will usually increase the cost of handling it, compared to a more specialized handler. Thus, failure promotion works best for failures that are infrequent. If a particular failure happens frequently, it may be necessary to handle it in a more specialized fashion. For example, RAMCloud does not use failure promotion for network communication problems; these failures are handled, and masked, in the remote procedure call system.

RAMCloud uses promotion and masking together when servers communicate with each other. When a master issues a remote procedure call (RPC) to a backup, network communication problems are masked by the RPC system, which will retry after most errors. The only failure returned to the RPC's caller is a crash of the backup, and this manifests itself in exactly one way: "target server not up." This error is returned not only if the backup crashed during the RPC, but also if it crashed before the RPC was initiated, if the specified server exited normally, or if it was never in the cluster in the first place. The process of detecting and handling the crash involves several intermediate stages, but these are not visible to the RPC's caller; either the RPC completes or the caller knows that the target will never participate in the cluster again. This turns out to be just the right amount of information needed in most situations, and it minimizes the amount of failure handling code. Once the coordinator decides that a server has crashed, the server may not rejoin the cluster; even if the server didn't actually crash, its only alternative is to restart with a new identity. This eliminated the need to write code for the "rejoin" case.

With the use of failure promotion, fault tolerance in RAMCloud consists of three primary cases, corresponding to the major server roles: master crashes, backup crashes, and coordinator crashes. When a master crashes, all of the information in its DRAM is lost and must be reconstructed from backup replicas. When a backup crashes, its

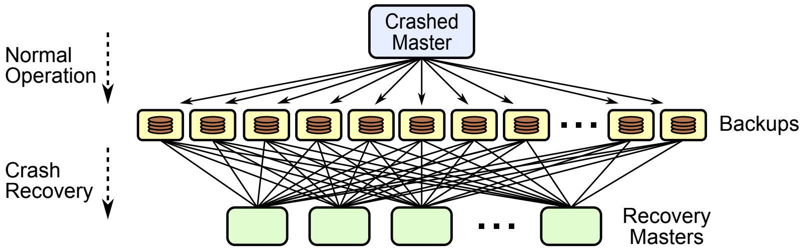


Fig. 14. Data flow for crash recovery. During normal operation each master scatters its backup replicas evenly across all of the backups in the cluster. During crash recovery, the backups retrieve this data and send it to a collection of recovery masters, which replay log entries to incorporate the crashed master's objects into their own logs. Each recovery master receives only log entries for the tablets it has been assigned.

replicas on secondary storage can usually be recovered after the backup restarts, and in some situations (such as a datacenter-wide power failure) RAMCloud will depend on this information. However, for most backup crashes RAMCloud will simply rereplicate the lost information without waiting for the backup to restart; in this case the backup's secondary storage becomes irrelevant. When the coordinator crashes, a standby coordinator will become active and recover the crashed coordinator's state from external storage.

These cases are discussed in more detail in the sections that follow.

7. MASTER CRASH RECOVERY

RAMCloud's approach to replication requires fast crash recovery; as a result, speed was the most important factor in the design of RAMCloud's mechanism for master crash recovery. Most large-scale storage systems keep multiple copies of data online, so the system can continue normal operation even if one copy is lost. RAMCloud, however, only keeps one copy of data online, due to the high cost of DRAM. This means that data stored on a master will be unavailable from the time the master crashes until RAMCloud has completed crash recovery. We considered the possibility of providing service during crash recovery using data on secondary storage, but rejected it because it would have increased access latencies by 100-1000x and reduced throughput by a similar factor; this would render the data effectively unavailable. Thus, in RAMCloud, crash recovery time impacts availability: the faster RAMCloud can recover from a crash, the smaller the availability gaps.

Our target for RAMCloud is to recover from master crashes in 1-2 seconds. We picked this range based on discussions with developers of several large-scale applications. They told us that occasional 1-2 second gaps in availability would not significantly degrade the user experience, since there are already other factors that can cause delays of that magnitude, such as long-haul networking hiccups.

There is no way to recover the data from a crashed master in 1-2 seconds using the resources of a single node. For example, a large RAMCloud server today might have 256 GB of DRAM holding 2 billion objects. Reading all of the data from flash drives in one second requires about 1000 flash drives operating in parallel; transferring all of the data over the network in one second requires about 250 10 Gbs network interfaces, and entering all the objects into a hash table in one second requires a thousand or more cores.

RAMCloud provides fast recovery by dividing the work of recovery across many nodes operating concurrently. Figure 14 illustrates the basic mechanism. During normal operation, each master scatters its segment replicas across the entire cluster; this allows the replicas to be read concurrently during crash recovery. If the master crashes, the coordinator selects a collection of existing servers to take over the master's data. These servers are called *recovery masters*, and the coordinator assigns each of them a subset of the crashed master's tablets. At this point a massive data shuffle takes place: each backup reads segment replicas, divides their log entries into buckets for each recovery master, and transmits the buckets to the corresponding recovery masters. Each recovery master adds the incoming log entries to its log and creates a hash table entry for the current version of each live object. Once this process completes, the recovery masters become the new homes for the crashed server's tablets. This approach is scalable: as a RAMCloud cluster increases in size, it can recover more data in less time.

Several issues must be addressed in order to achieve scalability, such as distributing work uniformly across the participating components and ensuring that all of the components can operate concurrently. In addition, fast crash recovery requires fast failure detection, and the system must deal with secondary errors that occur during recovery. The remainder of this section addresses these issues in detail by working through the lifecycle of a crash and then addressing issues such as secondary crashes and zombies. Additional details on master crash recovery are available in [Ongaro et al. 2011] and [Stutsman 2013].

7.1. Scattering log segments

For fastest recovery, the segment replicas for each RAMCloud master must be distributed uniformly across all of the backups in the cluster. However, there are several additional factors that must be considered when assigning replicas to backups:

— Replica placement must reflect failure modes. For example, a segment's master and each of its replicas must reside in different racks, in order to protect against top-of-rack switch failures and other problems that disable an entire rack.
— Different backups may have different bandwidth for I/O (different numbers of disks, different disk speeds, or different storage classes such as flash memory); replicas should be distributed so that each device requires the same amount of time to read its share of the data during recovery.
— All of the masters write replicas simultaneously; they must avoid overloading any individual backup. Backups have limited space in which to buffer partially-written head segments.
— Utilization of secondary storage should be balanced across the cluster.
— Storage servers are continuously entering and leaving the cluster, which changes the pool of available backups and may unbalance the distribution of replicas.

Making decisions such as replica placement in a centralized fashion on the coordinator was not an option, because it would limit RAMCloud's scalability. For example, a cluster with 10,000 servers could create 600,000 or more replicas per second; this could easily cause the coordinator to become a performance bottleneck.

Instead, each RAMCloud master decides independently where to place each replica, using a technique inspired by Mitzenmacher's "Power of Two Choices" [Mitzenmacher 1996]. We call this approach *randomization with refinement*. When a master needs to select a backup and storage device for a segment replica, it chooses several candidates at random from a list of all the devices in the cluster. Then it selects the best candidate, using its knowledge of where it has already allocated segment replicas and information about the speed of each device (backups measure the speed of their devices when they start up and provide this information to the coordinator, which relays it on to masters). The best device is the one that can read its share of the master's segment replicas (including the new replica and any other replicas already assigned to it) most quickly during recovery. A device is rejected if it is in the same rack as the master or any other replica for the current segment. Once a device has been selected, the master contacts its backup server to reserve space for the segment. At this point the backup can reject the request if it is overloaded, in which case the master selects another candidate.

The use of randomization eliminates pathological behaviors such as all masters choosing the same backups in a lock-step fashion. Adding the refinement step provides a solution nearly as optimal as a centralized manager ([Mitzenmacher 1996] and [Azar et al. 1994] provide a theoretical analysis; Section 7.11 measures the benefits in RAMCloud). For example, if a master scatters 8,000 replicas across 1,000 devices using a purely random approach, devices will have 8 replicas on average. However, some devices are likely to end up with 15-20 replicas, which will result in uneven device utilization during recovery. With two choices, the device allocations will typically range from 8-10 replicas; RAMCloud uses five choices, which typically results in a difference of only one replica between the largest and smallest allocations. Randomization with refinement also handles the entry of new backups gracefully: a new backup is likely to be selected more frequently than existing backups until every master has taken full advantage of it.

RAMCloud masters mark one of the replicas for each segment as the *primary replica*. Only the primary replicas are read during recovery (unless they are unavailable), and the performance optimizations described above consider only primary replicas. Masters use a slightly simpler randomized assignment mechanism for non-primary replicas, which doesn't consider speed of reading.

Scattering segment replicas across all of the backups of the cluster is attractive not just from a recovery standpoint, but also from a performance standpoint. With this approach, a single master can take advantage of the full disk bandwidth of the entire cluster during large bursts of write operations, up to the limit of its network interface.

7.2. Fast failure detection

If RAMCloud is to recover quickly after crashes, then it must also detect crashes quickly. Traditional systems may take as long as 30 seconds to determine that a server has failed, but RAMCloud must make that decision within a few hundred milliseconds. RAMCloud does so using a randomized ping mechanism. At regular intervals (currently 100ms) each storage server chooses another server in the cluster at random and sends it a ping RPC. If that RPC times out (a few tens of milliseconds) then the server notifies the coordinator of a potential problem. The coordinator attempts its own ping to give the suspicious server a second chance, and if that also times out then the coordinator declares the server dead and initiates recovery.

This approach distributes the work of failure detection among all of the cluster servers. The coordinator only gets involved once it appears that a server may have crashed. Randomization introduces the possibility that a crashed server may not be pinged for several rounds, but the odds of this are low. If a cluster has at least 100 servers, the probability of detecting a crashed machine in a single round of pings is about 63%; the odds are greater than 99% that a failed server will be detected within five rounds.

Fast failure detection introduces a risk that RAMCloud will treat performance glitches as failures, resulting in unnecessary recoveries and possible system instability. For example, if a server becomes overloaded to the point where it cannot provide timely responses to ping RPCs, it could be declared crashed. Crash recovery will move its tablets to other servers, which may cause the overload to move to one of them, resulting in a cascade of false failures. We do not yet have enough experience with the system to know how frequently this may occur.

Fast failure detection also conflicts with some network protocols. For example, most TCP implementations wait 200ms before retransmitting lost packets; when RAMCloud uses TCP it must also use a longer RPC timeout value, which delays the start of crash recovery.

7.3. Server lists

The servers in a RAMCloud cluster maintain a fairly coherent view of cluster membership, and this plays an important role in crash recovery. Every RAMCloud storage server stores a *server list* containing information about each of the servers in the cluster, such as its current state, its network location, and the speed of its disks. The coordinator maintains the master copy of this list, which it updates as servers enter and leave the cluster. Whenever the state of a server changes, the coordinator immediately pushes that change out to all the servers in the cluster. When a server enlists in the cluster, the coordinator enters the server list in the UP state; its state changes to CRASHED when the coordinator detects its failure and begins crash recovery; and it is removed from the server list once crash recovery is complete. Once a server has been marked CRASHED, it will never again be UP (if it restarts, it does so with a new server id).

The server list is used for many purposes in RAMCloud. For example, it is used by masters to select backups for segment replicas as described in Section 7.1. The server list plays a particularly important role in crash recovery because it is used to disseminate information about crashes. For example, the RPC system checks the server list after RPC timeouts in order to decide whether to return “server not up” as described in Section 6. The replica manager for each master uses the server list to find out when backups crash, so it can rereplicate lost segment replicas (see Section 8).

7.4. Setup and partitioning

Once the coordinator decides that a master has crashed, it decides how the work of recovery will be divided across the servers of the cluster, and it provides each server with the information it needs to perform its share of the work without further coordination. The coordinator first sends RPCs in parallel to every backup in the cluster to collect information about all of the segment replicas currently available for the crashed master. The backups also return log digests from any open segments, which the coordinator uses as described in Section 4.5 to verify that a complete copy of the master’s log is available. The coordinator then divides up the work of recovery among a set of recovery masters by grouping the tablets of the crashed master into *partitions* as described below. Once this is done, the coordinator issues a second round of RPCs to all the backups to inform them of the partitions, which they need in order to divide up the log data; the backups then begin reading replicas from secondary storage. Finally, the coordinator sends an RPC to each of the chosen recovery masters, with information about the partition assigned to that recovery master as well as information about all of the available replicas from the crashed master’s log. At this point the recovery masters begin the replay process.

The coordinator must partition the crashed master’s tablets in such a way that each partition can be recovered by one recovery master in 1-2 seconds. To do this, it limits both the total amount of log data and the total number of log entries in each partition. Based on measurements of recovery time on our current hardware (see Section 7.11), these limits are currently set at 500 MB of log data and 2 million log entries.

In order for the coordinator to enforce the limits on partition size, it must know the utilizations of tablets on the crashed master. These statistics must be collected and maintained in a decentralized fashion to avoid creating a scalability bottleneck, and the statistics must survive crashes of either masters or the coordinator. We considered storing the statistics on the coordinator, but this would have resulted in considerable traffic from masters to keep the statistics up-to-date; in addition, the statistics would

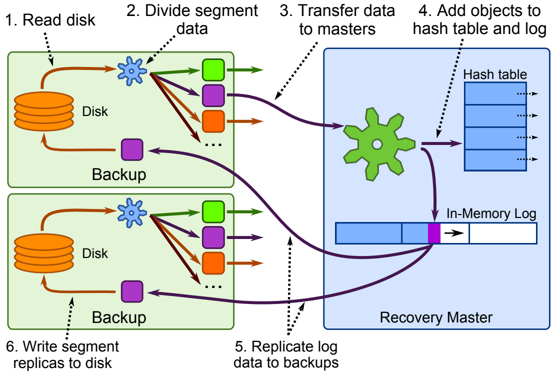

1. Read disk
2. Divide segment data
3. Transfer data to masters
4. Add objects to hash table and log
Disk
Backup
Disk
Backup
Hash table
In-Memory Log
Recovery Master
6. Write segment replicas to disk
5. Replicate log data to backups

Fig. 15. During recovery, segment data flows from disk or flash on a backup over the network to a recovery master, then back to new backups as part of the recovery master's log. All of these steps happen in parallel.

need to be stored durably to survive coordinator crashes, and this would have created additional performance and scalability problems for the coordinator.

As a result of these problems, we chose a distributed approach for managing tablet statistics. Each master keeps track of the total log space and log entry count consumed by each of its tablets. It outputs this information into its log in the form of a "tablet statistics" log entry written in each new head segment (see Figure 5). When a backup returns a log digest to the coordinator during recovery setup, it also returns the tablet statistics from the head segment, and the coordinator uses this information to partition the master's tablets. With this approach, the management of the statistics is completely distributed among the masters with no overhead for the coordinator; the log replication mechanism ensures durability and availability for the statistics. The tablet statistics are compressed in order to limit the log space they consume: exact information is recorded for large tablets, but only aggregate statistics are recorded for small tablets; see [Stutsman 2013] for details.

Once the coordinator has obtained the tablet statistics for the crashed master, it divides the master's tablets into partitions that satisfy the limits on space and log entry count. It uses a simple bin-packing algorithm that employs randomization with refinement in a fashion similar to that for replica placement in Section 7.1. If a tablet is too large to fit in a partition, the coordinator splits the tablet. For more details on the partitioning algorithm, see [Stutsman 2013].

7.5. Replay

The vast majority of recovery time is spent replaying segments to reconstruct partitions on the recovery masters. During replay the contents of each segment replica are processed in six stages as shown in Figure 15:

(1) A backup reads the replica from disk or flash into its memory.
(2) The backup divides the log records in the replica into separate buckets for each partition based on the table identifier and the hash of the key in each record.
(3) The records for each partition are transferred over the network to the recovery master for that partition. This process is driven from the recovery masters, which use their maps of segment replicas to request data from backups.
(4) The recovery master incorporates the data into its in-memory log and hash table.
(5) As the recovery master fills segments in memory, it replicates those segments over the network to backups with the same scattering mechanism used in normal operation.

(6) The backups write the new segment replicas to disk or flash.

RAMCloud harnesses concurrency in two dimensions during recovery. The first dimension is data parallelism: different backups read different segments from disk or flash in parallel, different recovery masters reconstruct different partitions in parallel, and so on. The second dimension is pipelining: all of the six stages listed above proceed in parallel, with a segment as the basic unit of work. While one segment is being read from disk on a backup, another segment is being partitioned by that backup's CPU, and records from another segment are being transferred to a recovery master; similar pipelining occurs on recovery masters. For fastest recovery all of the resources of the cluster must be kept fully utilized, including disks, CPUs, and the network.

7.6. Segment Replay Order

In order to maximize concurrency, recovery masters and backups operate independently. As soon as the coordinator contacts each backup to obtain its list of replicas, the backup begins prefetching replicas from disk and dividing them by partition. At the same time, masters fetch replica data from backups and replay it. Ideally backups will constantly run ahead of masters, so that data is ready and waiting whenever a recovery master requests it. However, this only works if the recovery masters and backups process replicas in the same order. If a recovery master accidentally requests the last replica in the backup's order then the master will stall: it will not receive any data to process until the backup has read all of its replicas.

In order to avoid pipeline stalls, each backup decides in advance the order in which it will read its replicas. It returns this information to the coordinator during the setup phase, and the coordinator includes the order information when it communicates with recovery masters to initiate recovery. Each recovery master uses its knowledge of backup disk speeds to estimate when each replica's data is likely to be loaded. It then requests replica data in order of expected availability. (This approach causes all masters to request replicas in the same order; we could introduce randomization to avoid contention caused by this lock-step behavior, but so far it does not seem to impact performance significantly.)

Unfortunately, there will still be variations in the speed at which backups read and process replicas. In order to avoid stalls because of slow backups, each master keeps several concurrent requests for replica data outstanding at any given time during recovery; it replays replica data in the order that the requests return.

Because of the optimizations described above, recovery masters will not replay segments in log order. Fortunately, the version numbers in log records allow the log to be replayed in any order without affecting the result. During replay the master simply retains the most recent version for each object and discards older versions. If there is a tombstone for the most recent version, then the object is deleted.

Although each segment has multiple replicas stored on different backups, backups prefetch only the primary replicas during recovery; reading more than one would waste valuable disk bandwidth. Masters identify primary replicas when scattering their segments as described in Section 7.1. During recovery each backup reports all of its segments, but it identifies the primary replicas and only prefetches the primary replicas from disk. Recovery masters request non-primary replicas only if there is a failure reading the primary replica, and backups load and partition these on-demand.

7.7. Cleanup

A recovery master has finished recovering its assigned partition once it has replayed data from each of the crashed master's segments. At this point it notifies the coordinator that it is ready to service requests for the data it has recovered. The coordinator updates its configuration information to indicate that the master now owns the tablets in the recovered partition, at which point the partition becomes available for client requests. Any clients attempting to access data on the failed server will have experienced RPC timeouts; they have been repeatedly asking the coordinator for new configuration information for the lost tablets, and the coordinator has been responding "try again later." Clients now receive fresh configuration information and retry their RPCs with the new master. Each recovery master can begin service independently without waiting for other recovery masters to finish.

Once all recovery masters have completed recovery, the coordinator removes the crashed master from its server list, and it propagates this information to the cluster as described in Section 7.3. When a backup receives the update, it frees the storage for the crashed master's segments.

7.8. Secondary failures

Unfortunately, additional failures may occur during the process described above. One or more recovery masters may fail; backups may fail, to the point where recovery masters cannot find replicas for one more segments; and the coordinator itself may fail. Furthermore, the coordinator may not be able to start recovery in the first place, if it cannot find a complete log.

RAMCloud uses a single mechanism to handle all of these problems; this was crucial in order to reduce the complexity of crash recovery. The coordinator repeatedly attempts to recover a crashed master, until there are no longer any tablets assigned to that master. Each attempt at recovery can make incremental progress in units of the partitions assigned to recovery masters. If a recovery master completes recovery successfully, the tablets in its partition are removed from those associated with the crashed master. A recovery master can abort its recovery if it encounters any problems, such as the inability to read any of the replicas for a particular segment or exhaustion of the master's log space (all errors are promoted to a single error: "this master couldn't recover its partition"). If this happens, or if the recovery master crashes, the tablets in its partition remain assigned to the crashed master. A particular recovery attempt completes once all recovery masters have either succeeded or failed. If there are still tablets assigned to the crashed master, then another recovery attempt is queued.

This mechanism also handles the case where recovery requires more partitions than there are masters available. In this case, each available master is assigned one partition and the other partitions are ignored during the current attempt. Once the current attempt completes, additional attempts will be started for the remaining tablets.

If the coordinator crashes during recovery, it will have left information on external storage about the crashed master. The new coordinator retrieves this information and starts a new recovery. The new coordinator does not try to continue with recoveries already in progress, since that would add complexity and the situation is unlikely to occur frequently. Recovery masters from an old recovery will continue working; when they notify the new coordinator that they have completed, the coordinator asks them to abort the recovery.

7.9. Multiple failures and cold start

If multiple servers crash simultaneously, RAMCloud can run multiple recoveries concurrently, as long as there are enough masters to serve as recovery masters. For example, if a RAMCloud cluster contains 5000 servers, each with 256 GB of DRAM, and a rack failure disables 40 of them simultaneously, the measurements in Section 7.11 indicate that all of the lost data could be recovered in about eight seconds.

However, if a large number of servers are lost at the same time, such as in a network partition, then it may not be possible to recover any of them. There are two issues that can prevent recovery. First, there may not be enough replicas available to assemble a complete log for any crashed master. Second, the remaining masters may not have enough unused log space to accommodate the lost data for any of the crashed masters. If either of these situations occurs, the coordinator will continually attempt recoveries but none will make any progress. The cluster will be unavailable until enough servers have restarted to provide the required data and capacity. Once this happens, recoveries will complete and the cluster will become available again.

The most extreme case is a *cold start* where the entire cluster crashes and restarts, such as after a datacenter power outage. RAMCloud handles cold starts using the existing crash recovery mechanism. After the coordinator restarts, it will detect a failure for every master in the cluster and initiate crash recovery. At first, recovery will be impossible, for the reasons given previously. Eventually, enough servers will restart for some recoveries to complete. As more and more servers restart, more recoveries will become possible, until the cluster eventually resumes full operation.

We considered an alternative approach to cold start that would be more efficient. Instead of treating all of the masters as crashed, they could be reconstructed exactly as they existed before the cold start. In this scenario, each restarting master would reclaim its own log data from backups, regenerate its in-memory log directly from the backup replicas, and reconstruct its hash table. This approach would eliminate the need to write new replicas for the recovered data, which would reduce cold start time by a factor of 4x (assuming 3x replication). However, this approach would require an entirely new mechanism with considerable complexity. It was not clear to us that the performance advantage would be significant, given all of the other costs of cold-starting a datacenter, so we chose to minimize complexity by reusing the existing crash recovery mechanism for cold start.

7.10. Zombies

RAMCloud assumes that a server has crashed if it cannot respond to ping requests. However, a temporary communication problem or performance glitch may cause the coordinator to decide a master has crashed, even though it is still alive. We refer to such masters as *zombies*. RAMCloud must prevent zombies from servicing client requests; otherwise a zombie server could produce inconsistent behavior for clients. For example, a zombie server might return stale data for an object that has been reconstructed on a recovery master, or it might accept a write request even though it no longer holds the legitimate copy of the object.

RAMCloud uses two mechanisms to ensure that zombies realize they are dead. The first mechanism prevents any writes by a zombie once crash recovery has started. In order to write data, a master must contact each of the backups for the head segment. However, the coordinator must have contacted at least one of these backups during the setup phase for recovery (otherwise it could not assemble a complete log); any server contacted by the coordinator will mark the master as crashed and refuse future replication requests from that server. As a result, a zombie will receive at least one rejection during its next write operation. It treats this as an indication that it may be a zombie, so it defers all incoming requests and contacts the coordinator to verify its status in the cluster. If the coordinator confirms that it is indeed dead, then it terminates itself.

Zombie reads are more difficult to prevent, since a zombie master can service them without communicating with any other servers in the cluster. RAMCloud extends the distributed failure detection mechanism to provide an additional mechanism for detecting zombies. As described in Section 7.2, each server periodically sends a ping request to another server chosen at random. In addition to confirming its liveness, the recipient of the ping checks its server list to see if the sender is UP; if not, it indicates

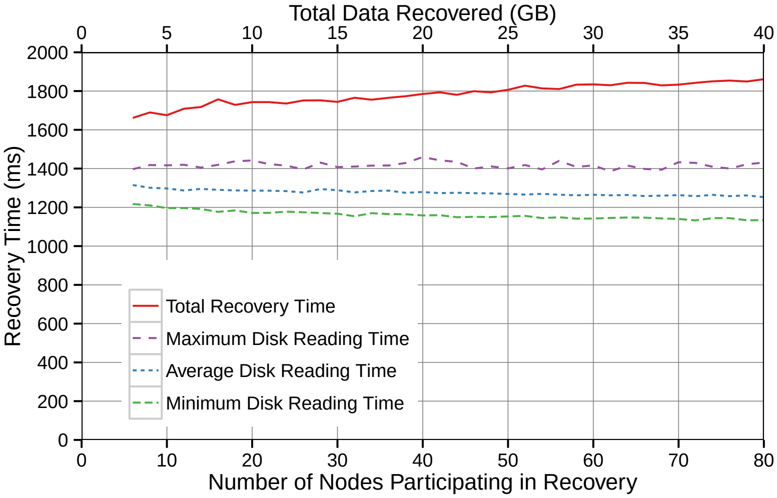


Fig. 16. Recovery performance as a function of cluster size. A master (not counted in "Nodes") was filled with N*500 MB of data (where N is the number of nodes participating in recovery), using 1 KB objects, then crashed and recovered. The y-axis measures total recovery time from when the master was determined to have crashed until all partitions were recovered and a client successfully accessed a recovered object. A horizontal line would indicate perfect scalability. Each node used for recovery ran one master and two backups, and contained two flash disks with a total of 460 MB/s read bandwidth. Each point is an average of five runs. The disk curves indicate the minimum, average, and maximum elapsed time for backups to finish reading replicas from disk.

that in its response. The sender treats that response as an indication that it may be a zombie, so it defers service and checks with the coordinator as described previously. In addition, if a ping request times out, the sender also checks its status with the coordinator; this handles situations where a zombie is partitioned from the rest of the cluster.

Unfortunately, the ping mechanism for detecting zombies is only probabilistic: a disconnected group of zombies could by chance select only each other for their ping requests. However, it is unlikely this situation would persist for more than a few rounds of pinging, so we assume that zombies will have detected their status before crash recovery completes. It is safe for zombies to continue servicing read requests during crash recovery: data cannot become stale until recovery completes and another server accepts a write request. To be safe, RAMCloud should enforce a minimum bound on recovery time; in the current implementation, recovery could complete quite quickly if a crashed master doesn't store much data.

7.11. Performance evaluation of master recovery

We used the test cluster described in Table II to measure the performance of master crash recovery. The results show that:

- The recovery mechanism is highly scalable: increasing the number of nodes in the cluster produces a near-linear increase in the rate at which data can be reconstructed after a master crash. In our 80-node cluster, RAMCloud recovered 40 GB of data from a single crashed master in about 1.9 seconds.
- An individual recovery master can recover 800MB of data per second if objects are large, or 2M objects per second if objects are small.

Table V. Throughput of a single recovery master as a function of object size. Each experiment used 80 backups, so the recovery master was the bottleneck; all objects were the same size in each experiment. Throughput is higher here than in Figure 16 because there is less contention for backup I/O bandwidth, network bandwidth, and memory bandwidth.

Object Size	**Throughput**	
(bytes)	**(Mobjs/sec)**	**(MB/sec)**
1	2.32	84
64	2.18	210
128	2.03	319
256	1.71	478
1024	0.81	824
2048	0.39	781
4096	0.19	754

— The randomized approach to replica placement is effective at distributing load evenly across backups during recovery.

The most important issue for master crash recovery is scalability: can RAMCloud take advantage of increases in cluster size to recover more data more quickly? If recovery throughput is scalable, then large clusters can be used both to reduce the total time for recovery and to recover masters with larger memories.

Figure 16 shows recovery speed over a series of measurements where we scaled both the size of the cluster and the amount of data recovered. The first experiment used 6 nodes to recover 3 GB of data from a crashed master; the next experiment used 7 nodes to recover 3.5 GB; and so on up to the final experiment, which used 80 nodes to recover 40 GB of data from the crashed master. The results demonstrate near-linear scalability: total recovery time increased only 12% across this range, even though the amount of data recovered increased by 13x.

In the experiments of Figure 16, the total throughput of each node was limited both by core count and memory bandwidth. Each node had only 4 cores, which was not quite enough to meet the needs of one recovery master replaying log data and two backups reading from flash disks and dividing log entries into buckets. In addition, there were many times during recovery where the aggregate memory bandwidth needed by these components exceeded the 10 GB/s capacity of the nodes. Newer processors provide both more cores and more memory bandwidth, which will improve recovery throughput and scalability. See [Stutsman 2013] for more details on these limitations.

We also analyzed the performance of a single recovery master to determine appropriate partition sizes; the results are shown in Table V. The table indicates that partitions should contain no more than 800MB of log data and no more than 2M log records to enable one-second recovery. If objects are small, the speed of recovery is limited by the per-object costs of updating the hash table. If objects are large, throughput is limited by network bandwidth needed for 3x replication. If 10Gbps Ethernet is used instead of Infiniband, partitions will need to be limited to 300MB.

Our final measurements in this section evaluate the effectiveness of the randomized replica placement mechanism described in Section 7.1: can it ensure that backups are evenly loaded during recovery? Figure 16 gives one indication that replicas were spread evenly across the available flash disks: the slowest disk took only about 10% more time to read all of its replicas than the average disk.

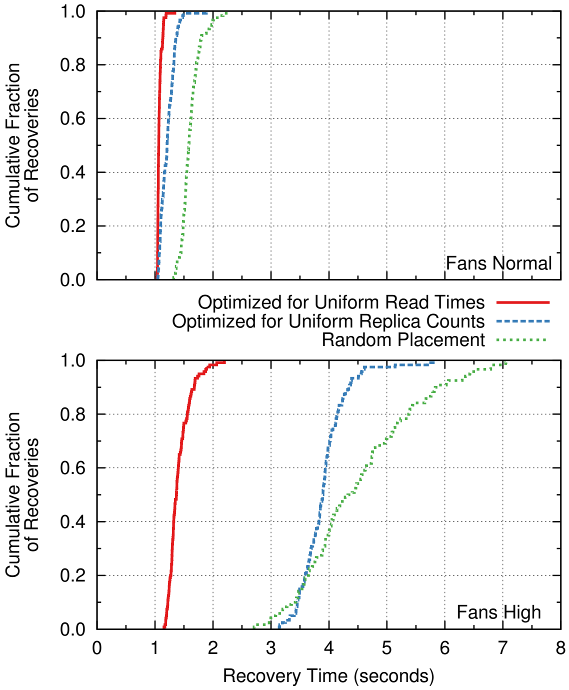


Fig. 17. The effectiveness of randomized segment replica placement. For this experiment hard disks were used for replica storage instead of flash disks. Each curve shows the cumulative distribution of recovery times over 120 recoveries. Each recovery used 20 recovery masters and 120 backups (120 total disks) to recover 12 GB of data from a crashed master. The disks provided about 12 GBs of combined read bandwidth, so ideally recovery should take about one second. Curves labeled “Optimized for Uniform Read Times” used the randomized replica placement algorithm described in Section 7.1; curves labeled “Random Placement” used a purely random approach with no refinement; and curves labeled “Optimized for Uniform Replica Counts” use the same algorithm as “Optimized for Uniform Read Time” except that disk speed was not considered (it attempted to place the same number of replicas on each backup). The top graph measured the cluster in its normal configuration, with relatively uniform disk performance; the bottom graph measured the system as it was shipped (unnecessarily high fan speed caused vibrations, resulting in a 4x variance in speed for half of the disks).

In addition, we ran a series of recoveries, each with a different randomized placement of replicas, and compared the distribution of recovery times for three variations of the placement algorithm: the full “randomization with refinement” algorithm, which considered both the number of replicas on each backup and the speeds of the backup devices; a purely random approach; and an intermediate approach that considered the number of segments but not device speed. The measurements used hard disks instead of flash disks, because the hard disks have significant variations in performance that create challenges for the replica placement algorithm. As shown in the top graph of Figure 17, the full algorithm produced better results than either of the alternatives: there was very little variation in its recovery time, and recovery time was almost always close to one second, which was optimal given the total available disk bandwidth. Average recovery time with the full algorithm was 33% better than purely random and 12% better than the “even segments” approach.

The bottom graph in Figure 17 shows results in a more extreme situation where disk speeds varied by more than a factor of 4x. In this scenario the full algorithm still produced relatively uniform recovery times, while both of the alternatives suffered significant performance degradation.

8. BACKUP CRASH RECOVERY

Each RAMCloud storage server typically serves as both master and backup. Thus, when a server crashes, it usually results in the loss of a backup as well as a master. Overall, backup crash recovery is simpler than master crash recovery; for example, a backup crash does not affect system availability, so RAMCloud need not take special measures to recover quickly. Nonetheless, backup crashes introduce several issues related to the integrity and proper replication of logs. This section describes the basic mechanism for recovering crashed backups, then discusses a few of the problems.

Backup crash recovery is handled by the masters in a totally distributed fashion. A master learns of the failure of a backup through the server list mechanism described in Section 7.3. When this happens, it assumes conservatively that any of its segment replicas stored on the backup have been lost permanently. To ensure proper replication of its log, it creates new replicas to replace the ones stored on the crashed backup. The masters carry out this process independently and in parallel.

When a backup restarts after a crash, it is assigned a new identity in the cluster but it will preserve any of the replicas on its secondary storage that are still needed. If a replica has been rereplicated by its master, then it is no longer needed and can be discarded. However, if the replica has not yet been replaced, then it must be retained to ensure adequate redundancy for crash recovery. For example, if the replica's master crashed at the same time as the backup and has not yet been recovered, then the replica on the restarting backup must be made available for use in the master's recovery. The backup decides whether to retain each replica by checking the state of the replica's master in its server list:

— If the master is no longer in the server list, it must have crashed and been successfully recovered, so the replica can be freed.
— If the master's state is CRASHED, the replica must be retained until recovery completes.
— If the master is up, then it will eventually replace the replica, even if the backup has restarted. The backup occasionally checks with the master to see if the replica is still needed. Once the master has rereplicated the segment, the backup frees its replica.

Backup crashes create two problems related to log integrity. The first problem occurs when a replica is lost for a master's head segment. It is possible for the master to replace the lost replica, write more objects to the head segment, and then crash. During this time, the backup may have restarted. If this happens, the restarted backup's replica of the head segment must not be used in the master's recovery, since it is incomplete. To handle this situation, the master provides information to the coordinator after rereplicating the head segment (but before adding any new log entries), which the coordinator uses to ignore the obsolete replica in future crash recoveries. For details on this mechanism, see [Stutsman 2013].

The second integrity problem arises if a master crashes while rereplicating a segment. In this situation, the partially-written replica must not be used in the master's crash recovery. However, the crash recovery mechanism assumes that all available replicas of a segment are equally valid, so it is safe to use any of them during recovery. To prevent data loss, rereplication uses a special “atomic” mode in which the backup will consider the replica invalid (and thus not offer it during recovery) until the master indicates that it is complete.

9. COORDINATOR CRASH RECOVERY

The coordinator's role is to manage the cluster configuration, so the only data it stores is metadata about the cluster. In RAMCloud 1.0 the coordinator keeps two kinds of metadata. First, it stores information about each server, which is kept in the server list. Second, the coordinator stores information about each table, including the name and identifier for the table, its tablet structure, and the identifier of the server storing each tablet. All of this state is kept in the coordinator's memory during normal operation, but it must survive coordinator crashes.

In order to ensure durability of its metadata, the coordinator writes the metadata to an external fault-tolerant key-value store. We currently use ZooKeeper as the external storage system [Hunt et al. 2010], but the coordinator accesses it through an internal interface that can support any storage system offering a key-value data model. Whenever the coordinator updates its in-memory state, it also writes the changes to ZooKeeper, and it does this synchronously before responding to the RPC that triggered the update. The coordinator stores one object in ZooKeeper for each slot in its server list, plus one object for each table. If the coordinator crashes, one of several standby coordinators will be chosen as the new active coordinator; it reads all of the information on external storage to initialize its in-memory state.

The coordinator stores an additional *leader object* in ZooKeeper to hold information about the active coordinator. The leader object acts as a form of lease [Gray and Cheriton 1989] for the active coordinator. The active coordinator must update the leader object regularly in order to preserve its authority; if it does not, then a standby coordinator will overwrite the leader object with its own information to become active. The leader object is also used by storage servers and clients to find the current coordinator, and to locate a new coordinator if the current one crashes.

Updates to coordinator state are typically distributed in nature: not only must the coordinator update its own state, but it must usually inform other servers of the state change as well. For example, when the coordinator creates a new table, it must notify one or more masters to take ownership of the tablets for the new table; when it updates the server list, it must ensure that the update is propagated to all of the servers in the cluster. This creates potential consistency problems, since the coordinator may crash partway through a distributed update (e.g., a new table has been recorded in ZooKeeper, but the table's master has not been told to take ownership).

In order to ensure the consistency of distributed updates, the coordinator updates the appropriate ZooKeeper object before sending updates to any other server or returning information to a caller; this ensures that future coordinators will know about any partially-completed updates. When a new coordinator reads in data from ZooKeeper, it identifies updates that may not have finished (see following paragraph for details), and it reissues all of the notifications. This means that some notifications may occur multiple times, so they have all been designed with idempotent semantics. For example, when the coordinator notifies a master to take ownership of a tablet, the RPC semantics are "take ownership of the following tablet; if you already own it, do nothing." The reissued notifications for different updates may be sent in any order, since updates for different ZooKeeper objects are independent and only a single update is in progress for each object at a time.

In order to minimize the number of updates that must be reissued during coordinator failover, each update is assigned a unique sequence number, which is stored in the ZooKeeper object. The coordinator keeps track of incomplete updates and occasionally writes a special ZooKeeper object containing the smallest sequence number not yet completed. During coordinator failover, the new coordinator only needs to reissue updates with sequence numbers greater than or equal to this value.

10. LIMITATIONS

This section discusses limitations in the current RAMCloud system.

10.1. Geo-replication

We have not yet attempted to support geo-replication in RAMCloud. RAMCloud assumes that any server in the cluster can communicate at low latency with any other server in the cluster, which is only true if all of the servers are in the same datacenter. This means that a datacenter-wide outage, such as a power failure, will make the cluster unavailable. In order to continue providing services during such outages, many large-scale applications require their data to be replicated in multiple geographically distributed datacenters (*geo-replication*). RAMCloud could potentially be used with geo-replication in either of two ways. The first alternative is to perform synchronous geo-replication (i.e., once a write completes, the system guarantees that at least one copy is stored in a different datacenter); this approach would result in long latencies for writes, but could still support fast reads. The second alternative is to perform geo-replication asynchronously (when a write returns, there will be durable local replication, but geo-replication may not yet be complete); this approach would retain fast write times, but could result in the loss of small amounts of data in the event of a datacenter outage.

10.2. System scale and fast crash recovery

RAMCloud's crash recovery mechanism creates a trade-off between system scale and crash recovery time: a small cluster cannot store very much data on each node if it requires fast recovery. Each recovery master can recover roughly 500 MB of log data in one second. Thus, if a cluster with N nodes is to support one-second crash recovery, each master in the cluster must store no more than $500 \times$N MB of data. For example, in a 10-node cluster, each node can store only 5 GB of data if it is to recover in one second; if 10-second recovery is acceptable, then each node can store 50 GB of data. Doubling the cluster size will quaduple the total amount of data it can store, since both the number of nodes and the amount of data per node will double.

The scale-vs.-recovery-time trade-off also impacts large clusters as server memory sizes increase. When we started the project, a server with 64 GB was considered large; servers of that size can be recovered in one second by a cluster with 128 nodes. However, in 2014 a large server might have 256 GB of memory, which requires a 512-node cluster for one-second recovery. Even larger cluster sizes will be required as memory sizes increase in the future. This is another example of uneven scaling: memory size has increased faster than other technologies such as network bandwidth. In order for RAMCloud's recovery approach to handle future scaling, increases in memory sizes must be matched by increases in network bandwidth, memory bandwidth, and number of cores per server, so that each server can recover more data in one second.

10.3. Data model

We chose a key-value store as RAMCloud's initial data model because its simplicity gave us the best chance of meeting our goals of low latency and large scale. However, we believe that higher-level features such as secondary indexes and multi-object transactions would make it significantly easier to build applications on RAMCloud. It is an open research question whether the full SQL/ACID data model of a relational database can be implemented at the latency and scale for which RAMCloud is designed, but we plan to experiment with higher-level features in the RAMCloud data model until either we reach this goal or discover fundamental limits.

We intend for RAMCloud to eventually provide linearizable semantics for all operations, but the linearizability in RAMCloud 1.0 is not complete. The current version of the system implements only "at-least-once" semantics, not linearizability: in the face of crashes, operations may execute multiple times. We are currently working on the remaining infrastructure required for full linearizability.

10.4. Protection

RAMCloud 1.0 does not contain any protection mechanisms: any client can modify any data in the system. However, we think that multi-tenant cloud computing environments are one of the most attractive places to use RAMCloud. These environments will require protection, at least at the granularity of tables, and also a mechanism for scoping table names so that different applications can reuse the same names.

10.5. Configuration management

RAMCloud provides only rudimentary features for managing the configuration of tablets. The system provides basic mechanisms for splitting tablets and moving tablets from one server to another, but it does not yet have higher-level policy modules that monitor server load and decide when and how to reconfigure. We expect to implement these features in future versions of the system.

11. LESSONS

One of the advantages of working on a large and complex system over several years with many developers is that certain problems occur repeatedly, and the process of dealing with those problems exposes techniques that have broad applicability. This section discusses a few of the most interesting problems and techniques that have arisen in the RAMCloud project so far.

11.1. Logging

When we first decided to base RAMCloud's storage mechanism around an append-only log, the decision was made primarily for performance reasons: it allowed the system to collect updates together and write them to secondary storage in large sequential chunks. However, the log-structured approach has provided numerous other benefits, some of which we did not realize until later in the project:

- The log facilitates crash recovery by organizing information as a collection of self-identifying log entries that can be replayed after a server crash.
- The log provides a convenient place to store additional metadata needed during crash recovery; this is much more efficient than using an external system such as ZooKeeper. For example, RAMCloud masters leave tablet usage statistics in the head segment of the log.
- The log enables consistent replication: markers can be placed in the log to indicate consistent points, so groups of related updates can be appended atomically. This feature will be key in implementing linearizability and transactions in the future.
- The immutability of the log makes concurrent access simpler and more efficient. For example, it allows the cleaner to run concurrently with read and write operations.
- The log provides a convenient way to neutralize zombie servers: once the backups for the zombie's head segment have been notified, the zombie cannot make any more updates.
- Perhaps most surprisingly, the log-structured approach uses DRAM quite efficiently; it enables higher memory utilization than any other storage allocator we have encountered.

11.2. Randomization

We have found randomization to be one of the most useful tools available for developing large-scale systems. Its primary benefit is that it allows centralized (and hence nonscalable) mechanisms to be replaced with scalable distributed ones. For example, we used randomization to create distributed implementations of replica assignment (Section 7.1) and failure detection (Section 7.2).

Randomization also provides an efficient and simple tool for making decisions that involve large numbers of objects. For example, when the coordinator partitions a crashed master's tablets among recovery masters, it must assign each tablet to one of 100 or more partitions. RAMCloud uses randomization with refinement for this: for each tablet, it selects a few candidate partitions at random and picks the most attractive of those. This approach is faster than scanning all of the partitions to find the best one, and it is simpler than creating a special-purpose data structure and algorithm to identify the optimal partition quickly. As the scale of a system increases, it becomes less and less important to make the best possible choice for each decision: a large number of "pretty good" decisions work just about as well.

11.3. Layering conflicts with latency

Layering is an essential technique in building large software systems because it allows complex functionality to be decomposed into smaller pieces that can be developed and understood independently. However, low latency is difficult to achieve in a system with many layers. Each layer crossing adds a small amount of delay, and these delays accumulate over dozens or hundreds of layer crossings to produce high latency without an obvious single culprit. Problems often come from highly-granular interfaces that require numerous small calls into a module; latency accumulates both from the cost of the method calls and from work that is performed repeatedly, such as table lookups and bounds checks. In traditional disk-based storage systems the overheads from layering are not noticeable because they are dwarfed by disk seek times; in RAMCloud, where we aim for request service times under 1 μs, layering accounts for a significant fraction of total latency.

Unfortunately, it is difficult to design a system in a modular fashion without incurring high overheads from layer crossings, especially if each module is designed independently. One approach we have used is to start from an end-to-end analysis of a task whose overall latency is important, such as servicing small read requests. We then ask the question "what is the minimum amount of work that is inevitable in carrying out this task?" Then we search for a clean module decomposition that comes close to the minimum work and introduces the fewest layer crossings. One key element of the approach is to design "thick interfaces," where a large amount of useful work is done for each layer crossing.

Another way of achieving both low latency and modularity is to design for a *fast path*. In this approach, initial setup involves all of the layers and may be slow, but once setup has been completed, a special fast path skips most of the layers for normal operation. Kernel bypass is an example of this: the kernel must be invoked to map NIC device registers into the application's address space, pin buffer pages in memory, etc. Once this is done, the application can communicate directly with the NIC without passing through the kernel.

11.4. Ubiquitous retry

Retry has turned out to be a powerful tool in building a large-scale fault-tolerant system. The basic idea is that an operation may not succeed on its first attempt, so the invoker must be prepared to try it again. We use retry for many purposes in RAMCloud, such as:

> *Fault tolerance.* Any system that tolerates faults must include a retry mechanism: if an operation fails, the system must correct the problem and then retry the operation. For example, if a server crashes, RAMCloud reconstructs the server's data on other servers and then retries any operations made to the crashed server.
>
> *Configuration changes.* Retry allows configuration changes to be detected and handled lazily. For example, when a tablet moves, clients may continue to send requests to the tablet's old master. When this happens, the master informs the client that it no longer stores the tablet. The client retrieves new configuration information from the coordinator and then retries the operation.
>
> *Blocking.* There are numerous situations in which a server cannot immediately process a request. For example, when a server starts up it must accept some RPCs as part of its bootstrap process, but it is not yet ready to provide full service. Or, if a server's log memory fills up, it cannot accept additional write operations until the cleaner runs and/or objects are deleted. Indefinite waits on the server are dangerous because they consume server resources such as threads, which can produce congestion or deadlock (for example, queued write reqests may block delete requests). These problems are particularly acute in large-scale systems, where there could be hundreds or thousands of clients attempting the blocked operation. RAMCloud's solution is to reject the requests with a retry status (the rejection can also indicate how long the sender should wait before retrying).

We initially introduced retries in an ad hoc fashion: only a few RPCs caused retries, and the retry status was returned to the outermost client wrapper, where it was handled in an RPC-specific way. Over time we found more and more uses for retry in more and more RPCs. Eventually we refactored the RPC system to include a mechanism for defining reusable *retry modules* that implement retries of various forms. For example, one retry module implements "tablet moved" retries for all RPCs that access objects using a table id and key. This approach allows many special cases to be masked at a low level, so that higher-level software need not be aware of them.

Retries are also used in several places outside the RPC system. For example, the coordinator retries a master crash recovery if the previous attempt did not completely recover.

11.5. DCFT modules and rules-based programming

RAMCloud contains several modules that must manage distributed resources in a concurrent and fault-tolerant way (*DCFT modules*). A DCFT module issues requests in parallel to a collection of distributed servers, and recovers from failures so that higher levels of software need not deal with them. For example, the code on each master that manages segment replicas is a DCFT module, as is the coordinator code that propagates server list updates to all the servers in the cluster, as is the client-level code that manages a multi-read operation. DCFT modules are exceptionally difficult to implement. Their behavior is highly nondeterministic, so it is difficult to code them in the traditional imperative fashion.

We eventually discovered that a rules-based approach works well for DCFT modules. With the rules-based approach, a DCFT module is based around a retry loop that repeatedly applies a set of rules until a goal is reached. Each rule consists of a condition to check against internal state and an action to execute if the condition is satisfied. Each rule handles a particular situation to make incremental progress towards the goal. Rules can fire in any order, depending on events such as RPC completions and

Table VI. A rough comparison between RAMCloud, MICA [Lim et al. 2014], and FaRM [Dragojević et al. 2014] for reads and writes of small objects. The RAMCloud and MICA configurations measured a single server; FaRM measured a cluster of 20 servers. RAMCloud and FaRM used Infiniband networking; MICA used 8 parallel 10-Gb Ethernet connections. RAMCloud used 2.9 GHz Xeon X3470 CPUs; MICA used 2.7 GHz Xeon E5-2680 CPUs; MICA used 2.4 GHz Xeon E5-2665 CPUs. Multi-read and multi-write latencies are omitted for RAMCloud, since they depend on the number of objects in each RPC.

Configuration	**Total Read Throughput** (Mobjects/s)	**Cores**	**Normalized Throughput** (Mobj/s/core)	**Read Latency** (µs)	**Durable Update Latency** (µs)
RAMCloud single ops	0.9	4	0.25	4.8	15.3
RAMCloud multi-ops	6.0	4	1.5		
MICA	60	16	3.75	24	N/A
FaRM key-value store	146	320	0.46	35	120

errors. The rules-based approach has made it significantly easier to implement DCFT modules; it is described in more detail in [Stutsman et al. 2014].

12. COMPARISONS WITH OTHER SYSTEMS

It is difficult to make meaningful comparisons between RAMCloud and other systems because virtually all other storage systems are optimized for throughput, not latency; in particular, few systems have been optimized to use kernel bypass for network communication. Nonetheless, this section compares RAMCloud with two recent systems that support kernel bypass and two high-performance key-value stores that do not yet support kernel bypass.

12.1. MICA and FaRM

MICA [Lim et al. 2014] and FaRM [Dragojević et al. 2014] are recent research prototypes that use high-speed networks with kernel bypass to implement high-performance storage systems. MICA implements a volatile cache in DRAM; it includes numerous optimizations to maximize throughput. FaRM implements distributed shared memory that can be accessed using Infiniband RDMA operations; in addition, FaRM implements a key-value store using the distributed shared memory, and it offers durable updates that replicate data to SSDs on multiple servers.

As can be seen in Table VI, both of these systems have higher throughput than RAMCloud, but RAMCloud's latency is considerably lower than either MICA or FaRM. The published read throughputs for the systems use configurations with different numbers of servers and cores, so Table VI normalizes the read throughput in terms of objects/second/core. MICA's read throughput per server core is roughly 10x higher than either RAMCloud or FaRM for single-object requests. Even if RAMCloud uses multi-read requests, its throughput per core is still less than half that of MICA. On the other hand, RAMCloud's read latency is 5x lower than MICA and more than 6x lower than FaRM. RAMCloud's latency for durable writes is about 8x lower than FaRM (MICA does not support durable updates).

MICA illustrates how architectural restrictions can enable significant performance improvements. In order to achieve its high throughput, MICA partitions the data stored on the server. Each core runs a thread that manages a separate partition and receives requests with a separate NIC connection; clients send requests directly to the appropriate thread. This eliminates the overheads that RAMCloud suffers when handing off requests between the dispatch thread and worker threads. It also eliminates most synchronization and minimizes the movement of data between caches.

MICA's approach is highly efficient, but unfortunately it depends on functional limitations of the MICA architecture such as its lack of durability and fault tolerance; in its current form it could not be used in RAMCloud. First, MICA cannot handle the

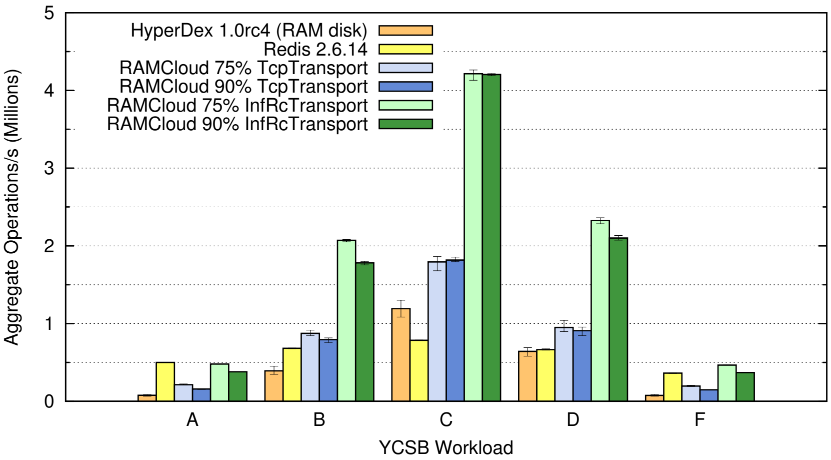


Fig. 18. Performance of HyperDex, Redis, and RAMCloud under the default YCSB workloads [Cooper et al. 2010]. Workloads B, C, and D are read-heavy workloads, while A and F are write-heavy; workload E was omitted because RAMCloud does not support scans. Y-values represent aggregate average throughput of 24 YCSB clients running on 24 separate nodes. Each client performed 100 million operations on a data set of 100 million keys. Objects were 1 KB each (the workload default). An additional 12 nodes ran the storage servers. HyperDex and Redis used kernel-level sockets over Infiniband. RAMCloud was measured with both TcpTransport (kernel-level sockets over Infiniband) and InfRcTransport (Infiniband with kernel bypass), and at 75% and 90% memory utilizations (each server's share of the 10 million total records comprised 75% or 90% of its total log memory). Each data point is averaged over 3 runs.

requirement that led to RAMCloud's threading architecture (the need to handle a ping request that checks for server liveness in the middle of a long-running request). In MICA, clients cannot assume that any particular request will ever receive a response; a slow or missing response is treated as a cache miss; thus the MICA approach cannot be used for a storage system that guarantees persistence. Second, the MICA architecture cannot safely handle multi-level requests, such as when a master receives a write request and then issues replication requests to backups. These could result in a distributed deadlock in the MICA architecture, where all cores are servicing top-level write requests, so none can serve the replication requests. RAMCloud's centralized dispatch thread allows it to manage resources to deal with these situations, albeit at a significant cost in performance.

The FaRM results show that distributed shared memory is not a good building block for higher-level facilities such as a key-value store: it is more efficient to implement the higher-level mechanism directly. The lowest level in FaRM implements distributed shared memory, and the RDMA operations at this level are quite efficient (nearly twice as fast as RAMCloud's RPCs). However, when a key-value store is then implemented as a layer above distributed shared memory, multiple RDMA operations must be invoked for each operation in the key-value store. As a result, the higher-level operations are much more expensive in FaRM than in RAMCloud. Updates are particularly expensive in FaRM because they require separate RDMA operations to synchronize between remote clients, allocate memory, and so on. RAMCloud implements a key-value store directly; the client sends a single RPC to the master, which can perform all of the synchronization and allocation locally at high speed. In this case the fastest application-level performance is not achieved by using the fastest low-level primitives.

12.2. Redis and Hyperdex

We also compared RAMCloud with HyperDex [Escriva et al. 2012] and Redis [red 2014], which are high-performance in-memory key-value stores. Redis keeps all of its data in DRAM and uses logging for durability, like RAMCloud. However, it offers only weak durability guarantees: the local log is written with a one-second fsync interval, and updates to replicas are batched and sent in the background (Redis also offers a synchronous update mode, but this degrades performance significantly). HyperDex [Escriva et al. 2012] offers similar durability and consistency to RAMCloud, and it supports a richer data model, including range scans and efficient searches across multiple columns. However, it is a disk-based system. Neither system takes advantage of kernel bypass for networking.

We used the YCSB [Cooper et al. 2010] benchmark suite to compare throughput for RAMCloud, HyperDex, and Redis. In order to make the systems comparable, we configured HyperDex to use a RAM-based file system to ensure that no operations wait for disk I/O, and we did not use the synchronous update mode in Redis. We configured all of the systems to communicate over Infiniband using TCP through the kernel, which meant that RAMCloud did not use its fastest transport. All systems were configured with triple replication.

As shown in Figure 18, RAMCloud outperforms HyperDex in every scenario, even when RAMCloud uses the slower TCP transport and runs at high memory utilization and despite configuring HyperDex so that it does not write to disks. RAMCloud also outperforms Redis, except in write-dominated workloads A and F. In these cases RAMCloud's throughput is limited by RPC latency: it must wait until data is replicated to all backups before replying to a client's write request, whereas Redis does not.

Figure 18 also contains measurements of RAMCloud using its fastest transport, which uses Infiniband with kernel bypass. This is the normal transport used in RAMCloud; it more than doubles read throughput and matches Redis' write throughput at 75% memory utilization. RAMCloud is 25% slower than Redis for workload A when RAMCloud runs at 90% utilization, but Redis uses the jemalloc [Evans 2006] memory allocator, whose fragmentation issues would likely require memory utilization less than 50% (see Figure 7). We doubt that Redis would benefit substantially if modified to use a faster transport, because its asynchronous approach to durability makes it less reliant on latency for performance than RAMCloud.

13. OTHER RELATED WORK

There are numerous examples where DRAM has been used to improve the performance of storage systems. Early experiments in the 1980s and 1990s included file caching [Ousterhout et al. 1988] and main-memory database systems [DeWitt et al. 1984; Garcia-Molina and Salem 1992]. In recent years, large-scale Web applications have found DRAM indispensable to meet their performance goals. For example, Google keeps its entire Web search index in DRAM [Barroso et al. 2003]; Facebook offloads its database servers by caching tens of terabytes of data in DRAM with memcached [mem 2011]; and Bigtable allows entire column families to be loaded into memory [Chang et al. 2008]. RAMCloud differs from these systems because it keeps all data permanently in DRAM (unlike Bigtable and Facebook, which use memory as a cache on a much larger disk-based storage system) and it is general-purpose (unlike the Web search indexes).

There has recently been a resurgence of interest in main-memory databases. Examples include H-Store [Kallman et al. 2008] and HANA [Sikka et al. 2012]. Both of these systems provide full RDBMS semantics, which is a richer data model than RAM-

Cloud provides, but neither is designed to operate at the low latency or large scale of RAMCloud.

RAMCloud's data model and use of DRAM as the primary storage location for data are similar to various "NoSQL" storage systems. Section 12.2 has already discussed Redis and HyperDex. Memcached [mem 2011] stores all data in DRAM, but it is a volatile cache with no durability. Other NoSQL systems like Dynamo [DeCandia et al. 2007] and PNUTS [Cooper et al. 2008] also have simplified data models, but do not service all reads from memory.

We know of no system that can match RAMCloud's low read and write latencies for remote accesses. As described in Section 12.1, MICA and FaRM use kernel bypass to achieve higher throughput than RAMCloud, but their latencies are 5-6x as high as RAMCloud's.

RAMCloud's storage management is superficially similar to Bigtable [Chang et al. 2008] and its related LevelDB library [lev 2014a]. For example, writes to Bigtable are first logged to GFS [Ghemawat et al. 2003] and then stored in a DRAM buffer. Bigtable has several different garbage collection mechanisms referred to as "compactions", which flush the DRAM buffer to a GFS file when it grows too large, reduce the number of files on disk, and reclaim space used by "delete entries" (analogous to tombstones in RAMCloud and called "deletion markers" in LevelDB). Unlike RAMCloud, the purpose of these compactions is not to reduce backup I/O, nor is it clear that these design choices improve memory efficiency. Bigtable does not incrementally remove delete entries from tables; instead it must rewrite them entirely. LevelDB's generational garbage collection mechanism [lev 2014b], however, is more similar to RAMCloud's segmented log and cleaning. Neither Bigtable nor LevelDB aims for latency as low as RAMCloud's.

RAMCloud's log-structured approach to storage management was influenced by ideas introduced in log-structured file systems [Rosenblum and Ousterhout 1992]. Much of the nomenclature and general techniques are shared, such as log segmentation, cleaning, and cost-benefit selection. However, RAMCloud differs in its design and application. The key-value data model, for instance, allows RAMCloud to use simpler metadata structures than LFS. Furthermore, as a cluster system, RAMCloud has many disks at its disposal, which reduces contention between cleaning and regular log appends.

Efficiency has been a controversial topic in log-structured file systems [Seltzer et al. 1993; Seltzer et al. 1995] and additional techniques have been introduced to reduce or hide the cost of cleaning [Blackwell et al. 1995; Matthews et al. 1997]. However, as an in-memory store, RAMCloud's use of a log is more efficient than LFS. First, RAMCloud need not read segments from disk during cleaning, which reduces cleaner I/O. Second, RAMCloud may run its disks at low utilization, making disk cleaning much cheaper with two-level cleaning. Third, since reads are always serviced from DRAM they are always fast, regardless of locality of access or placement in the log.

Although most large-scale storage systems use symmetric online replication to ensure availability, Bigtable is similar to RAMCloud in that it implements fast crash recovery (during which data is unavailable) rather than online replication. Many other systems, such as Bigtable and GFS, use aggressive data partitioning to speed up recovery. Many of the advantages of fast crash recovery were outlined by Baker in the context of distributed file systems [Baker and Ousterhout 1991; Baker 1994].

Randomization has been used by several other systems to allow system management decisions to be made in a distributed and scalable fashion. For example, consistent hashing uses randomization to distribute objects among a group of servers [Stoica et al. 2003; DeCandia et al. 2007], and Sparrow uses randomization with refinement to schedule tasks for large-scale applications [Ousterhout et al. 2013]. Mitzenmacher and others have studied the theoretical properties of randomization with refinement and have shown that it produces near-optimal results [Mitzenmacher 1996; Azar et al. 1994].

14. HISTORY AND STATUS

We began exploratory discussions about RAMCloud in 2009, and we started implementation in earnest in the spring of 2010. By late 2011, many of the basic operations were implemented and we were able to demonstrate fast crash recovery for masters; however, the system was not complete enough to use for real applications. In January 2014 we tagged version 1.0; that version includes all of the features described in this paper, and we believe it is mature enough to support applications. The system currently consists of about 100,000 lines of heavily commented C++11 code and another 45,000 lines of unit tests; it includes client bindings for C++, C, Java, and Python. We have tried to make the implementation "production quality," not just a research prototype. Source code for the system is freely available.

Usage of RAMCloud has been limited to date because the high-speed networking required by RAMCloud is still not widely available (RAMCloud's performance advantage drops significantly if it is used on 1Gbs networks over kernel TCP). Nonetheless, several groups have experimented with RAMCloud. Most notable among them is the ONOS project at the Open Networking Laboratory, which is using RAMCloud as the storage system for an operating system for software-defined networks [Berde et al. 2014]. ONOS requires low-latency durable storage in order to provide routing information to switches in a timely fashion.

15. CONCLUSION

RAMCloud is an experiment in achieving low latency at large scale: our goal is to build a storage system that provides the fastest possible access to the largest possible datasets. As a result, RAMCloud uses DRAM as the primary location for data, and it combines the main memories of thousands of servers to support large-scale datasets. RAMCloud employs several novel techniques, such as a uniform log-structured mechanism for managing all storage, a networking layer that bypasses the kernel to communicate directly with the NIC using a polling approach, and an approach to availability that substitutes fast crash recovery for online replication. The result is a system more than 1000x faster than the disk-based storage systems that have been the status quo for most of the last four decades.

We intentionally took an extreme approach in RAMCloud, such as using DRAM for storage instead of flash memory and designing the system to support at least 10,000 servers. We believe that this approach will maximize the amount we learn, both about how to structure systems for low latency and large scale and about what sort of applications an extreme low-latency system might enable.

Our ultimate goal for RAMCloud is to enable new applications that could not exist previously. We do not yet know what those applications will be, but history suggests that large performance improvements are always followed by exciting new applications that take advantage of the new capabilities. As RAMCloud and other low-latency storage systems become widely available, we look forward to seeing the applications that result.

REFERENCES

2011. memcached: a distributed memory object caching system. (Jan. 2011). http://www.memcached.org/.

2013. Google Performance Tools. (March 2013). http://goog-perftools.sourceforge.net/.

2013. Memory that never forgets: non-volatile DIMMs hit the market. (April 2013). http://arstechnica.com/information-technology/2013/04/memory-that-never-forgets-non-volatile-dimms-hit-the-market/.

2014. Cassandra. (April 2014). http://cassandra.apache.org/.

2014a. Leveldb - A fast and lightweight key/value database library by Google. (April 2014). http://code.google.com/p/leveldb/.

2014b. Leveldb file layouts and compactions. (April 2014). http://leveldb.googlecode.com/svn/trunk/doc/impl.html.

2014. Redis. (April 2014). http://www.redis.io/.

Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. ACM, New York, NY, USA, 53–64. DOI:http://dx.doi.org/10.1145/2254756.2254766

Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. 1994. Balanced allocations (extended abstract). In *Proceedings of the twenty-sixth annual ACM symposium on theory of computing (STOC '94)*. ACM, New York, NY, USA, 593–602. DOI:http://dx.doi.org/10.1145/195058.195412

Mary Baker and John K. Ousterhout. 1991. Availability in the Sprite Distributed File System. *Operating Systems Review* 25, 2 (1991), 95–98.

Mary Louise Gray Baker. 1994. *Fast Crash Recovery in Distributed File Systems*. Ph.D. Dissertation. University of California at Berkeley, Berkeley, CA, USA.

Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro* 23, 2 (March 2003), 22–28. DOI:http://dx.doi.org/10.1109/MM.2003.1196112

Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. ACM, New York, NY, USA, 1–6. DOI:http://dx.doi.org/10.1145/2620728.2620744

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS IX)*. ACM, New York, NY, USA, 117–128. DOI:http://dx.doi.org/10.1145/378993.379232

Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. 1995. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference (TCON'95)*. USENIX Association, Berkeley, CA, USA, 277–288. http://dl.acm.org/citation.cfm?id=1267411.1267434

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages. DOI:http://dx.doi.org/10.1145/1365815.1365816

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1277–1288. http://dl.acm.org/citation.cfm?id=1454159.1454167

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. DOI:http://dx.doi.org/10.1145/1807128.1807152

William Dally. 2012. Lightspeed Datacenter Network. Presentation slides. (2012).

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51 (January 2008), 107–113. Issue 1. DOI:http://dx.doi.org/10.1145/1327452.1327492

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles (SOSP '07)*. ACM, New York, NY, USA, 205–220. DOI:http://dx.doi.org/10.1145/1294261.1294281

David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data (SIGMOD '84)*. ACM, New York, NY, USA, 1–8. DOI:http://dx.doi.org/10.1145/602259.602261

Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi

Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: a distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architec-*

tures, and protocols for computer communication (SIGCOMM '12). ACM, New York, NY, USA, 25–36. DOI:http://dx.doi.org/10.1145/2342356.2342360

Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference*.

H. Garcia-Molina and K. Salem. 1992. Main Memory Database Systems: An Overview. *IEEE Trans. on Knowl. and Data Eng.* 4 (December 1992), 509–516. Issue 6. DOI:http://dx.doi.org/10.1109/69.180602

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*. ACM, New York, NY, USA, 29–43. DOI:http://dx.doi.org/10.1145/945445.945450

C. Gray and D. Cheriton. 1989. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. ACM, New York, NY, USA, 202–210. DOI:http://dx.doi.org/10.1145/74850.74870

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. DOI:http://dx.doi.org/10.1145/78969.78972

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference (USENIX ATC '10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://portal.acm.org/citation.cfm?id=1855840.1855851

Robert Johnson and Jeffrey Rothschild. 2009. Personal communications. (March 24 and August 20, 2009).

Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1 (August 2008), 1496–1499. Issue 2. DOI:http://dx.doi.org/10.1145/1454159.1454211

Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. 1997. Improving the performance of log-structured file systems with adaptive methods. *SIGOPS Oper. Syst. Rev.* 31, 5 (Oct. 1997), 238–251. DOI:http://dx.doi.org/10.1145/269005.266700

Michael David Mitzenmacher. 1996. *The power of two choices in randomized load balancing*. Ph.D. Dissertation. University of California, Berkeley. AAI9723118.

Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 29–41. DOI:http://dx.doi.org/10.1145/2043556.2043560

John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2011. The case for RAMCloud. *Commun. ACM* 54 (July 2011), 121–130. Issue 7. DOI:http://dx.doi.org/10.1145/1965724.1965751

John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. 1988. The Sprite Network Operating System. *Computer* 21 (February 1988), 23–36. Issue 2. DOI:http://dx.doi.org/10.1109/2.16

Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 69–84. DOI:http://dx.doi.org/10.1145/2517349.2522716

Dennis M. Ritchie and Ken Thompson. 1974. The UNIX Time-sharing System. *Commun. ACM* 17, 7 (July 1974), 365–375. DOI:http://dx.doi.org/10.1145/361011.361061

Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.* 10 (February 1992), 26–52. Issue 1. DOI:http://dx.doi.org/10.1145/146941.146943

Stephen M. Rumble. 2014. *Memory and Object Management in RAMCloud*. Ph.D. Dissertation. Stanford University.

Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, Berkeley, CA, USA, 1–16. http://dl.acm.org/citation.cfm?id=2591305.2591307

Margo Seltzer, Keith Bostic, Marshall Kirk Mckusick, and Carl Staelin. 1993. An implementation of a log-structured file system for UNIX. In *Proceedings of the 1993 Winter USENIX Technical Confer-*

ence (USENIX'93). USENIX Association, Berkeley, CA, USA, 307–326. http://dl.acm.org/citation.cfm?id=1267303.1267306

Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. 1995. File system logging versus clustering: a performance comparison. In *Proceedings of the USENIX 1995 Technical Conference (TCON'95)*. USENIX Association, Berkeley, CA, USA, 249–264. http://dl.acm.org/citation.cfm?id=1267411.1267432

Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 731–742. DOI:http://dx.doi.org/10.1145/2213836.2213946

Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.* 11 (February 2003), 17–32. Issue 1. DOI:http://dx.doi.org/10.1109/TNET.2002.808407

Ryan Stutsman, Collin Lee, and John Ousterhout. 2014. Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code. (2014). Stanford University technical report.

Ryan S. Stutsman. 2013. *Durability and Crash Recovery in Distributed In-Memory Storage Systems*. Ph.D. Dissertation. Stanford University.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301