DURABILITY AND CRASH RECOVERY IN DISTRIBUTED IN-MEMORY STORAGE SYSTEMS

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

Ryan Scott Stutsman
November 2013

This dissertation is online at: http://purl.stanford.edu/kk538ch0403

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

John Ousterhout, Primary Adviser

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

David Mazieres

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

Mendel Rosenblum

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

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

Abstract

This dissertation presents fast crash recovery for the RAMCloud distributed in-memory data center storage system. RAMCloud is designed to operate on thousands or tens-of-thousands of machines, and it stores all data in DRAM. Rather than replicating in DRAM for redundancy, it provides inexpensive durability and availability by recovering quickly after server crashes. Overall, its goal is to reconstitute the entire DRAM contents of a server and to restore full performance to the cluster in 1 to 2 seconds after failures. Consequently, RAMCloud provides continuous availability by recovering from failures so quickly that applications never notice failures.

The guiding design principle behind fast recovery is leveraging scale. RAMCloud scatters backup data across thousands of disks, and it harnesses hundreds of servers in parallel to reconstruct lost data. The system uses a decentralized log-structured approach for all of its data, in DRAM as well as on disk; this provides high performance both during normal operation and during recovery. RAMCloud employs randomized techniques at several key points to balance load and to manage the system in a scalable and decentralized fashion.

We have implemented and evaluated fast crash recovery. In an 80-node cluster, RAMCloud recovers 40 GB of data from a failed server in 1.86 seconds. Extrapolations based on measurements suggest that the current implementation approach should scale to recover 200 GB of DRAM from a failed server in about 2 seconds and may scale to recover 1 TB of data or more in the same period with few changes. Our analysis also suggests that 1 second recoveries may be as effective as increased on-disk redundancy at preventing data loss due to independent server failures while being less costly.

Fast crash recovery provides a foundation for fault-tolerance and it simplifies RAMCloud's overall design. In addition to handling single node failures, it is the key mechanism for recovering from multiple simultaneous machine failures and performing complete cluster restarts. RAMCloud servers always recover to a consistent state regardless of the timing of failures, which allows it to provide strong consistency semantics even across failures. In addition to fast recovery itself, this dissertation details how RAMCloud stores data in decentralized logs for durability, how the durability and consistency of data is protected from failures, how log replay is divided across all the nodes of the cluster after failures, and how recovery is used to handle the many failure scenarios that arise in large-scale clusters.

Acknowledgements

I owe a huge debt of gratitude to the hundreds of people who have provided me with encouragement, guidance, opportunities, and support over my time as a graduate student. Among those hundreds, there is no one that could have been more crucial to my success and happiness than my wife, Beth. She has supported my academic endeavor now for nearly a decade, patiently supporting me through slow progress and encouraging me to rebound after failures. I am thankful for her everyday, and I am as excited as I have ever been about our future together.

No one has made me smile more during my time at Stanford than my daughter, Claire. She is energetic, independent, and beautiful. She gives me overwhelming optimism, and nothing cheers me more than seeing her at the end of a long day. Getting to know her has been the greatest joy of my life, and I look forward to seeing her grow in new ways.

At Stanford, there is no one who has impacted my life more than my advisor, John Ousterhout. John is the rare type of person that is so positive, so collected, and so persistent that it simultaneously makes one feel both intimated and invigorated. He taught me how to conduct myself as a scientist and an engineer, but he also taught me how to express myself, how to focus amid confusion, and how to remain positive when challenges stack up. There is no one I admire more, and I will sorely miss scheming with him.

David Mazières has been more encouraging to me than anyone else in my time at Stanford. David persistently encourages me to worry less, to think for myself, and to take joy in what I do. He is the reason I ended up at Stanford, and I will miss his unique personality. Mikhail Atallah, Christian Grothoff, and Krista Grothoff changed the trajectory of my life by introducing me to research as an undergraduate. Their guidance and support opened opportunities that I would have never imagined for myself. Thanks to Mendel Rosenblum for feedback and review of this dissertation as well as sitting on my defense committee. Also, thanks to Christos Kozyrakis and Christine Min Wotipka for sitting on my defense committee.

I couldn't have asked for a better pair of labmates than Stephen Rumble and Diego Ongaro. I have learned more from them in the past 6 years than from anyone else. Working closely together on a such a large project was the highlight of my academic career. I am excited to see where they head in the future. This dissertation wouldn't have been possible without their help and continuous chastisement. I am looking forward to the results of the next generation of high-energy RAMCloud students: Asaf Cidon, Jonathan Ellithorpe, Ankita Kejriwal, Collin Lee, Behnam Montazeri Najafabadi, and Henry Qin. Also, thanks to the many people that

helped build RAMCloud over the past three years: Arjun Gopalan, Ashish Gupta, Nandu Jayakumar, Aravind Narayanan, Christian Tinnefeld, Stephen Yang, and many more. Stephen, you're weird and I'll miss you.

I also owe thanks to several current and former members of the Secure Computer Systems lab at Stanford. In particular, Nickolai Zeldovich, Daniel Giffin, and Michael Walfish all helped show me the ropes early on in my graduate career. Thanks to Andrea Bittau for continuously offering to fly me to Vegas.

Unquestionably, I owe enormous thanks to my parents; they impressed the value of education on me from a young age. I am perpetually thankful for their unconditional love, and I only hope I can be as good a parent for my children as they were for me. My sister, Julie, has been a persistent source of encouragement for me, and I owe her many thanks as well. My friends have also been a source of encouragement to me as I have pursued my degree. Thanks to Nathaniel Dale^{L}, Brandon Greenawalt, and Zachary Tatlock for all your support.

This research was partly performed under an appointment to the U.S. Department of Homeland Security (DHS) Scholarship and Fellowship Program, administered by the Oak Ridge Institute for Science and Education (ORISE) through an interagency agreement between the U.S. Department of Energy (DOE) and DHS. ORISE is managed by Oak Ridge Associated Universities (ORAU) under DOE contract number DE-AC05-06OR23100. All opinions expressed in this dissertation are the author's and do not necessarily reflect the policies and views of DHS, DOE, or ORAU/ORISE. This research was also partly supported by the National Science Foundation under Grant No. 0963859, by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by grants from Facebook, Google, Mellanox, NEC, NetApp, SAP, and Samsung.

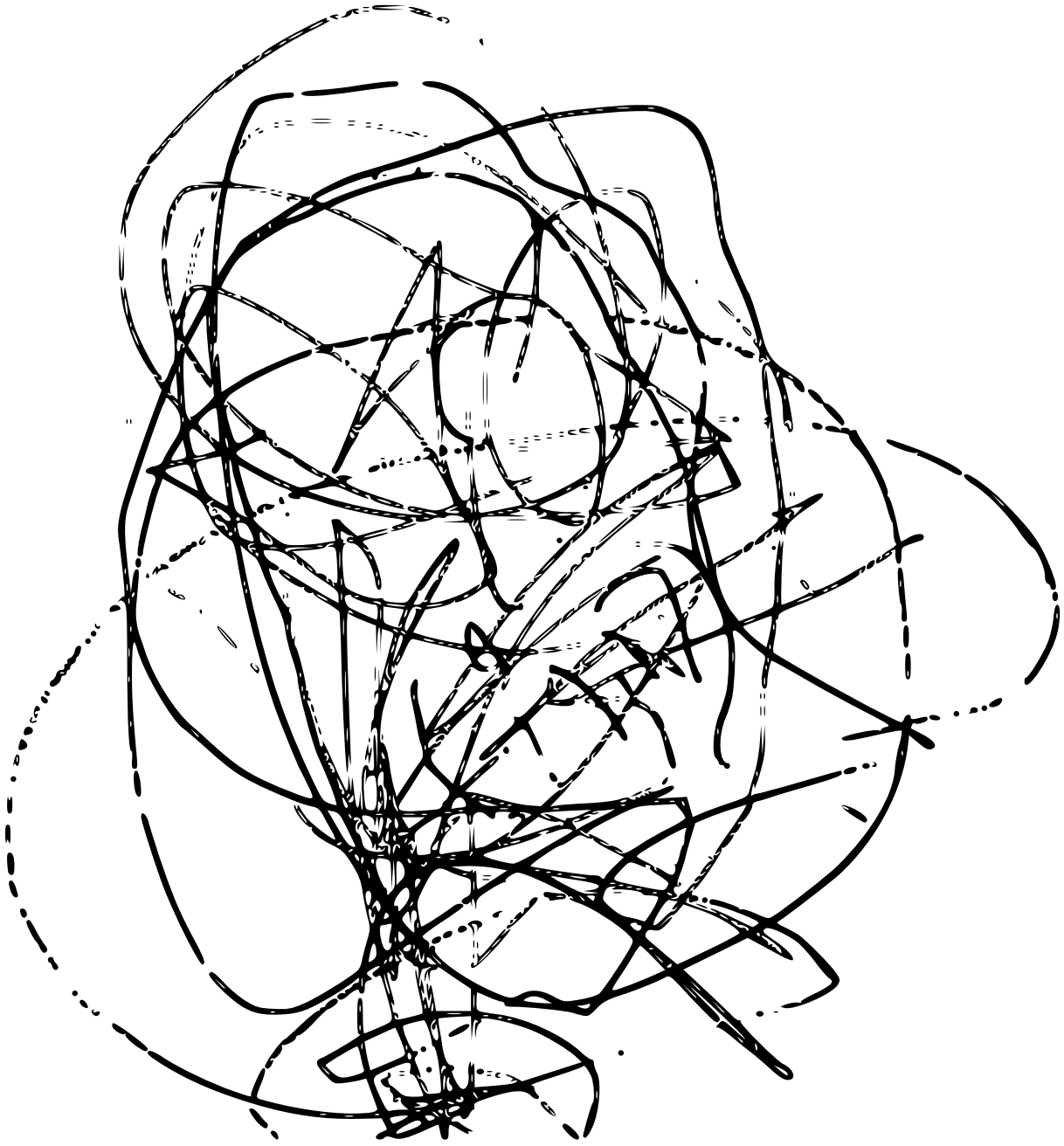


Figure 1: *Spiderweb* by Claire Stutsman.

Contents

Abstract iv

Acknowledgements v

1 Introduction 1

1.1 Dissertation Contributions 5

2 Motivation 6

3 RAMCloud Overview 10

3.1 Basics 10

3.2 Data Model 11

3.3 System Structure 12

3.3.1 The Coordinator 13

3.3.2 Tablets 13

3.4 Durability and Availability 14

3.5 Log-Structured Storage 15

3.6 Recovery 17

3.6.1 Using Scale 18

4 Fault-tolerant Decentralized Logs 21

4.1 Log Operations 23

4.2 Appending Log Data 24

4.3 Scattering Segment Replicas 26

4.4 Log Reassembly During Recovery . . . 32
4.4.1 Finding Log Segment Replicas . . . 33
4.4.2 Detecting Incomplete Logs . . . 33
4.5 Repairing Logs After Failures . . . 36
4.5.1 Recreating Lost Replicas . . . 37
4.5.2 Atomic Replication . . . 38
4.5.3 Lost Head Segment Replicas . . . 38
4.5.4 Preventing Inconsistencies Due to Overloaded Backups . . . 42
4.5.5 Replica Corruption . . . 43
4.6 Reclaiming Replica Storage . . . 43
4.6.1 Discarding Replicas After Master Recovery . . . 44
4.6.2 Redundant Replicas Found on Storage During Startup . . . 45
4.7 Summary . . . 47

5 Master Recovery **49**
5.1 Recovery Overview . . . 51
5.2 Failure Detection . . . 53
5.3 Setup . . . 56
5.3.1 Finding Log Segment Replicas . . . 56
5.3.2 Detecting Incomplete Logs . . . 57
5.3.3 Grouping Tablets Into Partitions . . . 57
5.3.4 Assigning Partitions to Recovery Masters . . . 61
5.4 Replay . . . 62
5.4.1 Partitioning Log Entries on Backups . . . 63
5.4.2 Segment Replay Order . . . 64
5.5 Cleanup . . . 67
5.6 Coordinating Recoveries . . . 67
5.6.1 Multiple Failures . . . 67
5.6.2 Failures During Recovery . . . 68
5.6.3 Cold Start . . . 72
5.7 Zombie Servers . . . 73

5.8 Summary . . . 75

6 Evaluation **77**

6.1 Experimental Configuration . . . 79
6.2 How Large Should Segments Be? . . . 81
6.3 How Large Should Partitions Be? . . . 84
6.4 How Well Does Recovery Scale? . . . 86
6.4.1 Bottlenecks in the Experimental Configuration . . . 88
6.4.2 Recovery Coordination Overhead . . . 96
6.5 The Limits of Fast Crash Recovery . . . 98
6.5.1 Scaling Up . . . 98
6.5.2 Scaling Down . . . 102
6.5.3 What Is the Fastest Possible Recovery? . . . 103
6.6 How Well Does Segment Scattering Work? . . . 104
6.7 Impact of Fast Crash Recovery on Data Loss . . . 106
6.7.1 Vulnerability to Correlated Failures . . . 109
6.8 Summary . . . 109

7 Related Work **112**

7.1 DRAM in Storage Systems . . . 112
7.1.1 Caching . . . 112
7.1.2 Application-Specific Approaches . . . 114
7.1.3 In-memory Databases . . . 115
7.2 Crash Recovery . . . 116
7.2.1 Fast Crash Recovery . . . 116
7.2.2 Striping/Chunk Scattering . . . 117
7.2.3 Partitioned Recovery . . . 117
7.2.4 Reconstruct Versus Reload . . . 118
7.3 Log-structured Storage . . . 119
7.4 Randomized Load Balancing . . . 120

8 Conclusion **121**

8.1 Future Directions for Fast Crash Recovery 122

8.2 Lessons Learned About Research 123

8.3 Final Comments 125

List of Tables

6.1 Experimental cluster configuration 80
6.2 Storage devices used in experiments 82
6.3 Recovery time metrics for an 80 node recovery 89
6.4 Network and storage statistics for an 80 node recovery 91

List of Figures

1 Spiderweb by Claire Stutsman vii

3.1 Cluster architecture 12
3.2 Buffered logging 16
3.3 Recovery bottlenecks 19

4.1 Basic log replication 27
4.2 Segment replica scattering 28
4.3 Transitioning between log digests 35
4.4 Neutralizing lost replicas after backup failures 41

5.1 Recovery masters replay objects for one partition 51
5.2 Partitioning tablets for recovery 60
5.3 Recovery parallelism and pipeline stages 62

6.1 Nodes and processes used in the experimental configuration 81
6.2 Bandwidth of storage devices used in experiments 83
6.3 Recovery time for partitions of varying object sizes 85
6.4 Recovery time as amount recovered and cluster size scales 86
6.5 Recovery time as cluster and size of lost data scales using 300 MB partitions 87
6.6 Recovery time distribution 90
6.7 Memory bandwidth for Xeon X3470 with 800 MHz DDR3 92
6.8 Memory bandwidth utilization during recovery 93
6.9 Recovery time with masters and backups on separate nodes 95
6.10 Management overhead as recovery size scales 97

6.11 Projection of maximum server size recoverable in 1 to 2 seconds 99
6.12 Recovery time distribution for various replica placement strategies 105
6.13 Impact of fast crash recovery on data loss 107

Chapter 1

Introduction

The role of DRAM in storage systems has been increasing rapidly in recent years, driven by the needs of large-scale Web applications. These applications access data intensively, and, due to the disparity of disk performance and capacity, they cannot be satisfied by disks alone. As a result, applications are keeping more and more of their data in DRAM. For example, large-scale caching systems such as memcached [37] have seen wide deployment in the past few years. In 2009, Facebook used a total of 150 TB of DRAM in memcached and other caches for a database containing 200 TB of disk storage [27]. Similarly, the major web search engines now keep their search indexes entirely in DRAM.

Although DRAM's role is increasing in large-scale data center applications, DRAM still tends to be used in limited or specialized ways. In most cases DRAM is just a cache for some other storage system, such as a database; in other cases (such as search indexes) DRAM is managed in an application-specific fashion. Both approaches contrast with disk storage, which is accessed with uniform and general-purpose interfaces (for example, filesystems and databases) that allow applications to use it easily.

Two main problems complicate using distributed caches to take advantage of DRAM. First, applications must be augmented to manage consistency between caches and the backing database, and the frequently resulting inconsistencies complicate the work of the developer. Second, because of the gap in performance between the backing database and the caches, even just a small fraction of cache misses dramatically reduces overall performance. Ultimately, this makes it difficult for the developer to predict the performance they

can expect from their storage.

Application-specific uses of DRAM, such as in-memory indexes, are problematic because they require hand-tailoring of applications. Such specialized DRAM use in applications can be effective, but it requires tight coupling between the application and the storage. Developers must repeat the effort for each new application; they are left designing complex and specialized consistency, availability, and fault-tolerance systems into their application.

RAMCloud solves these problems by providing a general-purpose storage system that makes it easy for developers to harness the full performance potential of large-scale DRAM storage. RAMCloud keeps all data in DRAM all the time, so there are no cache misses. It is durable and available, so developers need not manage a separate backing store. It is designed to scale up to ten thousand servers and to store petabytes of data while providing uniform low-latency access. RAMCloud achieves 5 µs round-trip times for small read operations, which is a 50-5000× improvement over access times of state-of-the-art data center storage systems. (Today, in 2013, data center storage systems typically have access times ranging from a few hundred microseconds for volatile caches up to 100 milliseconds or more for replicated disk-based systems.) RAMCloud's overall goal is to enable a new class of data center applications. These applications will be capable of manipulating massive datasets in a way that has previously only been possible for small datasets in the local memory of a single machine.

This dissertation describes and evaluates RAMCloud's scalable fast crash recovery which is at the heart of how RAMCloud provides durability and availability while retaining high performance and low cost. While durability, availability, and fault-tolerance make RAMCloud practical and simple for developers to use, RAMCloud's reliance on DRAM prevents it from using the same solutions that typical data center storage systems use. Specifically, four properties of DRAM guide the design of RAMCloud and motivate its use of fast recovery for fault-tolerance:

- **Volatility:** DRAM is not durable; data must be replicated on stable storage or it risks loss on power failure.

- **Performance:** DRAM is low-latency; in order to expose the full performance benefits to applications, RAMCloud's approach to durability must not impact the performance of normal operations. Any performance overheads jeopardize RAMCloud's potential to support new classes of data-intensive applications.

- **Cost:** DRAM is expensive per bit both in terms of fixed cost and operational cost. This makes approaches that replicate data in DRAM prohibitive. For example, triplicating data in DRAM would triple the cost and energy consumption of the cluster (and would still lose data on cluster-wide power failure).

- **Low Per-Node Capacity:** Typical data center servers can be cost-effectively equipped with 64 to 256 GB of DRAM today. The large-scale datasets typical of modern web applications require the DRAM capacity of many machines, so RAMCloud must scale to support thousands of storage servers.

  However, scaling across thousands of machines works against durability since, in a large cluster, server failures happen frequently. RAMCloud must compensate for this and ensure data is safe even when server failures are frequent.

RAMCloud's approach is to keep only a single copy of data in DRAM; redundant copies are kept on disk or flash, which are cheaper and more durable than DRAM. However, this means that a server crash will leave some of the system's data unavailable until it can be reconstructed from secondary storage. Scalable fast crash recovery solves this problem by reconstructing the entire contents of a lost server's memory (64 to 256 GB) from disk and resuming full service in 1 to 2 seconds. The assumption is that this is fast enough to be considered "continuous availability" for most applications.

This dissertation describes and evaluates scalable fast crash recovery. Additionally, it details the other problems that arise in providing fault tolerance in a large-scale distributed system and describes solutions and their implementation in RAMCloud. This includes retaining strong consistency (linearizability) despite faults, handling multiple simultaneous failures, handling massive failures including completely restarting a RAMCloud cluster, and cleaning up storage and resources across the cluster after servers crash.

There are several key aspects to the RAMCloud recovery architecture:

- **Harnessing scale:** Each server scatters its backup data across all of the other servers, allowing thousands of disks to participate in recovery. Simultaneously, hundreds of *recovery master* servers work together in a highly data parallel and tightly pipelined effort to rapidly recover data into DRAM to restore availability when servers fail.

- **Load balancing:** For crash recovery to take full advantage of the large aggregate resources of the cluster, it must avoid bottlenecks by spreading work evenly across thousands of CPUs, NICs, and disks. Recovery is time sensitive and involves many machines; without care, slow servers ("stragglers") can easily drag out recovery time, which manifests itself as unavailability.

- **Decentralization:** In order to scale to support clusters of thousands of machines, servers must operate independently both during normal operation and during recovery. Each server minimizes its requests to centralized services by having its own self-contained and self-describing log.

- **Randomized techniques:** Careful use of randomization is critical for RAMCloud's load balancing and decentralization. Servers rely on randomization at key points to enable decentralized decision making and to achieve precise balancing of work without the need for explicit coordination.

- **Log-structured storage:** RAMCloud uses techniques similar to those from log-structured file systems [42], not just for information on disk but also for information in DRAM. Its log-structured approach provides high performance by safely buffering updates and eliminating disks from the fast path while simplifying many issues related to crash recovery.

We have implemented the RAMCloud architecture in a working system and evaluated its crash recovery properties. Our 80-node cluster recovers in 1.86 seconds from the failure of a server with 40 GB of data, and the approach scales so that larger clusters can recover larger memory sizes in less time.

1.1 Dissertation Contributions

The main contributions of this dissertation are:

- The design and implementation of a large-scale DRAM-based data center storage system as reliable as disk-based storage systems with minimal performance and cost penalties. This is the only published approach to maintaining durability and availability for distributed DRAM-based storage that optimizes for both cost-effectiveness and low-latency operation. Overall, it achieves the reliability of today's disk-based storage systems for the cost of today's volatile distributed DRAM caches with performance 50-5000× greater than either.

- The design, implementation, and evaluation of RAMCloud's scalable fast crash recovery system which

  - makes DRAM-based storage cost-effective by eliminating the need for redundant data in DRAM while retaining high availability;
  - and improves in recovery time as cluster size scales by harnessing the resources of thousands of servers.

- The design, implementation, and evaluation of a decentralized log which

  - efficiently creates disk-based backups of data stored in DRAM while imposing no overhead for read operations and eliminating disk operations for writes;
  - scales to tens of thousands of machines through aggressive decentralization;
  - and provides strong consistency semantics while remaining durable in spite of failures.

Overall, RAMCloud's decentralized log and scalable fast crash recovery make it easy for applications to take advantage of large scale DRAM-based storage by making the system failure transparent and retaining its low latency access times.

Chapter 2

Motivation

RAMCloud's use of DRAM and its focus on large-scale deployment shape how it provides fault tolerance, which centers on fast crash recovery. This chapter describes why traditional replicated approaches to fault-tolerance are impractical in RAMCloud, primarily due to the high cost-per-bit of DRAM. This motivates fast crash recovery as an alternative to traditional fault-tolerance approaches. Overall, fast crash recovery provides continuous availability while allowing the use of inexpensive disks for redundancy and avoiding disk-limited normal-case performance.

RAMCloud is a data center storage system where every byte of data is present in DRAM at all times. RAMCloud aggregates the DRAM of thousands of servers into a single coherent storage system. RAMCloud's overall goal is to enable a new class of data center applications capable of manipulating massive datasets in a way that has only been possible for small datasets in the local memory of a single machine in the past. For more details on the motivation for RAMCloud and some of its architectural choices see [40].

Fault-tolerance is a key attribute of RAMCloud. Though RAMCloud stores all data in volatile DRAM, it is designed to be as durable and available as disk-based data center storage systems. Fault-tolerance is essential to enable developers to easily incorporate large-scale DRAM storage into applications.

Redundancy is fundamental to fault-tolerance. Data center storage systems rely on redundant copies of data to maintain availability when servers fail. Given that some replication is required for availability, systems have a choice on how to implement it. In today's

disk-based data center storage systems, the most common approach is to replicate all data and metadata on stable storage in the same form used to service requests during normal operation. When a server fails, another server with a replica of the data can continue to service requests as usual.

Does this approach work for RAMCloud? How should a large-scale DRAM-based storage system provide fault-tolerance? Today, there are no general-purpose, fault-tolerant, large-scale, DRAM-based storage systems, perhaps because DRAM presents special challenges. It is volatile and loses data on power failure. It is expensive, so redundancy in DRAM is impractically expensive. It is high-performance; any performance penalty in the storage system will be immediately reflected in client application access times. Finally, it is also less dense; more machines are needed to house large datasets, so it indirectly increases the frequency of machine failures in a cluster.

RAMCloud could use the same approach as other data center storage systems by storing each object in the DRAM of multiple servers; however, with a typical replication factor of three, this approach would triple both the cost and energy usage of the system. Even without triplication, in RAMCloud the DRAM of each machine contributes more than half of the hardware cost ($\sim$\$10/GB compared to $\sim$\$0.10/GB for disk) and energy consumption ($\sim$1 W/GB [35] compared to $\sim$5 mW/GB for disk). Thus, using DRAM for replication is prohibitively expensive. As a result, RAMCloud keeps only a single copy of each object in DRAM, with redundant copies on secondary storage such as disk or flash. Since the cost and energy usage of DRAM for primary copies dominates compared to the redundant copies on disk or flash, replication is nearly free. For example, the three-year total cost of ownership of 64 GB of DRAM is about \$800 compared to \$22 for the 192 GB of disk storage needed to backup the DRAM in triplicate.

However, the use of disks raises two issues. First, if data were to be written synchronously to disks for durability before responding to client requests, then the normal case performance of the system would be limited by the high latency of disk writes. Today's networking hardware allows round trips in a few microseconds, whereas a single disk access takes milliseconds; synchronous disk operations would slow RAMCloud by a factor of 1,000. The primary value of RAMCloud comes from its high performance, so disk-bound performance would severely limit its usefulness. Second, this approach could result

in long periods of unavailability or poor performance after server crashes, since data will have to be reconstructed from secondary storage. Data center applications often require continuous availability, so any noticeable delay due to failures is unacceptable.

This dissertation describes how RAMCloud uses scalable fast crash recovery and decentralized buffered logs to solve these two problems. Recovery minimizes unavailability due to server failures; buffered logs record data durably without the need for synchronous disk writes. During normal operation, each server scatters backup data across all of the nodes of the cluster, which temporarily store data in small non-volatile buffers. Each time a buffer is filled, it is written disk. When a server fails in RAMCloud, nodes cooperate to rapidly reconstitute the DRAM contents of the failed server. Since data for a failed server is stored on all of the servers of the cluster, it can be reloaded quickly by reading from all of the disks in parallel. Furthermore, the work of processing data as it is loaded from disks is split across hundreds of machines. By harnessing all of the disks, NICs and CPUs of a large cluster RAMCloud recovers from failures in 1 to 2 seconds and provides an illusion of continuous availability.

Fast crash recovery together with buffered logging solves each of the four key issues with using DRAM as a large scale storage medium:

- **Volatility:** Servers scatter replicas of data across the disks of other nodes in the cluster; data is protected against power failure, since it is always stored on disk or in a non-volatile buffer.

- **Performance:** Since writes are buffered in non-volatile storage and written to disk in bulk, access latency is determined by network delay rather than disk latency even for update operations.

- **Cost:** Fast crash recovery makes large-scale DRAM more cost-effective in two ways; it eliminates the need for expensive redundancy in DRAM, and it retains the high performance for updates that makes DRAM storage so valuable.

- **Reliability at Scale:** DRAM may require the use of thousands or tens of thousands of machines for large datasets, which increases the rate of failures RAMCloud must tolerate. Fast recovery more than compensates for the increased risk due to more

frequent failures. By restoring full replication quickly after a node failure, it reduces vulnerability to data loss. Section 6.7 analyzes the impact of recovery time on reliability and shows that improving recovery time reduces the probability of data loss nearly as significantly as increasing redundancy.

Overall, fast recovery is the key to making RAMCloud fault-tolerant and cost-effective while retaining access times determined by network latency rather than disk access times.

The next chapter gives an overview of RAMCloud and recovery. The remaining chapters describe how RAMCloud stores redundant data on disk without limiting performance and how clusters use that on-disk data to reconstitute the DRAM of failed servers to create an illusion of continuous availability.

Chapter 3

RAMCloud Overview

This chapter describes the key ideas behind RAMCloud, the high level components of the system, and its basic operation. Additionally, because crash recovery and normal request processing are tightly intertwined, this chapter also gives an overview of how single machine failures are handled and how that informs and motivates many design choices in RAMCloud.

3.1 Basics

RAMCloud is a storage system where every byte of data is present in DRAM at all times. The hardware for RAMCloud consists of hundreds or thousands of off-the-shelf servers in a single data center, each with as much DRAM as is cost-effective (64 to 256 GB today). RAMCloud aggregates the DRAM of all these servers into a single coherent storage system. It uses backup copies on disk or flash to make its storage durable, but the performance of the system is determined by DRAM, not disk.

The RAMCloud architecture combines two interesting properties: low latency and large scale. First, RAMCloud is designed to provide the lowest possible latency for remote access by applications in the same data center. Most of today's data center networks cannot meet RAMCloud's latency goals, but this is changing as the deployment of 10 gigabit Ethernet has started in data centers. The current implementation achieves end-to-end times of 5 µs for reading small objects ($\sim$100 to 1000 B) using inexpensive Infiniband networking

hardware, and initial tests indicate similar times are achievable with commodity 10 gigabit equipment. This represents an improvement of 50-5000× over existing data center-scale storage systems. Each RAMCloud storage server can handle about 650 thousand small read requests per second.

The second important property of RAMCloud is scale; a single RAMCloud cluster must support up to ten thousand servers in order to provide a coherent source of data for large applications (up to about 2.5 PB with today's machines). Scale creates several challenges, such as the likelihood of frequent component failures and the need for distributed decision-making to avoid bottlenecks. However, scale also creates opportunities, such as the ability to enlist large numbers of resources on problems like fast crash recovery.

RAMCloud's overall goal is to enable a new class of data center applications. These applications will be capable of manipulating massive datasets in a way that has only been possible for small datasets in the local memory of a single machine heretofore. For more details on the motivation for RAMCloud see [40].

3.2 Data Model

The current data model in RAMCloud is a simple key-value store. RAMCloud supports any number of tables, each of which may contain any number of objects. An object consists of

- a variable-length byte array that serves as its key (up to 64 KB) that is provided by the client,
- a variable length byte array of opaque data (up to 1 MB) that is provided by the client and left uninterpreted by RAMCloud,
- and a 64-bit system-managed version number that strictly increases each time the value for a particular key is set, deleted, or overwritten.

RAMCloud provides a simple set of operations for creating and deleting tables and for reading, writing, and deleting objects within a table. Objects are addressed using their keys and are read and written in their entirety and atomically. There is no built-in support for

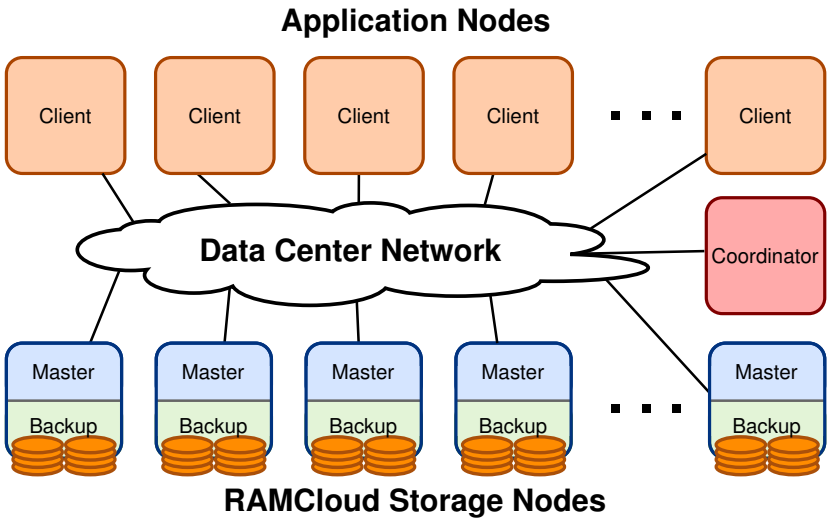


Figure 3.1: RAMCloud cluster architecture. Each storage server contains a master and a backup. A central coordinator manages the server pool and tablet configuration. Client applications run on separate machines and access RAMCloud using a client library that makes remote procedure calls.

atomic updates across multiple objects, but RAMCloud does provide a conditional update ("replace the contents of object O in table T only if its current version number is V"), which can be used to implement more complex transactions in application software. Clients can compare version numbers between objects retrieved from the same table and having the same key to determine which was more recently written.

3.3 System Structure

Figure 3.1 shows the composition of a RAMCloud cluster. Each of the storage servers has two components: a *master*, which manages RAMCloud objects in its DRAM and services client requests, and a *backup*, which stores redundant copies of objects from other masters using its disk or flash memory.

Client applications use a RAMCloud client library that presents the cluster as a single storage system. The library transparently fetches (and caches) cluster configuration information from the RAMCloud cluster coordinator (described in the next section), which allows it to direct application requests to the appropriate RAMCloud machines.

3.3.1 The Coordinator

Each RAMCloud cluster also contains one distinguished server called the *coordinator*. The coordinator manages cluster membership, initiates recoveries when machines fail, and maintains the mapping of keys to masters. The use of a centralized coordinator has a substantial impact on the design of the rest of the system. Centralization simplifies many operations, but, in order for the system to scale to thousands of machines, use of the coordinator must be limited. By design, the coordinator is involved in few client requests.

To maintain high availability, RAMCloud must take care to prevent the coordinator from being a single point of failure. The coordinator's core data structures are stored in ZooKeeper [25], a replicated, highly-available configuration storage system. At any one time, a RAMCloud cluster has a single operating coordinator with multiple "warm" passive coordinators that are fed the same state state changes as the active coordinator. One of these passive coordinators takes over in the case the active coordinator fails. ZooKeeper mediates the selection of the replacement coordinator using its consensus algorithm. The details of ZooKeeper and coordinator failover are outside the scope of this dissertation.

3.3.2 Tablets

Clients create and work with tables, but the coordinator assigns objects to masters in units of *tablets*. Tablets allow a single table to be distributed across multiple masters. Keys for objects in a table are hashed into a 64-bit space. The hash space for each table is divided into one or more tablets. Each tablet is a contiguous subrange of the key hash space for its table. The coordinator may assign a single master any number of tablets from any number of tables.

New tables are initially created with a single empty tablet assigned to a single server. Clients can request that a tablet be split at a particular key hash. The system also supports a basic form of tablet migration between masters. Tablet splitting together with migration allow large tables to be broken up and distributed across multiple masters. An automatic load balancer is planned for future work; it will migrate tablets to optimize for locality by minimizing the number of masters a particular application must access.

The coordinator stores the mapping between tablets and masters. The RAMCloud client

library maintains a cache of this information, fetching the mappings for each table the first time it is accessed. Once a client's configuration cache has been populated, a client can issue storage requests directly to the relevant master without involving the coordinator. If a client's cached configuration information becomes stale because a tablet has moved, the client library discovers this when it makes a request to a master that no longer contains the tablet, at which point it flushes the stale data from its cache and fetches up-to-date information from the coordinator. As will be seen in Section 3.6, recovery from server failures causes tablets to relocate across the cluster; clients use the same tablet location mechanism to find the new location for data after a server crash.

3.4 Durability and Availability

The internal structure of a RAMCloud storage server is determined primarily by the need to provide durability and availability. In the absence of these requirements, a master would consist of little more than a hash table that maps from ⟨*table identifier, object key*⟩ pairs to objects in DRAM. The main challenge is providing durability and availability without sacrificing performance or greatly increasing system cost.

One possible approach to availability is to replicate each object in the DRAM of several servers. However, DRAM-based replicas would still be vulnerable in the event of power failures. Machines with 64 to 256 GB of DRAM are common in data centers, and data center battery backups [44] do not last long enough to flush the entire DRAM contents of a server to disk. Furthermore, with a typical replication factor of three, this approach would triple both the cost and energy usage of the system; each server is already fully loaded, so adding more memory would also require adding more servers and networking.

Instead, RAMCloud keeps only a single copy of each object in DRAM, with redundant copies on secondary storage such as disk or flash. In a RAMCloud cluster, the DRAM of each machine contributes more than half of the hardware cost and energy consumption. Since the cost and energy usage of DRAM for primary copies dominates compared to the redundant copies on disk or flash, replication is nearly free. However, the use of disks raises two issues. First, if data must be written synchronously to disks for durability before responding to client requests, then the normal case performance of the system will

be limited by the high latency of disk writes. Second, this approach could result in long periods of unavailability or poor performance after server crashes, since data will have to be reconstructed from secondary storage. Section 3.5 describes how RAMCloud solves the performance problem. Section 3.6 gives an overview of recovery, which restores access to data quickly after crashes so that clients never notice failures.

3.5 Log-Structured Storage

RAMCloud manages object data using a logging approach. This was originally motivated by the desire to transfer backup data to disk or flash as efficiently as possible, but it also provides an efficient memory management mechanism, enables fast recovery, and has a simple implementation. The data for each master is organized as a log as shown in Figure 3.2. When a master receives a write request, it appends the new object to its in-memory log and forwards that log entry to several backup servers. The backups buffer this information in memory and return immediately to the master without writing to disk or flash. The master completes its request and returns to the client once all of the backups have acknowledged receipt of the log data. When a backup's buffer fills, it writes the accumulated log data to disk or flash in a single large transfer, then deletes the buffered data from memory.

Backups must ensure that buffered log data is as durable as data on disk or flash; information must not be lost in a power failure. As mentioned earlier, many existing data centers provide per-server or rack-level battery backups that can extend power briefly after outages. For example, one specification from Facebook [44] provides for about 45 seconds of battery power in the case of a power outage. Backups limit the amount of data they buffer and regularly flush data to stable storage to ensure they can safely write out any partial, dirty buffers in the case of an outage. (The amount backups buffer is configurable; to ensure safety, it should based on a conservative estimate of local storage performance and battery life.) An alternative to batteries is to use new DIMM memory modules that incorporate flash memory and a super-capacitor that provides enough power for the DIMM to write its contents to flash after a power outage [1]; each backup could use one of these modules to hold all of its buffered log data. Enterprise disk controllers with persistent cache memory [34] are widely available and could also be used.

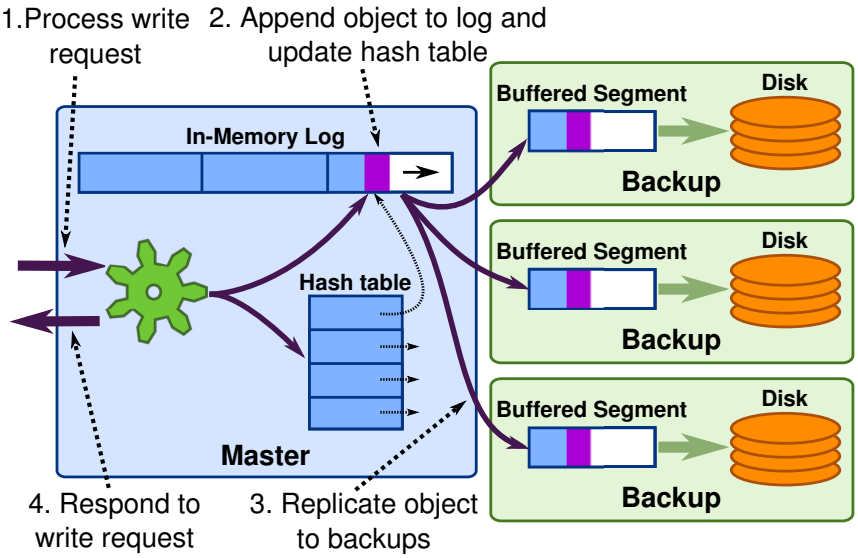


Figure 3.2: When a master receives a write request, it updates its in-memory log and forwards the new data to several backups that buffer the data in their memory. The data is eventually written to disk or flash in large batches. Backups must use an auxiliary power source to ensure that buffers can be written to stable storage during power failures.

RAMCloud manages its logs using techniques similar to those in log-structured file systems [42]. Each master's log is divided into 8 MB *segments*. The master keeps a count of unused space within each segment, which accumulates as objects are deleted or overwritten. It reclaims wasted space by occasionally invoking a *log cleaner* [42, 43] ; the cleaner selects one or more segments to clean, reads the live records from the segments and rewrites them at the head of the log, then deletes the cleaned segments along with their backup copies. Segments are also the unit of buffering and I/O on backups; the large segment size enables efficient I/O for both disk and flash.

RAMCloud uses a log-structured approach not only for backup storage, but also for information in DRAM. The memory of a master is structured as a collection of log segments which map directly to the segment *replicas* stored on backups. This allows masters to manage both their in-memory data and their backup data using a single mechanism. The log provides an efficient memory management mechanism, with the cleaner implementing a form of generational garbage collection. The details of memory management and log cleaning are outside the scope of this dissertation; see [43] for more on this topic.

In order to support random access to objects in memory, each master keeps a hash table that maps from a ⟨*table identifier, object identifier*⟩ pair to the current version of an object in a segment. The hash table is used both to look up objects during storage operations and to determine whether a particular object version is the current one during cleaning. For example, if there is no hash table entry for a particular object in a segment being cleaned, it means the object has been deleted.

The buffered logging approach allows writes to complete without waiting for disk operations, but it limits overall system throughput to the bandwidth of the backup storage. For example, each RAMCloud server can handle about 300 thousand 100-byte writes/second (versus 650 thousand reads/second) assuming 2 disks per server, 100 MB/s write bandwidth for each disk, 3 disk replicas of each object, and a 100% bandwidth overhead for log cleaning. Flash or additional disks can be used to boost write throughput.

3.6 Recovery

When a RAMCloud storage server crashes, the objects that had been present in its DRAM must be recovered by replaying the log the master left behind. This requires reading log segment replicas from backup storage back into DRAM, processing the records in those replicas to identify the current version of each live object, and reconstructing the hash table used for storage operations. The crashed master's data will be unavailable until the objects from the crashed server's log are read into DRAM and the hash table has been reconstructed.

Fortunately, if the period of unavailability can be made very short, so that it is no longer than other delays that are common in normal operation, and if crashes happen infrequently, then crash recovery may be made unnoticeable to application users. If recovery from server failures were instantaneous, applications and their users would be unable to determine whether storage server failures had occurred at all. Though instantaneous recovery is impractical, for certain classes of applications RAMCloud may be able to recover from failures quickly enough to avoid perception by users. Many data center applications are exposed to users as web applications accessed though a browser over the Internet. Because of the high variance in delivering content over the Internet, web application interfaces and

their users expect to occasionally experience delays of a few seconds. As a result, RAMCloud is designed to achieve 1 to 2 second recovery, which is fast enough to constitute "continuous availability" for many applications. Our goal is to build a system that achieves this speed for servers with 64 to 256 GB of memory and scales to recover larger servers as cluster size scales.

3.6.1 Using Scale

The key to fast recovery in RAMCloud is to take advantage of the massive resources of the cluster. This section introduces RAMCloud's overall approach for harnessing scale during recovery; Chapter 5 describes the process of recovery in detail.

As a baseline, Figure 3.3a shows a simple mirrored approach where each master chooses 3 backups and stores copies of all its log segments on each backup. Unfortunately, this creates a bottleneck for recovery because the master's data must be read from only a few disks. In the configuration of Figure 3.3a with 3 disks, it would take about 3.5 minutes to read 64 GB of data.

RAMCloud works around the disk bottleneck by using more disks during recovery. Each master scatters its log data across all of the backups in the cluster as shown in Figure 3.3b; each segment is replicated on a different set of backups. During recovery, these scattered replicas can be read simultaneously; with 1,000 disks, 64 GB of data can be read into memory in less than one second.

Once the segment replicas have been read from disk into backups' memories, the log must be replayed to find the most recent version for each object. No backup can tell in isolation whether a particular object found in a particular replica was the most recent version of that object, because a different replica on another backup may contain a more recent version of the object. One approach is to send all of the log segment replicas for a crashed server to a single *recovery master* and replay the log on that master, as in Figure 3.3b. Unfortunately, the recovery master is a bottleneck in this approach. With a 10 Gbps network interface, it will take about one minute for a recovery master to receive 64 GB of data, and the master's CPU can also become a bottleneck due to the cache misses caused by rapidly inserting objects into its hash table.

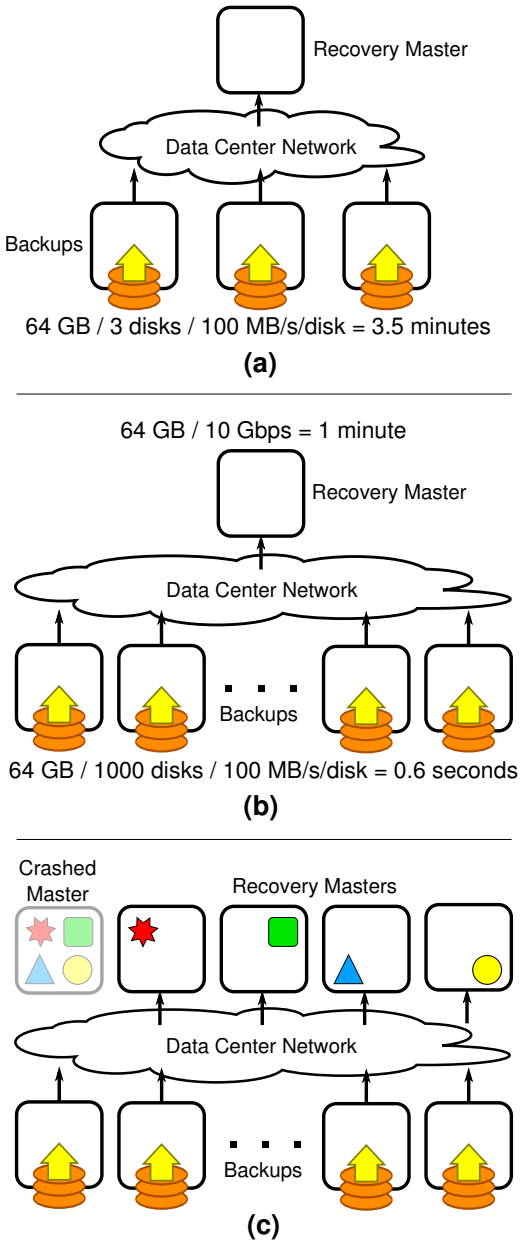


Figure 3.3: **(a)** Disk bandwidth is a recovery bottleneck if each master's data is mirrored on a small number of backup machines. **(b)** Scattering log segments across many backups removes the disk bottleneck, but recovering all data on one recovery master is limited by the network interface and CPU of that machine. **(c)** Fast recovery is achieved by partitioning the data of the crashed master and recovering each partition on a separate recovery master.

To eliminate the recovery master as the bottleneck, RAMCloud uses multiple recovery masters as shown in Figure 3.3c. During recovery RAMCloud divides up the tablets that belonged to the crashed master into *partitions* of roughly equal size. Each partition of the crashed master is assigned to a different recovery master, which fetches the log records for objects in its partition from backups. The recovery master then incorporates those objects into its own log and hash table. With 100 recovery masters operating in parallel, 64 GB of data can be transferred over a 10 Gbps network in less than 1 second. As will be shown in Chapter 6, this is also enough time for each recovery master's CPU to process the incoming data.

Thus, the overall approach to recovery in RAMCloud is to combine the disk bandwidth, network bandwidth, and CPU cycles of many of machines in a highly parallel and tightly pipelined effort to bring the data from the crashed server back online as quickly as possible.

The next chapter details how the logs that underlie both recovery and normal operation are created and maintained by masters. The log replication module on each master handles all aspects of data durability even in the face of failures; it recreates lost replicas when a backup with segment replicas fails, and it ensures damaged or inconsistent replicas cannot affect the correctness of recovery. By transparently handling many of problems that can arise with replicated data, the log replication module simplifies the complex task of recovery. The coordinator, the recovery masters, and all the backups of the cluster are left to focus solely on restoring access to the data of crashed servers as quickly as possible. Chapter 5 details how recovery divides work among thousands of machines and how it coordinates their resources to recover in 1 to 2 seconds.

Chapter 4

Fault-tolerant Decentralized Logs

RAMCloud's decentralized logs are at the core of all of its operations. This chapter describes how these logs are created and maintained in detail, while Chapter 5 explains the specifics of using these logs to rapidly recover the contents of failed servers.

First, this chapter describes the normal operation of logs, including how masters replicate their log data efficiently for durability, how logs are segmented and scattered across the cluster for fast recovery, and how logs are reassembled during recovery. Afterward, the chapter details the ways failures can affect logs and how RAMCloud compensates for them. The chapter concludes with details of how disk storage space is reclaimed.

As described in Section 3.5, each storage server's master service stores all object data in its own fault tolerant log that is scattered across the cluster. Each master has a *replica manager* whose primary responsibility is to construct and maintain this log, so that it can be used to safely recover the master's data after server failure with minimal delay. To the extent possible, the replica manager must ensure that the master's log is available and safe for replay at all times, since failures are unexpected by nature. Master recovery is complex and highly-optimized, and it must work correctly even under the most elaborate failure scenarios. By shifting the burden of producing and maintaining a safe, consistent, and available log to the master's log replication module, recovery is freed to focus solely on fast replay of the log.

RAMCloud's decentralized logs have several important properties:

- **Scalability:** A master never needs to consult the coordinator in order to append to

its log. This prevents bottlenecks on centralized metadata services (for example, the coordinator). For example, some similar data center filesystems also scatter data blocks across many machines, but they centrally track where each of the replicas for each block are located in the cluster (for example, Windows Azure Storage [6]). RAMCloud servers keep no centralized record of log contents. In part by avoiding centralization, RAMCloud is designed to scale to clusters of tens of thousands of machines.

- **High bandwidth:** Each master scatters log data across the entire cluster, which provides high bandwidth both for writing and for reading during recovery. This is key in meeting RAMCloud's 1 to 2 second crash recovery goal.

- **Cost effectiveness:** Because of the high cost per bit of DRAM, redundant copies of log data are only stored on disk or flash. This provides durability for the data stored in memory on masters without using more DRAM than is absolutely required for the lowest latency performance.

- **Low-latency operation:** For applications to gain the full performance benefit from DRAM, log operations must provide durability at the speed of the network (in microseconds) rather than the speed of disk (in milliseconds). Backups judiciously buffer updates in DRAM and rely on limited battery backups to safely flush data to disk on power failures. Because data can be written to disk off the fast path, client writes can be serviced quickly.

- **Durability and fault-tolerance:** Masters' replica managers react to server failure notifications from the coordinator and automatically take action to restore full replication to portions of the log affected by failure. This limits the complexity of preventing data loss to each master's replica manager.

- **Consistency and safety:** Each master's log provides strong consistency even while allowing concurrent updates to facilitate high throughput. In fact, logs support both concurrent log append operations and concurrent replication requests to backups. Each log guarantees stale data will never be seen when it is read for recovery, and any attempt to use a log that has experienced data loss will always be correctly detected.

These strong guarantees simplify the logic for appending to logs and for using log data during recovery.

The rest of this chapter first describes how RAMCloud manages its decentralized logs during normal operation and then details how the logs provide durability and consistency even as servers fail.

4.1 Log Operations

Each server has exactly one log, and the lifetime of every log is tied directly to the lifetime of its server. When a server enlists with the coordinator, it is assigned a server id that doubles as the server's log id. This id is used to tag and identify data written by a particular server regardless of where it is stored in the cluster. A server's log data is automatically deleted when the coordinator removes the server from the cluster, which typically happens after the failure and recovery of that server.

Each master organizes its DRAM as a log, and the master's replica manager works to replicate this in-memory log to stable storage on other machines. This ensures that the objects will be recovered when the master fails. From the perspective of the master replicating client data, the replica manager has a simple interface consisting of just four methods. Each is summarized below and explained in more detail later.

- **`allocateHead(`***`segment`***`)`** registers an empty 8 MB region of the master's DRAM called a *segment* with the replica manager. After a segment is registered, the replica manager incrementally replicates data as it is added to the segment by sending it to other servers for backup (the replica manager is notified of added data via the `sync` call, described below). The replica manager also takes responsibility for recreating replicas of any segment that are lost due to server failures. Policy on how a master allocates DRAM for segments is left outside the replica manager, so allocation can be tuned without changes to replication code.

- **`sync(`***`segment, offset`***`)`** waits for an object appended to the log to be safely replicated to backup servers. It blocks when called until a segment has been replicated safely up through a specified byte offset. Masters call this after adding data

to a segment to ensure that the data is durable before acknowledging the associated operation to the client.

- **`free(`***`segment`***`)`** reclaims log space. This call disassociates a segment's DRAM buffer from the replica manager, which allows it to be reused. It also sends a notification to backups to inform them that it is safe to reclaim the associated space on their stable storage. Section 4.6 details how space is reclaimed both in the master's DRAM and on the stable storage of backups.

- **`findReplicasForLog(`***`serverId`***`)`** is called by the coordinator during master crash recovery to find all log data for a failed server. By broadcasting to the cluster, it produces a map of where all segment replicas for a master's log are located across the cluster. This map is used by servers participating in recovery to find and acquire log data. Section 4.4 gives the details of this operation, and Chapter 5 describes how the replicas are used during recovery.

4.2 Appending Log Data

When a master receives a write request, it must make the new object durable; the master appends the new data to its DRAM log and forwards the log entry to a configurable number of backup servers. The default policy is to keep three copies of objects on disk in addition to the copy kept in the master's DRAM. These backups buffer the information in memory and return immediately to the master without writing to disk or flash. The master completes its request and returns to the client once all of the backups have acknowledged receipt of the log data. When a backup's buffer fills, it writes the accumulated log data to disk or flash in the background in a single large transfer, then deletes the buffered data from its memory.

Buffering and writing out data in large blocks in the background has two key performance benefits. First, disks are removed from the fast path; client write operations never wait on disks, giving RAMCloud write latencies that are only bounded by RPC round-trip times (microseconds rather than milliseconds). Second, reading and writing in large blocks amortizes the high cost of the seek and rotational latency of disks. Backups accumulate 8 MB of buffered data of a master's log before flushing it to disk. RAMCloud uses 8 MB

chunks in order to achieve effective bandwidth for each disk that is 85% of its maximum sequential read and write bandwidth even with low-end commodity hard disks (§6.2). This helps performance both while writing during normal operation and while reading during recovery.

However, while critical to RAMCloud's low-latency operation, buffering updates also requires that buffers on backups are as durable as disk storage. Otherwise, RAMCloud would lose recent object updates during power failures, since objects buffered on backups would be lost along with the data stored in the DRAM of masters. To solve this, backups depend on battery power to allow them to flush buffers to storage in the case of power loss. Unfortunately, typical data center batteries are limited [44] and do not last long enough to reliably flush 64 to 256 GB to disk during a power failure. To make this safe without requiring additional hardware, each backup limits its total buffer capacity to ensure all data can be flushed safely to disk on power failure. If a master attempts to replicate data to a backup whose buffers are full, the backup rejects the request and the master must retry elsewhere. Backup buffer capacity is configurable and should be based on a conservative estimate of the storage device performance (which determines how quickly it can write data to disk during a power failure) and expected battery life. Ideally, backups would only rely on a small fraction of their expected battery life; for example, backups could be configured with 1.5 GB of buffering. With two flash SSDs, each backup would be able to flush its buffers to disk in less than 5 seconds, which provides a wide margin of safety.

Masters allocate and manage space for their DRAM log in units of segments. Because new objects are always appended to the log, all new data goes to the most recently added segment in the log, called the *head* segment. Whenever a master's head segment becomes full, the master allocates an empty 8 MB DRAM segment and registers it with the replica manager using `allocateHead`. A segment also corresponds to the unit of buffering on backups, so a replica manager's main job is to replicate the DRAM segment identically into the buffers of backups as data is added. This same segment allocation process is used when a master first starts; before it can accept incoming data from client requests, it must first register a head segment in the same way.

After registering a new head segment, the contents of the old head segment are immutable. However, the old head segment may still have active replication operations ongoing for it, since the master may still be waiting for some of the data in the old head to be replicated. Whenever a replica for an immutable segment is fully replicated to a backup, it is flushed to disk and evicted from its DRAM buffer on the backup. The replica is never read or written again unless it is needed for recovery.

To improve throughput, masters service read and write requests from clients on multiple threads in parallel. This allows reads to proceed in parallel, but the replica manager also optimizes throughput for concurrent write operations by batching backup replication requests together. To store new data, a thread copies the data to the head segment of the in-memory log and then calls `sync` to wait for the data to be replicated. Meanwhile, if the replica manager is already busy sending or waiting on replication requests to backups, then other threads may append data to the in-memory copy of the head segment as well. Each time the replica manager completes a replication request to a backup, it checks to see if new data has been added to the head segment that needs be replicated. If it finds new data, then the replica manager sends out a single replication request to the backup covering all of the new data in the head segment. Figure 4.1 shows a detailed example of how batching works and how a master uses `sync` to ensure data is durable.

4.3 Scattering Segment Replicas

One of the main challenges of fast recovery is providing high read bandwidth from disk storage. For example, to recover 64 GB of DRAM in 1 second at least 64 GB/s of disk bandwidth is needed. Masters solve this problem by scattering segment replicas across all the storage devices attached to all the backups of the cluster; each log segment is backed up on a different set of servers (Figure 4.2). When a master fails, all of the backups of the cluster read replicas parallel, providing the high aggregate disk bandwidth needed for recovery.

To scatter log segments, whenever a new head segment is registered, a master's replica manager randomly chooses enough backups from among the other servers in the cluster to meet the log's replication requirement (three by default). The replica manager *opens* a

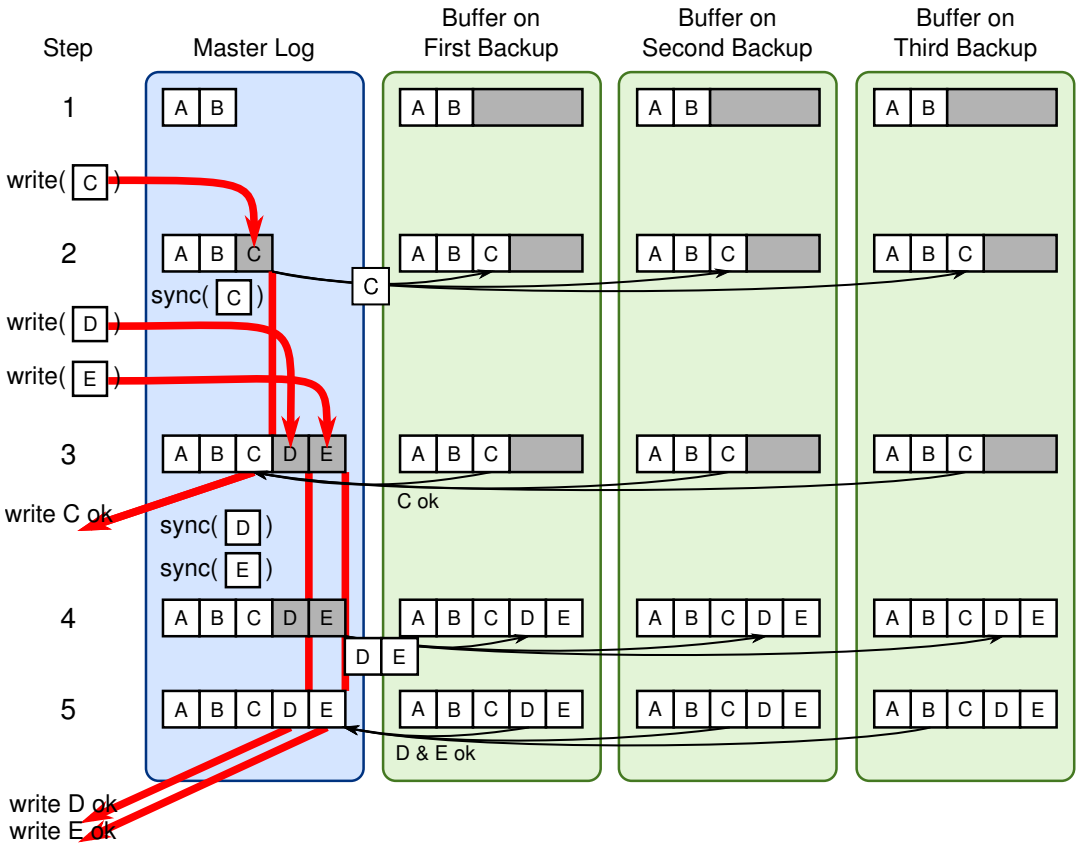


Figure 4.1: An example of how a master appends data to its DRAM log, how that data is replicated to backups, and how a master ensures appended data is durable for each operation. Objects colored gray on the master are not yet fully replicated and, thus, not guaranteed to be durable; objects colored white are fully replicated to backups. **Step 1:** the master's log contains two objects that are already replicated to the buffers of three backups. Subsequently, a client makes a request to write object C on the master. **Step 2:** the master copies the data to the head of its DRAM log and calls `sync` on the offset of the log up through C. This both blocks the master thread until the data has been replicated and notifies the replica manager that it should replicate all new log data. During the call to `sync`, additional requests arrive at the master to write objects D and E (with these operations being handled by additional threads). **Step 3:** Objects D and E are copied to the head of the in-memory log. Their threads call `sync` on the offsets in the log up through D and E, respectively. Replication of D and E to the backups is delayed on each backup until the backup is finished with the earlier replication request. Shortly afterward, the backups finish the replication of object C. The `sync` call for the write of C returns and the master returns to the client notifying it that the data has been stored. **Step 4:** After the completion of each replication RPC, the master checks for new log data to replicate; the master batches D and E in a single replication operation to each of the backups. **Step 5:** When the master receives acknowledgment of the replication of D and E, it returns a success notification to each of the respective waiting clients.

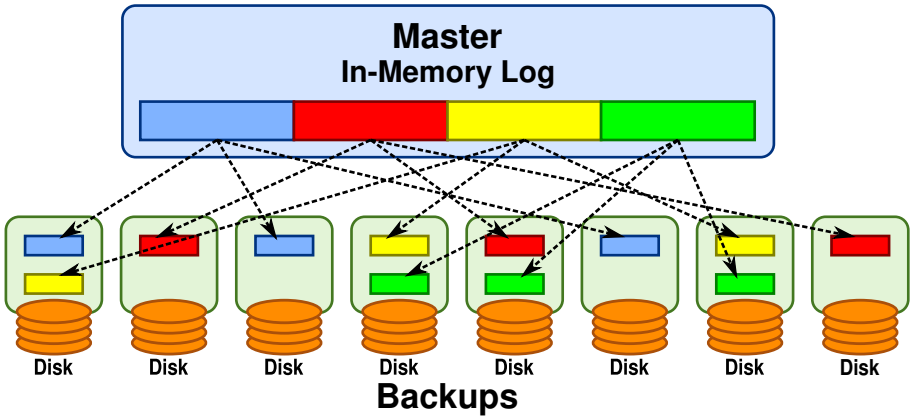


Figure 4.2: Masters randomly scatter replicas of segments from their DRAM log across all the backups of the cluster. By default, three replicas of each segment are written to disks on different backups in addition to the copy of the segment the master keeps in DRAM. Scattering segments enables recovery to take advantage of the high aggregate disk bandwidth of the cluster since all the disks of the cluster can read log data in parallel.

replica on each chosen backup. Backups respond by allocating both disk and buffer space and buffering a header for the replica. The header contains the id of the master that created the replica and a segment id that is used during recovery.

An added benefit to this approach of scattering replicas across backups is that each master can write at a rate that exceeds the disk bandwidth of an individual machine. When a master fills a segment replica buffer on one set of backups, it does not wait for the buffers to be flushed to disk before starting replication for a new segment on a different set of backups. This allows a single master to use the disk bandwidth of many backups in parallel, with different groups of backups replicating different segments. As an extreme example, a single master could hypothetically write to its log using the aggregate disk bandwidth of the entire cluster if other masters in the cluster were idle. In practice, the aggregate disk bandwidth of the cluster is likely to exceed the rate at which a single master can transmit data over the network.

Carefully choosing segment replica locations is critical in minimizing recovery times. In effect, as a master creates log segments during normal operation, it is optimizing the placement of its replicas to speed the recovery of its data in the case of its demise.

Ideally, in order to to recover the data from a failed server's log as quickly as possible, all the disks in the cluster

- would begin reading replicas from disk immediately,
- would read ceaselessly at maximum rate,
- collectively would read only one replica of each segment,
- and would all finish reading their share of the data in unison.

That is, for the fastest recovery, the work of reading the replicas of the failed server from disk should be distributed uniformly across all of the storage devices in the cluster. Unfortunately, there are several factors that make this difficult to achieve:

- Replica placement must reflect failure modes. For example, a segment's master and each of its backups 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. Masters must be aware of these differences in order to distribute replicas in a way that balances disk read times across all storage devices during recovery.
- Masters are all writing segments simultaneously, and they must avoid overloading any individual backup for best performance. Request load imbalance would result in queuing delays on backups. Furthermore, backups have limited buffer space; backups will ensure safety by rejecting new replicas, but unnecessary rejections would result in lost performance.
- Utilization of secondary storage should be balanced across the cluster.
- Servers are continuously entering and leaving the cluster, which changes the pool of available backups and may unbalance the distribution of segments.

One possible approach is to make all replica placement decisions on the coordinator. The coordinator has complete information on the number of servers in the cluster, their rack placement, and their storage capacities and bandwidths. By using this information and tracking where it had placed replicas, the coordinator could balance load evenly across backups and balance each master's log data across backups to minimize recovery times.

Unfortunately, making segment replica placement decisions in a centralized fashion on the coordinator would limit RAMCloud's scalability. For example, a cluster with 10,000 servers could back up 100,000 segments or more per second, so requests to the coordinator to determine replica placement would become a performance bottleneck.

Instead, each master decides independently where to place each replica, using randomization with an additional optimization step. When a master needs to select a backup and storage device for a segment replica, it uses the so-called "power of two choices randomized load balancing [38]." It chooses a few candidate devices at random from a list of all the storage devices from all of the backups in the cluster. Then, the master selects the best from among the candidate devices using its knowledge of where it has already allocated replicas and information about the speed of each device. To make this possible, backups measure the speed of their disks when they start up and provide this information to the coordinator, which relays it on to masters. The best storage device is the one that can read its share of the master's segment replicas most quickly from disk during recovery. A storage device is rejected if it is in the same rack as the master or any other replica for the same segment.

Using randomization gives almost all the desired benefits; it avoids centralization that could become a bottleneck, and it avoids pathological behaviors. Randomization naturally prevents masters from choosing backups in lock-step. It also balances request load and storage utilization across the cluster.

Adding the extra refinement step to masters' random backup selection balances expected disk reading time nearly as optimally as a centralized manager (see [38] and [2] for a theoretical analysis). For example, if a master uses a purely random approach to scatter 8,000 segments (enough to cover 64 GB of DRAM) across 1,000 backups with one disk each, then each disk will have 8 replicas on average. However, with 80% probability some disk will end up with 18 replicas or more, which will result in uneven disk utilization during recovery. Since recovery time is determined by the slowest disk, this imbalance would

more than double master recovery time. Adding just a small amount of choice makes the replica distribution nearly uniform and also allows for compensation based on other factors such as disk speed; Section 6.6 shows this technique works well even when the performance of many of the disks in a cluster is severely impaired. Choosing the best among two candidate devices is sufficient to get most of the benefit of the approach, but masters choose among five for each replica they place; there is little cost in doing so, and it slightly improves balance. Specifically, considering just two choices consistently results in 9 or 10 replicas on some of the backups, which would add 80 to 160 ms to recovery time over considering five candidates. This power of choice technique also handles the entry of new backups gracefully. Devices on a new backup are likely to be selected (five times) more frequently than existing devices until every master has brought the count of replicas on the new backup into balance with the rest of its backups.

While scattering replicas randomly has several benefits, it also has a drawback; as the cluster size scales, the system becomes increasingly vulnerable to correlated failures. For example, even for RAMCloud cluster sizes as small as 1,000 nodes, if ten machines fail simultaneously, all the replicas for at least one segment will be lost with near certainty. Intuitively, as more servers are added to the cluster, more segments must be scattered across the cluster. Each of these segments is stored on some 4-combination of the servers of the cluster (the master and its three backups). If the entire set of servers for *any* of these 4-combinations is lost, then some data will be unrecoverable. Because a cluster of 1,000 nodes will have nearly 8 million segments at capacity, even with just ten machines simultaneously unavailable, it is exceedingly likely that for at least one of these 4-combinations all four of its servers are among the ten that have failed. Thus, it is almost certain for at least one segment that all of its replicas would be unavailable. This problem affects not only RAMCloud but also other popular data center filesystems like HDFS [46]. The details of this problem are outside the scope of this dissertation, but a full analysis as well as a solution that has been integrated into RAMCloud can be found in [8].

Ideally, only one replica of each segment would be read from storage during recovery; otherwise, backups would waste valuable disk bandwidth reading unneeded data, slowing recovery. Masters solve this by marking one replica of each segment they create as the

primary replica. Only the primary replicas are read during recovery unless they are unavailable, in which case backups read the redundant replicas on demand (§5.4.2). This typically happens when the backup that contained a primary replica crashes, or if a primary replica is corrupted on storage. The recovery time balancing optimizations described above consider only primary replicas, since the other replicas are never intended to be read. The redundant replicas of a segment are scattered uniformly randomly across all the backups of the cluster, subject to the constraint that they are not placed on backups in the same rack as one another, in the same rack as the master, or in the same rack as the backup of the primary replica.

Another alternative would be to store one of the segment replicas on the same machine as the master rather than forcing to all replicas to be stored on remote machines. This would reduce network bandwidth requirements for replication, but it has two disadvantages. First, it would reduce system fault tolerance: the master already has one copy in its DRAM, so placing a second copy on the master's disk provides little benefit. If the master crashes, the disk copy will be lost along with the memory copy. It would only provide value in a cold start after a power failure. Second, storing one replica on the master would limit the burst write bandwidth of a master to the bandwidth of its local disks. In contrast, with all replicas scattered, a single master can potentially use the disk bandwidth of the entire cluster (up to the limit of its network interface).

4.4 Log Reassembly During Recovery

Recovering from a log left behind by a failed master requires some reassembly steps because the log's contents are scattered. First, the coordinator must find replicas of the failed master's log segments among the backups of the cluster. Then, the coordinator must perform checks to ensure that none of the segments are missing among the discovered replicas. A treatment of the more complex log consistency issues that arise due to failures is deferred until Section 4.5, which details how masters and the coordinator prevent stale, inconsistent, or corrupted replicas from being used during recovery.

4.4.1 Finding Log Segment Replicas

Only masters have complete knowledge of where their segment replicas are located, but this information is lost when they crash; as described in Section 4.3, recording this information on the coordinator would limit scale and hinder common-case performance. Consequently, the coordinator must first find the replicas that make up a log among the backups of the cluster before recovery can begin.

At the start of recovery, the coordinator reconstructs the locations of the crashed master's replicas by querying all of the backups in the cluster using the `findReplicasForLog` operation. Each backup maintains an in-memory list of replicas it has stored for each master, and quickly responds with the list for the crashed master. The coordinator aggregates the locations of the replicas into a single map. By using RAMCloud's fast RPC system and querying multiple backups in parallel, the replica location information is collected quickly.

Using broadcast to locate replicas among backups may sound expensive; however, broadcasting to the entire cluster is already unavoidable at that start of recovery, since all of the servers of the cluster participate. In fact, the cost of this broadcast is mitigated, since the backups of the cluster begin reading replicas from storage immediately after receiving the broadcast; this allows backups to begin loading and processing recovery data even while the coordinator finishes setting up the recovery.

4.4.2 Detecting Incomplete Logs

After backups return their lists of replicas, the coordinator must determine whether the reported segment replicas form the entire log of the crashed master. Redundancy makes it highly likely that the entire log will be available, but the system must be able to detect situations where some data is missing (such as if multiple servers crash simultaneously).

RAMCloud avoids centrally tracking the list of the segments that comprise a master's log by making each log self-describing; the completeness of the log can be verified using data in the log itself. Each new head segment added to the log includes a *log digest*, which is a list of identifiers for all segments in the log at the time the segment was created. The log digest contains the exact list of segments that must be replayed in order to recover to a consistent state. Log digests are small (less than 1% storage overhead even when

uncompressed, assuming 8 MB segments and 8,000 segments per master).

During the replica discovery phase of recovery, as part of `findReplicasForLog`, each backup returns the most recent log digest from among the segments it is storing for that log. The coordinator extracts the log digest from the most recently written segment among those on all the backups, and it checks the list of segments in the log digest against the list of replicas it collects from the backups to ensure a complete log has been found.

One threat to this approach is that if all the replicas for the most recent segment in the log are unavailable, then the coordinator could unknowingly use an out-of-date log digest from a prior segment. In this case, the most recent digest the coordinator could find would not list the most recent segment, and it would have no indication that a newer segment or log digest ever existed. This must be handled, otherwise the system could silently recover to an inconsistent state that may not include all the data written before the master crashed.

To prevent this without resorting to centralization, a master marks replicas that contain an up-to-date digest and clears the mark on each replica when its digest becomes out-of-date. Figure 4.3 shows how this works as a master adds a new head segment to its log. When a master uses `allocateHead` to transition to a new head segment, a log digest is inserted and each of the replicas is marked as *open* on backups. The open bit indicates that the digest found in the replica is safe for use during recovery. Only after all of the replicas for the new segment contain the new log digest and have been marked as open does the master mark the replicas for the old head segment in the log as *closed*. This approach guarantees that the replicas from at least one segment are marked as open on stable storage; if the coordinator cannot find an open replica during recovery, then the log must be incomplete.

One complication with this approach is that open replicas from two different segments may be found during recovery: replicas from the head segment or its predecessor may simultaneously be marked as open on stable storage. One additional constraint ensures this is safe. The log ensures that no data is ever replicated into a segment until all of the replicas of the preceding segment have been marked closed (Section 4.5.3 describes what happens if this is impossible due to an unresponsive backup). As a result, the coordinator can use any of the log digests it finds in open segments; if a newer open head segment had existed and all of its replicas were lost, then the replicas could not have contained object data.

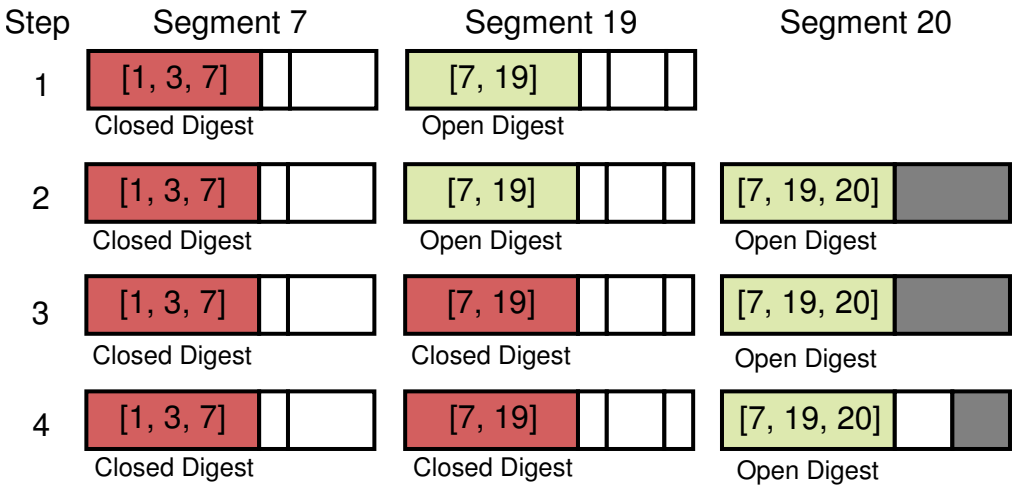


Figure 4.3: When adding a segment to its log, a master orders operations to ensure that if it crashes at any point, it will always leave behind an up-to-date log digest that covers all the segments in the log that contain data needed for recovery.
Step 1: A master has a log that contains two segments: 7 and 19. Segment 19 is the current head segment and is full. The master adds a new segment to its log using `allocateHead`.
Step 2: The master injects a new log digest into Segment 20 upon its registration and waits for the digest to be fully replicated on all of the segment's backups.
Step 3: After the digest in Segment 20 is replicated, the master sends a message to the backups for Segment 19 marking it as closed. During recovery, log digests from closed replicas are ignored. After this step, only the log digest in replicas of Segment 20 will be used during recovery.
Step 4: The master can safely begin to add data to Segment 20, knowing that the data stored in it is guaranteed to be recovered.
If the master fails between Steps 2 & 3: Recovery could use the digest from replicas of Segment 19 or 20, but either is safe since Segment 20 contains no data.
If the master fails after Step 3: Recovery is guaranteed to use the digest in replicas of Segment 20, so the master can safely add data to the segment.

If an open replica containing a log digest cannot be found or a replica for each segment in that digest cannot be found, then RAMCloud cannot recover the crashed master. In this unlikely case, RAMCloud notifies the operator and waits for backups to return to the cluster with replicas of the missing segments. Alternatively, at the operator's discretion, RAMCloud can continue recovery with loss of data. Section 5.6.1 details how RAMCloud handles multiple servers failures where some data is unavailable.

So far these rules describe how to prevent RAMCloud from recovering when it is unsafe to do so; however, a dual to that concern is to ensure, to the extent possible, that RAMCloud always recovers if it safe to do. For example, the replication rules given earlier prevent a master from replicating data into a segment until all preceding segments are closed, but there is no limit on how far ahead a master can allocate empty open segments on backups. This allows masters to open several segments in advance, working ahead to hide the initial cost of contacting backups. If a master works ahead to create several open segments that only contain a log digest and no data, then the system is prone to detecting data loss even when none has occurred. The problem occurs when the log digest for one of these empty segments is flushed to backups before the log digest for a preceding empty segment. The log digest from the later segment will be found on recovery, but it includes the preceding segment that has yet to be created on backups. During recovery, the coordinator would assume data had been lost when it had not (since the missing segment must have been empty). To prevent this problem while still allowing masters to open log segments in advance, replicas are not opened on backups for a segment until replicas for all segments preceding it in the log have been opened on backups. That is, replicas for segments are always opened in log order on backups.

4.5 Repairing Logs After Failures

A master's log is key in recovering from server failures, but failures can damage the log. Each master must take corrective steps after the loss or corruption of log segment replicas. Section 4.5.1 describes how a master recreates lost replicas in order to maximize the chance that the coordinator will find a complete log if the master crashes. Subsequent subsections describe how inconsistent replicas can arise under specific failure scenarios, and

how RAMCloud prevents these replicas from being used during recovery.

RAMCloud allows recovery to use *any* available replica of a segment in order to improve the chance that master recovery will be possible after a crash. This read-any property creates a tension between durability and availability; only requiring a single replica of each segment for recovery improves reliability, but it also requires the immediate correction of any inconsistencies before operation can continue after a failure. The set of replicas that RAMCloud can use for recovery must be consistent at all times, and this constraint guides the steps masters take to safeguard their logs when a server fails.

4.5.1 Recreating Lost Replicas

Each server acts as both a master and a backup, so when a server fails not only are the DRAM contents of the master lost, but segment replicas stored on its disk are lost also. Without action, over time replica loss could accumulate and lead to data loss. Masters restore full durability after backup failures by recreating lost replicas.

The coordinator notifies all of the masters whenever a backup fails. Every log is likely to have had at least some replicas on the failed backup, since masters scatter replicas across the entire cluster. Each master checks its segments to identify those that had a replica on the failed backup, and it begins creating new replicas in the background.

Recovering from a lost backup is faster than recovering from a lost master. New replicas are scattered across all of the disks in the cluster, and all of the masters recreate lost replicas concurrently. If each master has 64 GB of memory then each backup will have about 192 GB of data that must be rewritten (assuming 3 replicas for each segment). A cluster with 200 backups each having 1 GB of unused durable buffer space can store the new replicas in less than one second; maintaining an additional 1 GB of durable buffer space would require a few seconds of spare battery backup capacity, which should be practical for most deployments. For comparison, master recovery requires about 30% more disk I/O. In master recovery, 256 GB of data must be transferred to recover the DRAM of a failed master: 64 GB must be read, then 192 GB must be written for replication while incorporating the data into the logs of the recovery masters (§5.4).

Lost replica recreation is not on the critical path to restoring availability after a failure,

since the replicas on a backup are stored redundantly on other backups. However, rapidly recreating lost replicas improves durability, because it reduces the window of time that segments are under-replicated after failures (an analysis of this effect is given in Section 6.7). On the other hand, aggressively recreating replicas increases peak network and disk bandwidth utilization after failures. In particular, large network transfers are likely to interfere with normal, client requests, which are small. In the future, it will be important that RAMCloud retain low-latency access times and high performance even with high background network bandwidth utilization, though this will require innovation at the network transport layer that is deferred to future work.

4.5.2 Atomic Replication

Unfortunately, using the normal segment replication routines to recreate segment replicas could lead to inconsistencies when the cluster experiences repeated failures. If a master fails while in the middle of recreating a replica on a backup, then the partially recreated replica may be encountered while recovering the master. RAMCloud assumes it can recover using any available replica for each of the segments, so it must ensure these inconsistent replicas are never used.

A master prevents these inconsistencies by recreating each lost replica atomically; such a replica will never be used unless it has been completely written. To implement this, a master includes a flag in each replication request that indicates to the backup that the replica is incomplete. During recovery, backups check this flag and do not report incomplete replicas to the coordinator. Once the data in the backup's durable replica buffer is complete and identically matches the segment in the master's memory, the master clears the flag on the replica. Effectively, the replica atomically becomes available for use in recovery when it becomes consistent with the master's DRAM copy of the segment.

4.5.3 Lost Head Segment Replicas

Replacing lost replicas restores durability after failures, but the lost replicas themselves are also a problem; if they are found later and used in a master recovery, then they could cause recovery to an inconsistent state. If a master is replicating its head segment to a

backup that fails, it recreates the replica on another backup and continues operation. In the meantime, the backup containing the lost replica may restart, but the lost replica stored on it may be out-of-date. The replica would be missing updates the master had made to its head segment since the backup first failed. If the master were to fail, then recovery could use this out-of-date replica and recover to an inconsistent state.

One solution to this problem is to have masters suspend all update operations until the backup containing the lost replica restarts. However, this is impractical, since the data on masters must be highly available. A master cannot be suspend operations for the minutes it could take for a backup to restart. Worse, a master cannot be sure if a backup with a lost replica will ever restart at all.

Another approach is to never use replicas found on storage after a backup restarts; this would ensure that any replicas that missed updates while inaccessible would not be used during recovery. Unfortunately, this approach does not work under correlated failures. The simplest example is a power failure. If all the servers in a cluster were forced to restart, then none of the replicas found on disk could be used safely. The operator would be faced with a difficult choice; if she restarted the cluster using the replicas found on disk, the system might be in an inconsistent state. The only other option would be to discard all of the data.

Ideally, even if all of the replicas for a segment were temporarily unavailable, RAMCloud would recover if consistent replicas of the segment were later found on disk. To make this possible, masters take steps to neutralize potentially inconsistent replicas. By eliminating problematic replicas, masters can resume performing client requests quickly while still ensuring consistency, and recovery can still use the up-to-date replicas found on backups when they restart while ensuring that stale replicas are never used. When a master discovers that a replica of its head segment has become unavailable, it takes the following steps to neutralize potentially inconsistent replicas.

1. First, the master suspends replication and acknowledgment for all ongoing client operations. This is because a replica exists on the failed backup that cannot be changed or removed, so ongoing operations that were not known to have been recorded to that replica must be held until the lost replica can be neutralized. If master recovery occurs before the lost replica is neutralized, it is safe to use any of the replicas including the lost replica, since all of them include the data for all acknowledged client

operations. Barring additional failures, the master only has to hold operations for a few milliseconds before it is safe to resume normal operation.

2. Second, the master atomically creates a new replica elsewhere in the cluster just as described in Section 4.5.2. This step eliminates the master's dependency on the lost replica to fulfill its replication requirement; after this new replica is created, the master is free to kill off the lost replica without jeopardizing durability.

3. Third, the master neutralizes the lost replica. This is a challenge, since the failed backup cannot be contacted. Instead, the master informs the coordinator of the lost replica, and the coordinator agrees not to use the replica during recovery. Minimizing the burden on the coordinator is critical; more detail on how this is achieved is given below.

4. Finally, after neutralizing the out-of-date replica, the master resumes normal operation and begins replicating and acknowledging client requests.

One possible approach masters could use to invalidate potentially inconsistent replicas is to keep an explicit list of "bad" replicas on the coordinator for each master's log. If a master were informed of the crash of a backup to which it was actively replicating a segment, the master would add a pair to the list on the coordinator that contains the affected segment id and id of the failed backup server. During recovery, the coordinator would ignore any replicas that are part of the invalid replica list for the crashed master.

Unfortunately, these invalid replica lists would be cumbersome to maintain; each list would need to be part of the coordinator's persistent state, and when a master freed a segment that had a replica in the list, the corresponding list entry would need to be removed so that the list would not grow without bound.

Instead, each master stores a single *log version number* on the coordinator to achieve the same effect. Figure 4.4 gives an illustration of how this mechanism works. As a master replicates its head segment, its replicas are stamped with the master's current log version. During recovery, the coordinator compares the failed master's log version number with the versions the master stamped on each replica to ensure they are up-to-date. When a master loses a replica for its head segment, it increments the local copy of its log version number,

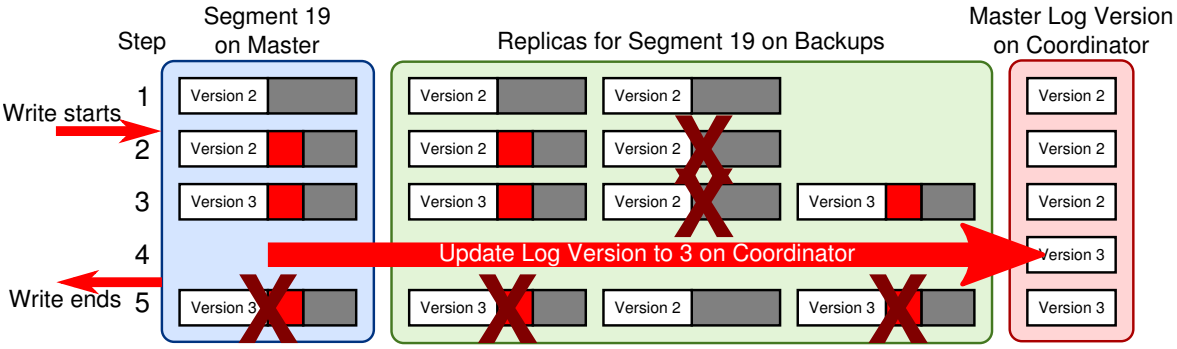


Figure 4.4: An example of how a master and the coordinator work together to neutralize inconsistent replicas. Inconsistent replicas arise when a master is actively replicating its head segment to a backup that fails. In this example the master's goal is to maintain two on-backup replicas of its segment. Replicas lost due to server failures are indicated with an 'X'.
Step 1: A master has a partially filled segment tagged with log version number 2. Each of its two backups is fully up-to-date and consistent with the master's in-memory segment. A client starts a write operation, which requires the master to append data to its in-memory head segment.
Step 2: The master successfully replicates the newly appended data to its first backup, but the second backup fails before the replication operation is acknowledged to the master.
Step 3: The master sends a new version number of 3 to its first backup, which tags its replica with the new version number. Finally, the master recreates a new, up-to-date replica on a third backup, elsewhere in the cluster.
Step 4: The master notifies the coordinator that its log version number is now 3, and it waits for an acknowledgment from the coordinator that it will never use replicas with a lower version number. Only after this point can the master safely acknowledge the pending client write.
Step 5: Afterward, if the master and both backups were to crash, and the backup containing the out-of-date replica were to restart the coordinator would not use the replica.
Importantly, a crash at any point in this sequence is safe. If the master were to crash during steps 2, 3, and 4, then the master may or may not be recovered with the ongoing write operation; however, since the write has not been acknowledged, this is consistent with the client's expectations. After step 4, the client is guaranteed to see the effects of its write even if the master fails and recovers.

it recreates the replica elsewhere, and it updates the log version number on the remaining available replicas for its head segment. Afterward, it updates its log version number on the coordinator as well; this ensures that the coordinator will discard any replicas with a lower log version during recovery. Replicas marked as closed on stable storage are excluded from this protocol; they can never be a source of inconsistency since they are immutable and are recreated atomically when lost. To achieve this, masters set the log version number of replicas to ∞ as they close them.

While this approach involves the coordinator, it does not significantly limit the scalability of the cluster, since it only requires RPCs to the coordinator on failures (as opposed to regularly during normal operation). A master has a single head segment that it is (by default) triplicating across the backups of the cluster, so a typical backup will only have three replicas being actively replicated to it at a time from the head segments of all of the masters of the cluster. Therefore, a typical backup failure will only result in about three RPCs to the coordinator to invalidate replicas; this is a trivial load even in a cluster of 10,000 machines where server failure is expected about once every 15 minutes.

4.5.4 Preventing Inconsistencies Due to Overloaded Backups

As mentioned in Section 4.2, overloaded backups may need to reject replication operations from masters. A backup is overloaded when

- it has insufficient disk space to store a replica,
- it has insufficient buffer space to store a replica, or
- it has insufficient network bandwidth to accept new replica data quickly.

Rejecting replication requests is a form of transient failure; if backups arbitrarily refuse replication operations then the replicas they store may become inconsistent.

To prevent inconsistencies, a backup may only reject a master's initial replication request for each replica. If a backup rejects this first request to store data for a replica, then it never records any state about the segment and the master retries elsewhere. However, if the backup acknowledges it has accepted and replicated the data in the first request, then it must have already reserved all resources needed to ensure the durability of the replica. This

means backups must always allocate the needed disk and buffer space for the replica before they respond to an replication request that starts a new replica. Once a replica is opened on a backup, all well-formed replication requests for that replica by the master are guaranteed to succeed in the absence of a serious failure (for example, a network partition or a failed disk), so replicas never become inconsistent due to rejected replication operations.

4.5.5 Replica Corruption

Finally, RAMCloud must address inconsistencies that arise due to storage or media corruption. Masters prevent the use of corrupted replicas by embedding a checksum in metadata stored along with each replica. The checksum covers log entry types and lengths in the prefix of the replica up through a length specified in the metadata. The metadata itself (excluding the checksum field) is covered by the checksum as well. Upon reading a replica during recovery, a backup computes a checksum of the metadata and the log entry types and lengths stored in the specified prefix of the replica and compares the result with the checksum stored in the metadata. If the checksums do not match, then the replica will not be used. Object data is not covered by the checksum. Each object is covered by an individual checksum that is checked by recovery masters during recovery (§5.6.2).

4.6 Reclaiming Replica Storage

As clients append new objects to a log, these new object values supersede older values in the log, and logs accumulate entries over time that waste space both in DRAM and on disk. Since recovery only depends on the most recent value associated with each key, these stale values are not needed. Eventually, this wasted space grows and must be consolidated and reclaimed for reuse with new objects.

To combat this, masters use cleaning techniques similar to those used in log-structured filesystems [42] to "repack" log data and reclaim space. Each master keeps a count of unused space within each segment, which accumulates as objects are deleted or overwritten. The master reclaims wasted space by occasionally invoking a *log cleaner*. The cleaner selects one or more segments to clean and rewrites the live entries from them into a smaller

set of new segments. Afterward, the master is free to reuse the DRAM that contained the cleaned segments. Similarly, replicas of the cleaned segments are no longer needed; the master contacts backups that hold these replicas, notifying them that it is safe to reuse the space that the replicas occupy on storage and any associated buffers. The details and policies of the log cleaner are analyzed elsewhere [43].

Typically, masters notify backups when it is safe to reuse storage space, but due to server failures, replicas cannot always be removed from backups using explicit free RPCs from the masters that created them. There are two cases when a replica must be freed by some other means: when the master that created it crashes (§4.6.1), and when a backup with the replica on its storage crashes and restarts later (§4.6.2).

4.6.1 Discarding Replicas After Master Recovery

When a master crashes, all of the backups are likely to contain some replicas from the crashed master, and eventually these replicas must be freed from storage. However, the normal-case approach of using explicit free notifications to backups does not work, since the master that created them is gone. At the same time, RAMCloud needs these replicas immediately after a master failure to perform recovery, so backups must not be overly eager to discard replicas when masters crash. As a result, backups are notified when a failed master has been recovered. Afterward, the replicas of the master's log segments are no longer needed and backups discard them.

To make this possible, each backup tracks changes to a *server list* that contains the identities, statuses, and network addresses of other cluster members. The coordinator maintains the authoritative copy of this list and disseminates updates to cluster members as server statuses change. Initially, when a server joins the cluster it is marked as *up*. When a server crashes, its entry in the server list is marked as *crashed* until all tablets owned by the crashed master are recovered elsewhere. At that point, the server list entry is removed. During the course of normal operation, a backup retains replicas from a server that is marked as up or crashed in the server list, and it reclaims space used by replicas from a server that not present in the server list (which indicates the server is down).

4.6.2 Redundant Replicas Found on Storage During Startup

A second problem arises because of backup failures. As described in Section 4.5.1, masters recreate replicas that were stored on failed backups. If masters have already recreated the replicas stored on a failed backup before a new backup process restarts on the same node, then the newly starting process may find replicas on storage that are no longer needed. On the other hand, if multiple servers failed at the same time, then a restarting backup process may find replicas on storage that are needed to recover some failed masters: all of the other replicas of some segments may have been lost before they could be recreated.

Therefore, each backup scans its storage when it first starts up, and it checks replicas left behind by any old, crashed RAMCloud process on that node to see if they are still needed. As before, for each of the replicas it finds on disk, it uses the server list to determine whether the replica should be retained or discarded. Just as during normal operation, replicas for a crashed server must be kept, since the server may be under active recovery. Replicas for a down server can be discarded, since its data must have been recovered elsewhere.

Replicas from a master that is still up present a challenge. If a replica from a server marked as up in the server list is found on storage, then the server that created it may have already recreated the replica elsewhere, in which case it can be discarded. However, it is also possible that the master has not yet recreated the lost replica, in which case the replica found on storage may still be needed if the master crashes. For example, this occurs when a backup finds a replica on storage for a master that is marked as up in the backup's server list, but the master has actually crashed and the backup has not been notified yet; in that case, the replica on storage may be critical for recovering the failed master and must not be discarded by the backup.

On startup, if a backup finds a replica on storage that was created by a master that is still up in the backup's server list, then the backup queries the originating master directly to determine whether the replica is safe to discard or not. When queried about a specific replica, a master informs the calling backup that it is safe to discard the replica if the master's corresponding segment is already adequately replicated. If the segment is still undergoing replication, then the master tells the backup to keep the replica. In this case, the backup will recheck with the master for this replica periodically to see if the segment

has been fully replicated (in the expectation that the master will eventually recover from the loss of the replica making it safe to discard). If the master fails in the meantime, then the backup reverts to the usual rules; it retains the replica until the master has been recovered. This process of querying replica statuses from masters only lasts briefly when a server starts until all replicas found on storage have been recovered elsewhere by masters or the masters from which they originated have crashed; after that, replicas for servers that are up in the server list are retained until explicitly freed, as usual.

This approach of querying the masters requires one additional constraint. The problem is that when queried about whether a particular replica is needed, a master's reply is only based on whether or not the corresponding segment is fully replicated. This presupposes that the master has already been informed that the backup crashed. However, this presupposition could be wrong: it may be the case that the master has yet to receive or process the notification of the failure that led to the restart of the backup. If this is the case, then the master is counting the old replica found on storage as a valid replica for the segment. If the master reports to the backup that the segment is fully replicated and the restarting backup process deletes it, then the segment will be under-replicated until the master receives the server failure notification and recreates the replica elsewhere.

To solve this, the coordinator guarantees that all servers in the clusters will learn of a backup's crash before they learn that the backup has been restarted. When a server process first starts up and *enlists* with the coordinator, it provides the server id of the prior RAMCloud server process that ran on that machine, which it recorded on its storage device. The coordinator ensures that when it sends server list change events to the cluster members, the failure notification for the former server precedes the enlistment notification for the server that replaces it. After enlistment, the restarting server updates the server id on its storage, so that the next crash will be handled correctly.

To ensure that a master has been informed of the failure of a restarting backup before it replies to a replica status query, the master ensures that a querying backup is up in its server list. The ordering constraint on server list updates ensures that if the backup is up in the master's server list, then the master must have processed the failure notification about the failed process that the new backup is replacing. When a master processes a failure notification of a backup, it immediately "forgets" about replicas on the failed backup and

begins replication elsewhere for affected segments. Hence, if a backup querying a master about a replica is up in the server list of the master and the master is not actively replicating the segment, then the master must have already re-replicated the segment elsewhere. When a querying backup is not in a master's server list, then the master tells the backup to ask about the replica again later. This ensures that a master never prematurely tells a backup to discard a replica.

4.7 Summary

Each master in RAMCloud uses a log that provides high durability for client data, low-latency client reads and writes during normal operation, and high bandwidth reading for fast recovery. Masters store one copy of their log entirely in local DRAM to support low-latency data access. Masters replicate their logs for durability using cheap disk-based storage on other nodes, called backups. Because backups buffer updates (requiring them to briefly rely on battery backups during power failures), masters can durably replicate objects with latencies limited by network round trip times rather than hard disk seek times. To enable the high bandwidth while reading the log needed for fast recovery, masters divide their logs into segments and scatter replicas of them across the entire cluster, so replicas can be read from all the disks of the cluster in parallel.

A master's replica manager works in the background to maintain redundancy of log segments as nodes fail, and it takes steps to ensure the master will only be recovered when a consistent and up-to-date view of its log is available. Masters inject self-describing metadata (log digests) into logs so that they can detect data loss during recovery. When backups crash and restart, some of the replicas they hold on storage may become stale. Masters ensure these stale replicas are never used during recovery by tagging each replica with a log version number that is also stored on the coordinator. The coordinator uses the log version number to ensure that recovery only uses up-to-date replicas. The log version number on the coordinator is only updated by a few servers in the cluster when a backup fails, so it will not become a bottleneck, even in large clusters. By containing the complexity of fault-tolerance and consistency, replica managers ease the task of recovery.

Replicas stored on backups are typically freed by a master's log manager when they are

no longer needed, but space reclamation must happen in other ways when masters crash. Backups receive server status notifications from the coordinator. When a master has been fully recovered, backups discard the replicas created by that master that are still on storage. When a backup fails and a new backup process restarts on the same node, some of the replicas found on storage may no longer be needed. Backups query the originating master for each on-storage replica to see if it is still needed. Once the master has recreated the replica elsewhere in the cluster, the old replica is removed from the restarting backup's storage.

The next chapter describes how RAMCloud uses these logs to recover the contents of a failed master in just 1 to 2 seconds.

Chapter 5

Master Recovery

When a master crashes in RAMCloud, the data it contained is unavailable until it can be recovered into DRAM on other servers in the cluster. As described in Chapter 4, each master stores the objects in its DRAM to a log that is stored redundantly across all of the disks of the cluster. This chapter details how RAMCloud uses that log to restore availability when a master fails.

RAMCloud uses fast recovery instead of relying on expensive in-memory redundancy or operating in a prohibitively degraded state using data on disk. If master recovery is no longer than other delays that are common in normal operation and if crashes happen infrequently, then crash recovery will be unnoticeable to applications. RAMCloud's goal is to recover failed servers with 64 to 256 GB of DRAM in 1 to 2 seconds. For applications that deliver content to users over the Internet, this is fast enough to constitute continuous availability.

RAMCloud's master failure recovery mechanism relies on several key concepts that make it fast and safe while minimizing complexity.

- **Leveraging scale:** Most importantly, RAMCloud leverages the massive resources of the cluster to recover quickly when masters fail. No single machine has the disk bandwidth, network bandwidth, or CPU cycles to recover the contents of a crashed server quickly. Recovery takes advantage of data parallelism; the work of recovery is divided across thousands of backups and hundreds of recovery masters, and all of the machines of the cluster perform their share of recovery concurrently.

- **Heavy pipelining:** Not only is recovery parallelized across many machines to minimize recovery time, but equally critically, each server pipelines its work to efficiently use all of its available resources. Each machine overlaps all of the stages of recovery data processing; during recovery, a single server may be loading data from disk, processing object data, sending log entries out over the network, receiving log entries from other machines over the network, reconstructing data structures, and sending out data for replication, all at the same time.

- **Balancing work:** Recovery time is limited by the server that takes the longest to complete its share of work, so precisely balancing recovery work across all the servers of the cluster is essential for fast recovery. Large scale makes this a challenge; with hundreds or thousands of machines working together, recovery must operate with minimal coordination, yet each individual machine must stay busy and complete its share of work at about the same time as the rest of the cluster.

- **Autonomy of servers:** Each server makes local choices to the extent possible during recovery. This improves scalability since the cluster needs little coordination. After initial coordination at the start of recovery, backups and recovery masters are given all the information they need to work independently. Recovery masters only call back to the coordinator when their share of recovery is complete.

- **Atomicity and collapsing error cases:** In a large-scale cluster, failures can happen at the worst of times; as a result, RAMCloud must operate correctly even when failures happen while recovering from failures. Failure handling is simplified during recovery by making the portion of recovery on any individual server atomic and abortable. That is, if a server participating in recovery crashes or encounters a problem, it can give up on its share of the work without harm. Portions of a recovery that were aborted are retried later, hopefully when the conditions that prevented success have subsided. This reduces the number of exceptional cases that recovery must handle, and it allows most issues to be handled by retrying recovery. Similarly, multiple failures and total cluster restart are handled using the master recovery mechanism, which eliminates the need for a separate, complex mechanism and exercises the master recovery code more thoroughly.

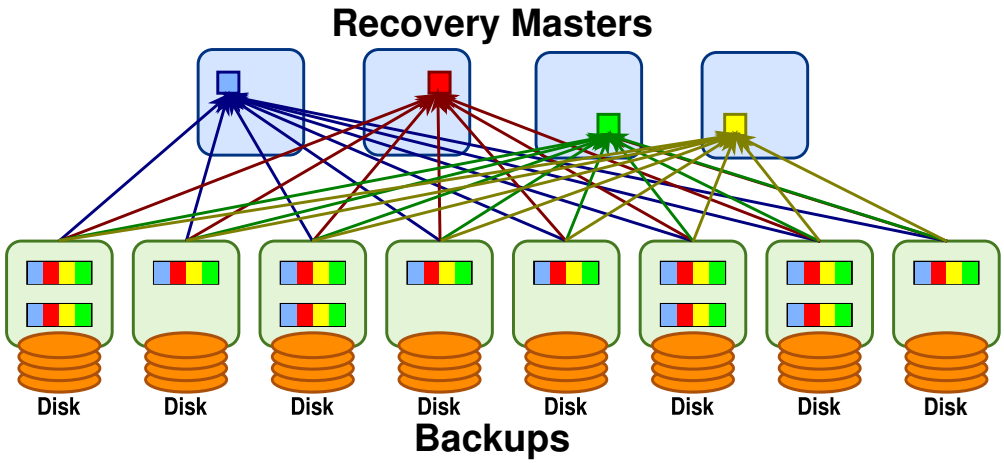


Figure 5.1: The work of recovery is divided across all the backups of the cluster and many recovery masters. Each backup reads segment replicas from its disk and divides the objects into separate buckets for each partition. Each recovery master only fetches the objects from each backup for the keys that are part of its assigned partition.

5.1 Recovery Overview

When a RAMCloud storage server crashes, the objects that had been present in its DRAM must be recovered by "replaying" the master's log before the objects can be accessed by clients again. This requires reading log segment replicas from backup storage back into DRAM, processing the records in those replicas to identify the current version of each live object, and reconstructing a mapping from each key to the current version of each object. The crashed master's data will be unavailable until the objects from its log are read into DRAM and a mapping from each key to the most recent version of each object has been reconstructed.

As described in Chapter 3, recovery must overcome two key bottlenecks. First, data must be read from storage at high bandwidth in order to recover an entire server's DRAM in 1 to 2 seconds. To solve this, each master scatters its log data across all of the disks of the cluster as 8 MB segments. This eliminates disks as a bottleneck, since the log data can be read from all of the disks of the cluster in parallel. Second, no single server can load all of the failed master's data into DRAM and reconstruct a hash table for the objects in 1 to 2 seconds. Consequently, the failed master's data is partitioned across 100 or more

recovery masters by key. Each recovery master replays the log entries for a subset of the crashed master's keys, reconstructs a hash table for its assigned portion of the failed master's key space, and becomes the new home for the recovered objects. This allows RAMCloud to spread the work of replaying log data across hundreds of NICs, CPUs, and memory controllers. Figure 5.1 illustrates how recovery is spread across all of the machines of the cluster and how data is loaded to disks and shuffled among recovery masters during recovery.

As an example, a cluster of 1,000 machines should be able to recover a failed server with 64 GB of DRAM in less than one second. Recovery masters process data as it is loaded by backups. With a single 100 MB/s disk on each server, a failed server's log can be read in about a half of a second. Similarly, 100 recovery masters can load that data into memory in about a half of a second using a 10 Gbps network. Furthermore, backups and recovery masters overlap work with each other to minimize recovery time.

Overall, recovery combines the disk bandwidth, network bandwidth, and CPU cycles of many of machines in a highly parallel and tightly pipelined effort to bring the data from the crashed server back online as quickly as possible. The coordinator supervises the recovery process, which proceeds in four phases that will be described in detail in the sections that follow:

1. **Failure detection:** When a server crashes, the first step in recovery is detecting the failure. Failure detection must be fast since any delay in detecting a failure also delays recovery. To speed detection, servers monitor one another, and peers of a crashed server detect its failure and report it to the coordinator.

2. **Setup:** During the setup phase of recovery, the coordinator decides how the work of recovery will be divided across the machines of the cluster, and it provides each machine will all the information it needs to perform its share of the work without further coordination. During this phase, the coordinator tells backups to start loading replicas, verifies the complete log is available on backups, and assigns the crashed master's tablets to recovery masters.

3. **Replay:** The replay phase is both data-parallel and pipelined. Each recovery master sends requests to several backups in parallel to fetch the objects relevant to the portion

of the crashed master it is recovering. Recovery masters incorporate the data returned by backups into their log and hash table and re-replicate the data as part of their log, just as during normal operation. At the same time, backups load segment replicas from the crashed master stored on disk, bucket objects by partition, and accept new replica data. Machines operate independently of the coordinator during the replay phase.

4. **Cleanup:** Each recovery master notifies the coordinator when it finishes replay, and it begins servicing requests for the portion of the tablets that it recovered. Once all of the recovery masters complete and all of the partitions of the key space have been recovered, the backups of the cluster begin to reclaim the on-disk storage occupied by the failed master's log and free up any state and buffers used as part of recovery. If recovery fails for any of the partitions of the crashed master, then the log data is retained and recovery is retried later.

The following sections provide detail on each of these phases.

5.2 Failure Detection

Server failures must be detected quickly in order to recover quickly, since detection time delays recovery. Whereas traditional systems may take 30 seconds or more to decide that a server has failed, RAMCloud must make that decision in a few hundred milliseconds. Fast failure detection also improves durability, since the recreation of lost replicas from failed backups starts more quickly.

RAMCloud detects failures in two ways. First, if a server is slow in responding to an RPC, the caller (either a RAMCloud client or another RAMCloud server) pings the callee periodically in the background until the RPC response is received. These pings are application-level RPCs (as opposed to lower-level ICMP pings that would only be serviced by the kernel), so they provide some measure of assurance that the remote RAMCloud server is still functional. If the callee is responsive to the ping requests, then the caller assumes the callee is still alive and that the callee is still processing the caller's RPC request. If a ping request times out, then the caller assumes the callee has crashed.

When a one server suspects another has failed, it notifies the coordinator, which then investigates the server. This extra step prevents servers from being detected as failed when a communication problem between two servers is due to a problem at the caller rather than the callee. After receiving a failure suspicion notification, the coordinator issues ping requests to the suspected server. If the coordinator's application-level ping requests timeout, then the server is marked as failed and the coordinator initiates recovery for it.

Unfortunately, this first technique only detects the failure of servers that are actively servicing RPCs; if some servers only service RPCs infrequently, then the approach described above may delay recovery. RAMCloud uses a second technique so that these failures can be detected more quickly. Cluster members continuously ping one another (at application-level) to detect failures even in the absence of client activity; each server periodically issues a ping RPC to another server chosen at random. Unanswered ping requests are reported to the coordinator. Just as before, the coordinator verifies the problem by attempting to directly ping the server, then initiates recovery if the server does not respond.

Randomizing ping operations means the detection of failed idle servers is only probabilistic. There is some chance that failed idle servers could go undetected indefinitely. However, this is exceedingly unlikely in practice. The probability of detecting a crashed machine in a single round of pings is about 63% for clusters with 100 or more nodes; the odds are greater than 99% that a failed server will be detected within five rounds.

As an alternative, the coordinator could periodically ping all cluster members to ensure their liveness. The current coordinator can broadcast to 80 machines in 600 µs by using low-latency Infiniband messaging and by sending pings in parallel. We expect this approach to scale to broadcast to 10,000 nodes in about 75 ms, so it is plausible for the coordinator to monitor the liveness of a large cluster. However, the coordinator would need to keep its network card mostly utilized to detect failures quickly, leaving little capacity for other operations.

Having servers randomly ping one another reduces the load on the coordinator, improves failure detection time, and scales naturally as the cluster grows. Since the work of pinging the servers is divided up, servers can afford to ping each other more frequently than a single coordinator could in a large cluster. Server are pinged much more frequently

than every 75 ms; with high probability each server is pinged every 100 to 500 µs. Consequently, RAMCloud speeds the failure detection of idle servers without substantially increasing coordinator load by distributing the work of monitoring for failures across the cluster; this is another example of how RAMCloud uses randomized techniques to avoid centralized bottlenecks that could limit scalability.

Short failure detection timeouts also present some potential pitfalls. For example, overly hasty failure detection introduces a risk that RAMCloud will treat performance glitches as failures, resulting in unnecessary recoveries. The load of these unnecessary recoveries could destabilize the system further by causing additional recoveries. In the worst case, the result would be a sort of "recovery storm" where recoveries would trigger waves of cascading failures. Currently, it is unclear whether this will be a problem in practice, so it is left to future work. One potential way to mitigate this threat is to have the coordinator give servers a second chance before recovering them. In this approach, the coordinator would ping a server marked for recovery one final time just before recovering it. If it responded to the ping, then it would be restored to normal status in the cluster without undergoing recovery. This would give misbehaving servers more time to respond to pings when the cluster is already performing another recovery, since the coordinator performs recoveries one-at-a-time. This approach might be effective in fending off recovery storms, since the grace period given to a misbehaving server would be proportional to the number of concurrently failed servers in the cluster.

Another potential pitfall is that short timeouts may preclude the use of some network protocols. For example, most TCP implementations wait 200 ms before retransmitting lost packets. Since RAMCloud pings are configured to timeout more quickly, a single lost packet could lead to a spurious failure detection. RAMCloud supports TCP, but it must use a long failure detection interval, which lengthens unavailability on failures. Other transports, such as Infiniband, support lower timeouts and faster failure detection, and RAMCloud provides an IP-capable transport similar to TCP that also supports lower timeouts.

5.3 Setup

Once the coordinator has verified that a master has failed, it enters the setup phase of master recovery. During this phase, the coordinator gives each server all of the information it needs to perform its share of the work of recovery, so once this phase completes, recovery proceeds independently of the coordinator. The coordinator finds the log segment replicas left behind by the crashed master, verifies that the log is undamaged and safe to use for recovery, divides up the work of recovery into equal size partitions, and distributes the work of recovery across several of the servers in the cluster.

5.3.1 Finding Log Segment Replicas

As described in Section 4.4.1, a master does not keep a centralized list of the segments that comprise its log. This prevents the coordinator from becoming a bottleneck during normal operation. Each master tracks where its segments are replicated, but this information is lost when it crashes.

The coordinator discovers the locations of the crashed master's replicas by querying all of the backups in the cluster using `findReplicasForLog` (§4.4). The coordinator aggregates the locations of the consistent replicas into a single map.

This query broadcast is fast; the coordinator uses RAMCloud's fast RPC system and queries multiple backups in parallel. The current coordinator implementation can broadcast to 80 machines in 600 µs. Scaling this approach to 10,000 nodes would result in broadcast times of about 75 ms, so in large clusters it may make sense to structure broadcasts in a tree-like pattern.

The coordinator broadcast is also used to give an early start to disk reading operations on backups, so its overhead is mitigated. Backups start reading replicas from storage immediately after they reply with a replica list. Even as the coordinator finishes setup, backups are already reading the data from storage that the recovery masters will request during the replay phase.

5.3.2 Detecting Incomplete Logs

After all of the backups return their lists of replicas, the coordinator must determine whether the reported segment replicas form the entire log of the crashed master. Redundancy makes it highly likely that the entire log will be available, but the system must be able to detect situations where data is missing (for example, during massive failures such as network partitions or power outages). As described in Section 4.4.2, backups return a log digest to the coordinator during `findReplicasForLog` that is used to verify whether the set of replicas found constitutes a complete and up-to-date view of the log from the failed server.

In the unlikely case that `findReplicasForLog` cannot find a complete log, the coordinator aborts and retries the recovery. In the meantime, if crashed servers rejoin the cluster, then a future recovery for the server may succeed. If replicas for some of the segments are persistently unavailable, then the user can drop the affected tablets, which will cause the coordinator to give up retrying the recovery. Another option would be to allow the user to permit recovery with an incomplete log. The user would have no guarantees on the completeness or consistency of the data after recovery, and RAMCloud currently provides no interface for this.

For safety and consistency, each recovery master must restrict the replicas it replays to those that are part of the coordinator's aggregated replica location list. The coordinator ensures that the replicas in its list are all consistent and represent the entire log of the crashed master (§4.4). At the end of the setup phase, the coordinator provides this list to each recovery master (§5.3.4), and recovery masters only replay replicas from the list. Even if new servers with additional replicas are added to the cluster during recovery, recovery masters must not use replicas stored on them, since some of them may be stale.

5.3.3 Grouping Tablets Into Partitions

Next, the coordinator must divide up the work of recovery among a set of recovery masters. The coordinator does this by grouping the tablets of the crashed master into *partitions*. Then, the coordinator assigns each partition to exactly one recovery master for recovery (§5.3.4). Later during the replay phase of recovery, each recovery master replays the objects from the crashed master's log that are part of its partition and re-replicates them as part of

its own log. When recovery completes, the recovery masters become the new home for the tablets in their assigned partitions.

RAMCloud partitions log data entirely at recovery time rather than having masters divide their data during normal operation. Waiting until recovery time ensures that the work of splitting log data across into partitions is only done once and only when it actually matters: at the time the server has crashed. Furthermore, delaying partitioning allows the partition scheme to be based on the final distribution of the object data across the crashed master's tablets. Another benefit is that the coordinator could factor in the global state of the cluster. For example, the coordinator could use server availability or load information in determining partitions. The current implementation does not factor global cluster state into partitioning.

The coordinator's goal in determining the partitions is to balance the work of recovery such that all recovery masters finish within the 1 to 2 second recovery time budget. Two key factors determine the time a recovery master needs to perform its share of the recovery. First, recovery masters will be limited by network bandwidth for large partitions. Second, recovery masters will be limited by cache misses for hash table inserts for partitions with many objects. As a result, the coordinator must take care to ensure that no single recovery master is saddled with recovering a partition that is too large or that has too many objects. What constitutes "too much" or "too many" is determined experimentally (§6.3) by benchmarking recovery masters. For our machines, each partition is typically limited to 500 MB to 600 MB and 2 to 3 million objects.

In order to create a balanced set of partitions, the coordinator must know something about the tablets that were on the crashed master. One possibility would be for each master to record tablet statistics on the coordinator when the balance of data in its tablets shifts. However, RAMCloud's high performance means tablet sizes can shift quickly. A master may gain 600 MB of data in about 1 second and data can be deleted even more quickly (an entire 64 GB server can be emptied in a fraction of a second). With a centralized approach in a large cluster, tablet statistics on the coordinator might be updated more than 10,000 times per second.

Instead, masters rely on decentralized load statistics that they record as part of their log along with the log digest in each segment. Whenever a master creates a new segment, it

records the number of objects and the number of bytes in each of its tablets at the start of the segment. However, a single master may contain millions of small tablets. To prevent tablet statistics from using a substantial fraction of the log (and DRAM), statistics are only approximate after the largest two thousand tablets on each master; the remaining tablets are summarized as one record containing the total number of objects and bytes in the remaining smaller tablets along with a count of how many smaller tablets there are. These statistics are recovered from the log along with the log digest during the `findReplicasForLog` operation at the start of recovery (§4.4). A log digest is always necessary for recovery, so this approach is guaranteed to find a usable statistics block if recovery is possible.

Since statistics may be summarized if there are more than two thousand tablets on a master, partitioning will not always divide data perfectly. In practice, the error from this optimization is minimal. For example, if a 64 GB master with more than 2,000 tablets is divided into 600 MB partitions, then on average each full partition will have 20 tablets or more. The coordinator groups partitions using a randomized approach, so the likelihood that any single partition is saddled with many of the largest tablets from those with averaged statistics is low. Even masters with millions of tablets do not pose a problem; likelihood of imbalance goes down as more and smaller tablets are grouped into partitions.

After acquiring the tablet statistics from the log, the coordinator performs its randomized bin packing algorithm to split work. The coordinator considers each tablet in turn. For a tablet, the coordinator randomly chooses five candidate partitions from among set that remain underfilled (that is, those that would take less than 1 to 2 seconds to recover given their current contents). For each candidate partition, the coordinator computes how full the partition would be if the tablet under consideration were added to it. If adding the tablet would result in a partition that would take too long to recover (that is, it does not "fit"), then it tries the next candidate. If the tablet fits, then it is added to the partition. If the partition becomes full enough, then it is removed from consideration in future rounds. If the tablet does not fit into any of the candidate partitions, then a new partition is created, the tablet is added to it, and it is added to the set of potential candidate partitions. The algorithm initially starts with an empty set of partitions, and terminates when all of the tablets have been placed in a partition.

This randomized bin packing will not produce optimal results; with high probability it

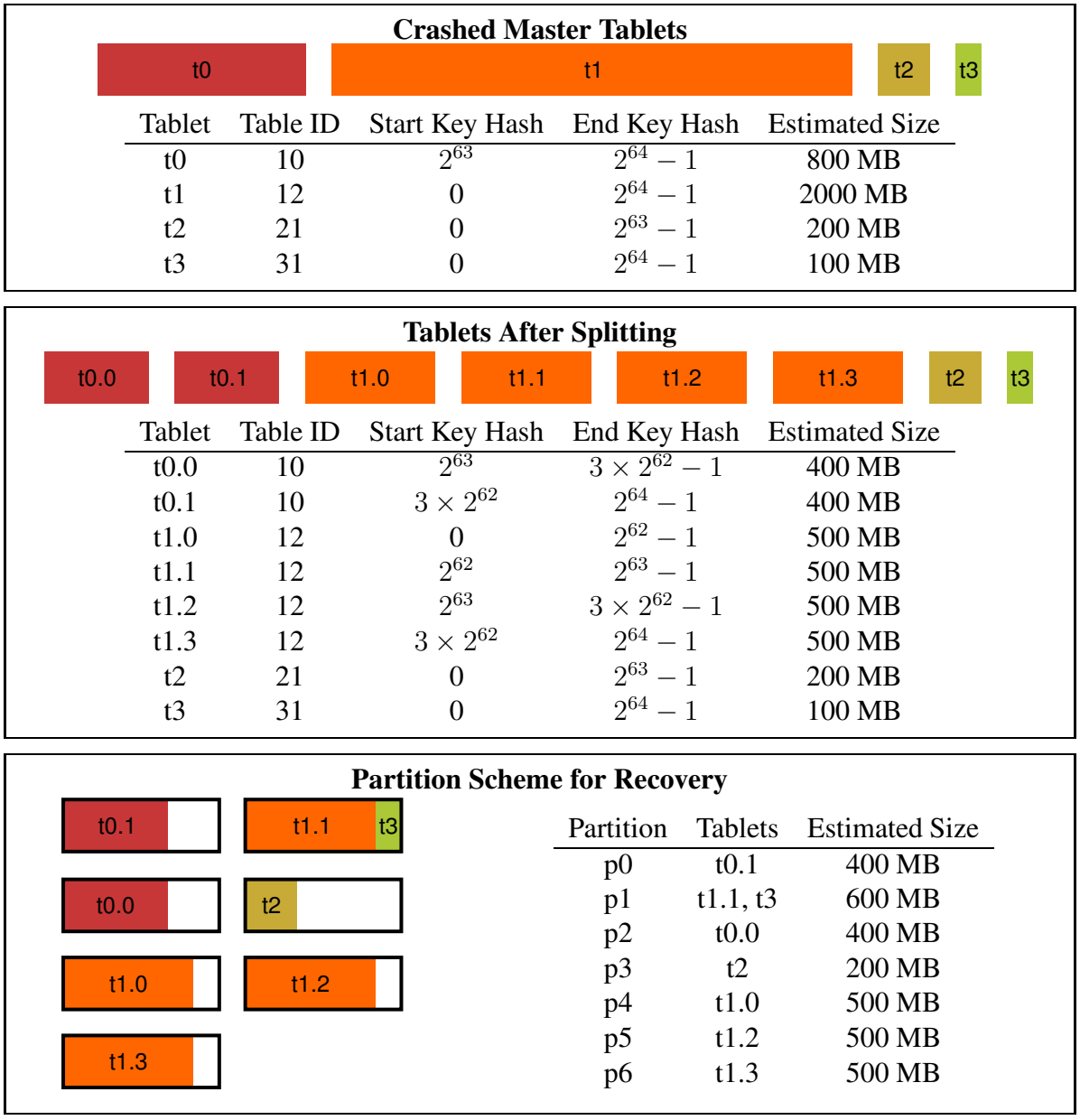


Figure 5.2: The coordinator uses statistics from a crashed master's log to partition tablets. The top box shows the crashed master's estimated tablet statistics from its log. The middle box shows the tablets after tablets that are too large to be recovered by a single recovery master are split. The bottom box shows the result of the random partition packing. Here, the coordinator limits partitions to 600 MB of object data; in practice, the coordinator limits the number of objects in each partition as well. This run resulted in seven partitions, four tablet splits, and 1200 MB of wasted capacity; an ideal packing would have resulted in six partitions, four splits, and 500 MB of wasted capacity.

will spread recovery across a few more partitions than strictly necessary. However, spreading recovery across a few additional recovery masters has little impact on recovery time. What is essential is that partition size is tightly bounded, which this algorithm ensures; otherwise, some recovery masters would drag out recovery time.

Some tablets may be too large to be recovered quickly even when placed alone in an empty partition. To solve this, the coordinator performs a splitting pass over all of the tablets before grouping them into partitions. Each tablet is broken into the minimum number of equal-sized tablets that meet the size and object count constraints. In splitting tablets, the coordinator approximates good places to split by using the statistics collected from the log and by assuming all objects and object data are spread uniformly across the tablet by its key hash function. Figure 5.2 illustrates an example of how the coordinator splits and partitions tablets.

One potential pitfall of splitting tablets during recovery is that, without care, repeated failures might lead to compounded splitting that would explode the number of tablets in the cluster. Left unchecked, this could create a burden on clients, since it would increase the number of tablet-to-master mappings they would need to access data. Fortunately, minimizing the number of splits per tablet and equalizing the size of the resulting tablets mitigates this problem. One key property shows that cascading splits will not happen unless intervening tablet growth requires it: whenever a tablet is split, each of the resulting tablets must be at least half the size of the partition size (otherwise, it would have fit in a single partition to begin with). This holds regardless of the number of times a tablet undergoes recovery: a tablet will never be split across more than twice the optimal number of partitions needed to contain it. This same argument holds for tablets that are split due to the amount of data they contain and for tablets that are split due to the number of objects they contain.

5.3.4 Assigning Partitions to Recovery Masters

After determining the partitions that will be used for recovery, the coordinator assigns each recovery master a partition with a `recover` RPC. As part of the RPC, each of recovery master receives two things from the coordinator:

- a partition that contains the list of tablets it must recover, and

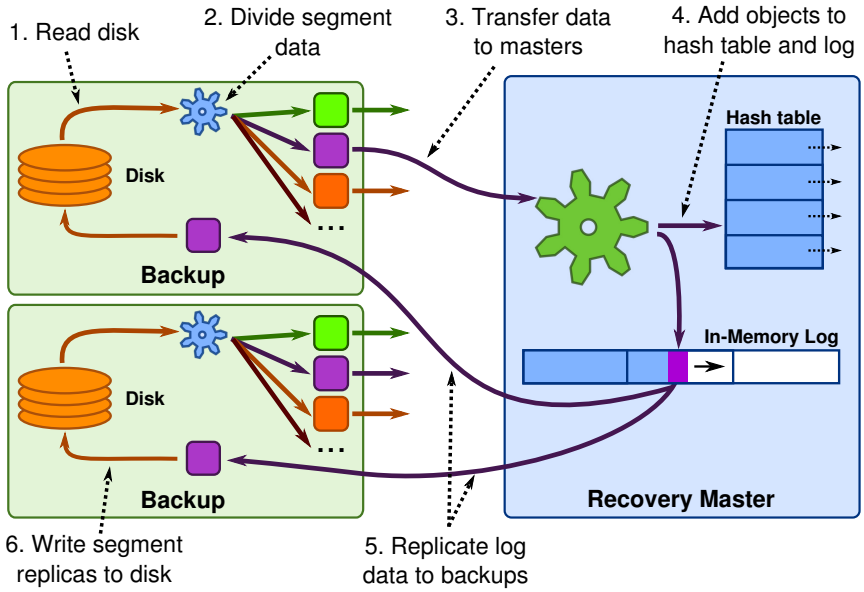


Figure 5.3: During recovery, data in replicas flows from disk or flash on a backup over the network to a recovery master, then back to new backups as part of the recovery master's log.

- a replica location list that contains the locations of all the crashed master's log segment replicas.

It is possible that the coordinator's partitioning algorithm results in more partitions than the number of available masters. In this case, just one partition is assigned to each of the available masters. The remaining partitions are left marked as failed and are recovered in a subsequent recovery (failed recovery masters are handled in the same way, as described in Section 5.6.2).

5.4 Replay

The vast majority of recovery time is spent replaying segments on the recovery masters. During replay, log entries are processed in six stages (see Figure 5.3).

1. A replica is read from disk into the memory of a backup.
2. The backup divides the object records in the replica into separate groups for each partition based on the table id and the hash of the key in each log record.

3. The records for each partition are transferred over the network to the recovery master for that partition. The recovery master fetches records from backups as it becomes ready for them to prevent backups from inundating it.

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 backup writes the new segment replicas from recovery masters to disk or flash.

RAMCloud harnesses concurrency in two dimensions during recovery. The first dimension is data parallelism: different backups read different replicas from disk 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 concurrently, with segment replicas as the basic unit of work. While one replica is being read from disk on a backup, another replica is being partitioned by that backup's CPU, and records from another replica are being transferred to a recovery master; similar pipelining occurs on recovery masters. For fastest recovery all of the resources of the cluster should be kept fully utilized, including disks, CPUs, and the network.

5.4.1 Partitioning Log Entries on Backups

The objects in each segment replica must be bucketed according to partition during recovery, because each master intermingles all of the objects from all of its tablets in each segment. Backups use the partitioning scheme created by the coordinator and help out recovery masters by grouping object data as it is loaded from replicas on disk. As a result, each recovery master can quickly fetch just the objects it is responsible for replaying. Overall, deferring partitioning and spreading the work of partitioning across all of the backups of the cluster saves network bandwidth and CPU load on masters during normal operation; masters never have to perform any kind of rebalancing or bucketing. Furthermore, backups vastly outnumber recovery masters in a RAMCloud deployment, so the load of partitioning is spread across all of the nodes of the cluster. Partitioning is fully pipelined with the rest of

recovery and proceeds in parallel with disk operation and data replay on each node, which mitigates the penalty of waiting until recovery time to partition.

One drawback to partitioning log data at recovery time is that a crashed master's objects must be rewritten into recovery masters' logs. If objects were not rewritten, then if one master recovered part of another master and crashed, then the logs for both servers would have to be replayed for recovery. This would slow recovery times and would make it difficult to reclaim replica storage. Rewriting the objects essentially persists the data in its partitioned form, which ensures that crashes are always recoverable using a single log. The burden of rewriting data to storage is mitigated if backups have enough non-volatile buffer capacity in aggregate to store the entire set of new replicas generated during a recovery (by default, about three times the size of the crashed master's DRAM).

As an alternative to partitioning at recovery time, masters could do it during normal operation, as they log objects. Each master could group its tablets into partitions and place objects from different partitions into different logs. When the master crashed, each recovery master would only have to replay the replicas from the log for its assigned partition. This also has the advantage that recovery masters would not have to rewrite log data during recovery; instead, the could adopt the already-partitioned replicas left behind by the crashed master as part of their log. Unfortunately, this approach has two disadvantages. First, masters would actively replicate roughly one hundred segments at a time during normal operation (one for each partition). This would require 100 times more non-volatile DRAM buffering on backups than having all objects appended to a one log. Second, partitioning objects when they are first written to the master's log makes it difficult to repartition later. In RAMCloud, tablet sizes can change rapidly since each master supports hundreds of thousands of operations per second. It would be costly for masters to frequently "repack" objects in their log to accommodate repartitions due to changing tablet sizes.

5.4.2 Segment Replay Order

In order to maximize concurrency, recovery masters and backups operate independently. As soon as the coordinator contacts a backup to obtain its list of replicas, the backup begins prefetching replicas from disk. Almost immediately afterward, when the coordinator has

determined the partitions it will use for recovery, masters begin fetching segment data from backups and replaying it. Ideally, backups will constantly run ahead of masters, so that data from a replica 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 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.

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 (§4.3) to estimate when each replica's data is likely to be ready, and it requests data in order of expected availability.

This approach causes all masters to request segments in the same order, which could lead to contention if masters persistently work in a lock-step fashion. To solve this, servers could randomize their requests for data to spread load. One simple approach would be to use an approach similar to the randomness with refinement used in segment scattering. When deciding which replica data to request, a recovery master would choose a few candidate servers randomly; among candidates a recovery master would send a request to the candidate that is most likely to have unreplayed data ready. The use of refinement would ensure that each recovery master replays data from each of the backups at an even rate, while the use of randomness would avoid hot spots. In in our current cluster, convoying does not appear to be a bottleneck, so this optimization remains unimplemented.

Unfortunately, even with replicas read in a fixed order, there will still be variations between backups in the speed at which they read and process replicas. In order to avoid stalls because of slow backups, each recovery master keeps several concurrent requests for data outstanding at any given time during recovery. The current implementation keeps four requests outstanding to backups; this is enough to hide network latency and compensate for when a few backups are slow in responding without causing unnecessary queuing delays. Recovery masters replay data in whatever order the requests return.

Because of the optimizations described above, recovery masters will end up replaying replicas in a different order than the one in which the segments were originally written.

Fortunately, the version numbers in log records allow the log to be replayed in any order without affecting the result. During replay, each recovery master simply retains the version of each object with the highest version number, discarding any older versions that it encounters.

Although each segment has multiple replicas stored on different backups, only the primary replicas are read during recovery; reading more than one would waste valuable disk bandwidth. Masters identify one replica of each segment as the primary replica when scattering their segments as described in Section 4.3. During recovery, each backup reports all of its replicas, 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 them only on demand. This typically happens when the backup that contained a primary replica crashes, or if a primary replica is corrupted on storage. Each non-primary replica is cached in DRAM after it is first accessed, since all of the recovery masters will most likely need it.

Since I/O for non-primary replicas is done on demand, in the worst case, recovery time could be extended 30 to 80 ms for every non-primary replica accessed. In practice, this is unlikely for two reasons. First, depending on when a non-primary replica is accessed during recovery it may not impact recovery time at all, since recovery can work ahead on other segments and come back to replay data from the non-primary replica later. Second, if the primary replicas for more than one segment have been lost, then several non-primary replicas may be requested (and, thus, loaded) in parallel from different backups.

Recovery masters use the list of replica locations provided by the coordinator to determine when they have replayed the entire log of the crashed master. The coordinator verifies that the replica location list contains at least one replica from every segment in the crashed master's log. A recovery master finishes replay when at least one replica for every segment mentioned in the replica location list has been replayed. Section 5.6.2 describes what happens when this is impossible due to failures or data corruption.

5.5 Cleanup

After a recovery master completes the recovery of its assigned partition, it notifies the coordinator that it is ready to service requests. The coordinator updates its tablet configuration information to indicate that the master now owns the tablets in the recovered partition, at which point the partition is available for client requests. Clients with failed RPCs to the crashed master have been waiting for new configuration information to appear on the coordinator; they discover it 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 broadcasts its removal to the remaining servers of the cluster. Backups listen for these events and remove any recovery state (in-memory partitioned replica data, etc.) as well as the on-disk replicas for the recovered server as part of the log replica garbage collection mechanism described in Section 4.6.

5.6 Coordinating Recoveries

5.6.1 Multiple Failures

Given the large number of servers in a RAMCloud cluster, there will be times when multiple servers fail simultaneously. When this happens, RAMCloud recovers from each failure independently. The only difference in recovery is that some of the primary replicas for each failed server may have been stored on the other failed servers. In this case, the recovery masters will need to use secondary replicas for some segments, but recovery will complete as long as there is at least one replica available for each segment.

RAMCloud can recover multiple failures concurrently; for example, if a RAMCloud cluster contains 5,000 servers with flash drives for backup, the measurements in Chapter 6 indicate that a rack failure that disabled 40 masters, each with 64 GB of DRAM, could be recovered in about 2 seconds. Multiple simultaneous recoveries are supported but must be explicitly enabled; by default, the coordinator only performs one recovery at a time.

If many servers fail simultaneously, such as in a power failure that disables many racks,

RAMCloud may not be able to recover immediately. This problem arises if no replicas are available for a segment or if the remaining masters do not have enough spare capacity to take over for all the failed masters. In this case, RAMCloud must wait until enough machines have rebooted to provide the necessary data and capacity. This happens naturally, since the coordinator repeatedly retries a recovery until it succeeds; if some replicas are unavailable the coordinator will hold back recovery until they can be found. RAMCloud clusters should be configured with enough redundancy and spare capacity to make situations like this rare.

5.6.2 Failures During Recovery

Nearly all failures and exceptional conditions that occur during recovery are solved by aborting all or part of a recovery and retrying a new recovery for the tablets later. Each recovery master works independently and may succeed or fail at recovering its assigned partition without affecting the other recovery masters. A recovery master can give up on recovery if it is impossible to complete, and the coordinator treats any crash of a recovery master as an implicit recovery abort for that recovery master. As a result, some or all of the tablets of the crashed master may still be unavailable at the end of a master recovery.

To remedy this, the coordinator repeatedly attempts to recover tablets from failed servers until all tablets are available. Whenever a problem that prevented recovery from succeeding is corrected, a follow-up recovery attempts restore availability shortly thereafter. Furthermore, a partition is likely to be assigned to a different recovery master in a follow-up recovery, which resolves problems that may be specific to a single recovery master.

The coordinator currently only performs one master recovery at a time, so recoveries do not always start right away when a server fails. Instead, the coordinator enqueues the new recovery and delays its start until the ongoing recovery completes. The queue of recoveries is serviced in first-in-first-out (FIFO) order, so a repeatedly failing recovery for one master cannot block recovery for another master.

Whenever a recovery master wishes to abort an ongoing recovery, it must undo some of the effects of the recovery to ensure safety after future recoveries. Specifically, aborting the recovery of a partition requires a recovery master to scan its hash table and remove entries

for keys that were part of the partition that it failed to recover. This prevents leftover stale entries in the hash table from causing consistency issues if the recovery master is later assigned ownership of some of the keys from the partition it partially recovered. Removing the hash table entries makes the log data left behind by the aborted recovery eligible for log cleaning, so these entries eventually get cleaned up in the usual way.

Some of the many failure scenarios that can arise during recovery are discussed below; many rely on the atomic and abortable nature of recovery on each individual recovery master.

- **A server crashes during recovery.** When a server crashes it creates two sets of problems since each server acts as both a master and a backup.
  - **Handling the loss of a backup.** Since each segment is replicated, recovery can succeed even with the loss of some backups. In the case that a backup that contained the last remaining replica of a segment crashes during recovery, any recovery master that had not yet replayed the objects from that segment must abort the recovery of its partition. This does not preclude other recovery masters from recovering their partitions successfully, depending on the timing of the loss of the final replica.
  - **Handling the loss of a master.** In this case, a recovery is enqueued for the newly crashed master, which will be performed after the ongoing recovery. If the master that crashed during an ongoing recovery was acting as a recovery master, then it will also fail to recover its assigned partition. In this case, the coordinator enqueues a new recovery for the master whose partition was not successfully recovered. When this new recovery is started, the coordinator only attempts to recover tablets that were not already successfully recovered in the prior recovery. In total, when a recovery master fails two new recoveries are enqueued (in addition to the ongoing recovery): one for the tablets owned by the failed recovery master and one to retry recovery for the tablets of the master that was only partially recovered.

- **A recovery master runs out of memory.** If a recovery master runs out of log space recovering a partition, it aborts the recovery and reports the failure to the coordinator. Just as when a recovery master crashes, the coordinator schedules a follow-up recovery to retry the recovery of the partition elsewhere in the cluster.

- **A segment replica is corrupted or misidentified.** Each segment replica's metadata and identifying information is protected by a checksum stored along with it on disk. This checksum covers the master and segment ids, the replica length, and a bit indicating whether its log digest is open or closed. It also covers the length and type of all of the log records in the replica, but it does not include the contents of the log entries themselves (they have individual checksums). Verifying a replica's checksum ensures it can be identified and iterated over safely, but objects must be individually verified against their checksums to ensure they have not been corrupted. Backups discard all data from replicas whose checksums do not match their metadata. Recovery masters automatically request data for discarded replicas from other backups as needed.

  Backups never verify object checksums; instead an object's checksum is only verified just before it is replayed. This prevents an object from passing a check on the backup, only to be corrupted in transit to the recovery master. When an object with a corrupted checksum is encountered, the recovery master aborts the replay of the remainder of the replica, and it tries to fetch another replica of the same segment on another backup. The version numbers on log records (§5.4.2) ensure replaying the same replica multiple times is safe.

- **A recovery is split across more partitions than there are masters available.** Such a situation implies the existing cluster is too small to meet the 1 to 2 second recovery goal. In this case, every master in the cluster is assigned a single partition to recover, and the remaining unassigned partitions are left marked as needing recovery. A follow-up recovery is enqueued and is started immediately following the completion of the first recovery. By iteratively retrying recovery, each of the available masters in the cluster will recover some fraction of the failed server's tablets every 1 to 2 seconds until all of its tablets have been returned to service.

- **No backups have storage space left.** When backups run out of space either during recovery or during normal operation, they begin to reject segment replication requests from masters. As a result, operations stall until new backups join the cluster with additional storage capacity or until the user frees space by deleting objects or dropping tablets.

  In the current implementation, it may be dangerous to allow backups to fill to capacity, since deleting objects requires new log records to be written. If recovery were to fill backups completely, it may become impossible to release storage space and resume operation. One fix to this is to have recovery masters abort recovery if they take too long and receive too many backup replication rejections. In-memory log space used by the recovery objects would become reclaimable, and the log cleaner on the recovery masters would free space on the backups. In practice, it will be important for operators to ensure that RAMCloud has adequate disk space to perform recoveries. An operator user interface that can aid with these types of administrative tasks is left for future work.

- **The coordinator crashes during recovery.** The coordinator persists its critical state to a consensus-based high-availability storage system [25], and RAMCloud uses coordinator standbys to take over if the active coordinator crashes. When a coordinator crashes, a leader election chooses a new coordinator process, which reloads the critical cluster state and resumes operation. However, coordinators do not persist information about ongoing recoveries. Instead, they mark a server (and its tablets) as failed in their persistent server list. Newly elected coordinators always restart master recoveries based on server and tablet status information left behind by the previous coordinator. When a new coordinator is elected it scans its server list to find any crashed and unrecovered servers, and it enqueues a recovery for each of them. As usual, it performs the recoveries one-at-a-time.

  With this approach, it is possible that the cluster may still be running a master recovery started by a failed coordinator process when a new coordinator takes over and starts a competing recovery for the same master. The new coordinator must ensure only one of these recoveries succeeds, since it is imperative for consistency that each

tablet is only served by one master at a time. Coordinators solve this by tagging each master recovery with a randomly generated id. When recovery masters complete their portion of recovery, they report to the current coordinator using id of the recovery they are participating in. If the coordinator has failed in the meantime, it will have no record of a master recovery with the reported id, so it notifies the recovery master that it should abort the recovery instead of servicing requests for the data. In this way, ongoing recoveries are neutralized after a coordinator rollover.

A second, less serious problem arises since a newly elected coordinator may try to assign a partition to recovery master that is already participating in an existing (doomed) recovery. If a recovery is already ongoing on a recovery master, the recovery master should abort its ongoing recovery and start work on the new recovery. Currently, this is unimplemented, since recovery masters block while performing a recovery; recovery may block for a few seconds after a coordinator rollover until preoccupied recovery masters complete their ongoing recovery (necessarily, ending in abort).

5.6.3 Cold Start

On rare occasion, an entire cluster may fail or need to be restarted, such as during a power outage. In these cases, RAMCloud must guarantee the durability of the data that was written to it, and it must gracefully resume service whenever servers begin restarting.

When the coordinator comes back online, it proceeds with normal coordinator recovery, and it begins continuously retrying recovery for each of the logs left behind by the former masters. When servers restart, they retain all replicas they find on disk until the objects in them are recovered and it can safely determine they are no longer needed (§4.6). When the first few servers come back online, recovery will be impossible, since some of the segments will be missing for every log that needs recovery. Once enough servers have rejoined the cluster, the coordinator's recovery attempts will begin to succeed and tablets will begin to come back online.

RAMCloud uses its normal master recovery during cluster restart, but it may eventually make more sense to use a different approach where masters are reconstructed exactly as

they existed before the cold start. When restarting all servers, the coordinator could recover each crashed master as a single unit, rather than partitioning its tablets. Thus, data in a crashed master's replicas would not need to be partitioned, and ownership of its replicas could be given directly to the recovery master. In effect, the recovery master could load the log replicas into DRAM and then resume operations on the log, as if it had created them. Consequently, recovery masters would avoid the need to recreate replicas of the crashed masters' data. Such an optimization would reduce disk traffic and cold start time by a factor of four. In the end, we choose to use the existing recovery mechanism, which fits with the goal of minimizing and collapsing error cases during recovery. For example, transferring log ownership becomes complicated when servers experience repeated failures during cold start (see §5.4.1). To make the approach viable, RAMCloud would need to be extended to support logs comprised of replicas written with multiple log ids, its server ids would have be adjusted to support reuse, or the coordinator and backups would need to support a cross-backup, multi-replica, atomic server id change operation.

5.7 Zombie Servers

RAMCloud seeks to provide a strong form of consistency (linearizability [22]), but server failures and network partitions make this a challenge. One key invariant RAMCloud uses to ensure strong consistency is that requests for a particular object in are always handled by at most one master at a time. If one master performs all operations for a particular object at a time, then it can provide atomicity and strong consistency by ordering all operations on each object. Unfortunately, due to network partitions, the coordinator may not be able to reach a master even while it continues to service client requests. A master that is suspected of failure may continue to operate as a "zombie", and it must stop servicing requests before it can be safely recovered. Otherwise, if the contents of a master were recovered elsewhere while it continued to service requests, then clients could read and write differing and inconsistent views of the same keys.

RAMCloud uses two techniques to disable operations by these zombie masters: one for write operations and another for read operations. To prevent a zombie master from servicing a client write request, the coordinator broadcasts to the cluster indicating that the

master is entering recovery. Backups refuse all replication operations from masters that are under recovery, so any requests from the master are rejected by backups afterward. Whenever a master's replication request is rejected in this way, it stops servicing requests, and it attempts to contact the coordinator to verify its status. The master continuously tries to contact the coordinator until it is successful, and its other operations remain suspended. When the master successfully contacts the coordinator, if recovery for the master has already started, then the master commits suicide: it terminates immediately. Otherwise, it resumes normal operation. This approach ensures that by the time recovery masters begin replaying log data from a crashed master, the master can no longer complete new write operations.

Under correlated failures and network partitions, the coordinator may not be able to contact all of the backups of the cluster to prevent a zombie master from replicating new log data. A zombie master may continue to replicate data to its head segment if the backups holding the segment's replicas could not be contacted by the coordinator. Fortunately, if this case arises, then recovery would not proceed on the coordinator; the coordinator would be missing all the replicas for the head segment, so recovery would be reattempted later.

RAMCloud uses its distributed failure detection mechanism to help ensure that a zombie master stops servicing client read requests before a recovery for its tablets completes. Recall that each server occasionally pings another randomly-chosen server in the cluster for failure detection. A master uses the information it gains from these pings to quickly neutralize itself if it becomes disconnected from the cluster. As part of each ping request, the server sending the ping includes its server id. If the sending server is not marked as up in the server list on the receiving server, then a warning is returned. Just as with rejected backup replication requests, a server begins deferring all client operations and checks in with the coordinator when it receives a warning. If recovery has already started for the master, then it terminates immediately. Otherwise, it resumes normal operation. Similarly, if a server is separated from the cluster and cannot contact any other servers, then the server suspects itself to be a zombie and takes the same steps by checking in with the coordinator.

This approach to preventing stale reads is only probabilistic; groups of servers that are disconnected may continue to ping one another, even after the coordinator has detected their failure and started recovery. RAMCloud's goal in this respect is to make the probability

of stale reads no more likely than the probability of silently losing data (which would also constitute a violation of linearizability). The probability of a client experiencing stale reads is highly unlikely in practice, since several rounds of pings will have occurred by the time recovery completes for one of the disconnected servers. If a substantial fraction of the cluster is disconnected, then the coordinator cannot perform recovery anyway (all of the replicas for some segment are likely to be in the disconnected set of servers). If a small fraction of the cluster is disconnected, then each disconnected server is likely to check in with the coordinator quickly, since most of ping attempts to it and from it will fail.

Neutralizing zombie masters strengthens consistency but is insufficient alone to provide linearizability; two other guarantees are needed, one of which remains unimplemented. First, when recovering from suspected coordinator failures, RAMCloud must ensure that only one coordinator can manipulate and serve the cluster's configuration at a time. Without this guarantee, independent coordinators could assign the same tablets to different masters, again creating different and inconsistent views of some of the keys. This problem is prevented by using the consensus-based ZooKeeper [25] for coordinator election. Second, linearizability requires exactly-once semantics, which requires support from the RPC system that remains unimplemented in RAMCloud.

5.8 Summary

When a master crashes in RAMCloud, the data stored in its DRAM becomes unavailable. RAMCloud uses fast crash recovery to restore availability by reconstituting the data from the failed server's DRAM in 1 to 2 seconds. The key to fast recovery is to leverage scale by dividing the work of recovery across all of the nodes of the cluster. Recovery employs all of the CPUs, NICs, and disks of the cluster to recover quickly.

Fast crash recovery depends on fast failure detection; RAMCloud nodes use randomized and decentralized probing to speed the detection of crashed nodes.

At the start of recovery, the coordinator broadcasts to find replicas of log data for the crashed master. It verifies that the set of replicas it discovers constitutes a complete copy of the log by checking it against a log digest that is extracted from the head segment of the crashed master's log. The coordinator uses statistics extracted from replicas and a

randomized bin packing algorithm to divide the crashed master's tablets into roughly equal sized partitions. Each partition is assigned to a separate recovery master.

During the replay phase, which is the main portion of recovery, backups load replica data and bucket objects according to partition. At the same time, each recovery master fetches objects from backups relevant to the tablets it is assigned to recover. It copies these into its in-memory log, updates its hash table, and re-replicates the log data to backups. After a recovery master has replayed objects from all of the segments left behind by the crashed master, it begins servicing requests for its assigned tablets. The replay phase is highly parallel and pipelined; each node is reading data from disk, partitioning objects found in replicas, fetching data from backups, updating its hash table, and re-replicating objects to other nodes in the cluster all at the same time.

Most failures that can happen during recovery are handled by cleanly aborting recovery and retrying it again later. Recovery safely deals with the loss of masters or backups, the loss or corruption of replica data, resource exhaustion, and coordinator crashes. The coordinator must be sure a master has stopped servicing requests before the master can be safely recovered. The coordinator disables a zombie master's writes by notifying backups that the master is dead, and it reuses the decentralized failure detector to quickly stop zombie masters from servicing read requests. Cold start is handled by iteratively applying the single master recovery mechanism.

Overall, fast crash recovery quickly restores availability and full performance to the cluster after a server crash, rather than relying on expensive in-memory redundancy or operating in a degraded state using data on disk. Its short recovery time is no longer than other delays that are common in normal operation, so recovery is unnoticeable to applications. Fast crash recovery is the key to providing continuous availability in RAMCloud.

Chapter 6

Evaluation

We implemented and evaluated the RAMCloud architecture described in Chapters 3 through 5. Ideally, evaluation would answer the question, "can a realistic RAMCloud deployment reliably restore access to the lost DRAM contents of a full-size failed server in 1 to 2 seconds?"

Unfortunately, this question cannot be answered directly without hundreds of nodes, so instead we focus on evaluating the scalability of crash recovery. If RAMCloud can recover quickly from increasingly large failures as cluster size grows, then it should be possible to recover the entire contents of a failed server in a sufficiently large cluster. Any design will have limits to its scalability, so it is reasonable to expect that as the cluster grows recovery bandwidth will see diminishing gains. Then, the key questions are, "how much data can be recovered in 1 to 2 seconds as cluster size scales," and "what are the limits of recovery scalability?"

Each section of this chapter addresses a different question related to the performance of recovery, mostly aimed at understanding its scalability. Some of the key questions and related results are summarized here.

How well does recovery scale? (§6.4)

A crashed server with 40 GB of data can be recovered in 1.86 seconds, which is only only 12% longer than it takes to recover 3 GB with 6 servers. As a result, we expect that the entire contents of a 64 GB crashed master could be recovered in about 2 seconds with a cluster of 120 machines similar to those used in our experiments.

What is the overhead of adding nodes to the cluster? (§6.4)

The coordination overhead of adding a node to recovery is only about 600 µs, so the current implementation should scale to spread recovery across hundreds of nodes.

What limits recovery times in small clusters? (§6.4.1)

In our 80-node cluster, recovery times are limited by our CPUs, since each backup must perform many times more work than in an expected RAMCloud deployment. During recovery, CPU utilization and memory bandwidth remain saturated at nearly all times. Our CPUs only provide 4 cores with a single memory controller; newer CPUs have more cores and more memory controllers should improve recovery times.

How large should segments be? (§6.2)

Backups utilize as least 85% of the maximum bandwidth of their storage devices by using 8 MB segments with both flash and traditional hard disks.

How large should partitions be? (§6.3)

Each recovery master can recover 75 to 800 MB/s, depending on the average size of objects in the crashed master's log. To target 1 to 2 second recovery times for expected workloads, the size of each partition must be limited to around 500 to 600 MB and 2 to 3 million objects.

How well does segment scattering compensate for variances in disk performance? (§6.6)

When subjected to vibration, disk performance in our cluster varies by more than a factor of four. RAMCloud's randomized segment scattering effectively compensates for this by biasing placement based on storage performance. As a result, recovery times only increase by 50% when more than half of the disks are severely impaired.

What is the impact of fast recovery on data loss? (§6.7)

Shorter recovery times are nearly as effective at preventing data loss than increased redundancy; our analysis shows that decreasing recovery times by a factor of 100 (which RAMCloud does compared to typical recovery times), improves reliability by about a factor of 10,000.

This chapter also makes projections about the limits of fast recovery based on the experimental observations, and it explores how future hardware trends will affect recovery capacity and times. Some of these projections are summarized below.

How large of a server could be recovered in 1 to 2 seconds with a large cluster? (§6.5.1)

It may be possible to recover 200 GB of data from a failed server in 2 seconds using about 560 machines. Spreading recovery across more servers would result in longer recovery times, since the cost of coordinating hundreds of servers would degrade recovery time.

How will recovery need to adapt to changing hardware over time? (§6.5.1)

In order to keep up with DRAM capacity growth, in the future RAMCloud will need to recover more data per server and scale across more recovery masters. Improving recovery master performance will require multicore and non-uniform memory access optimizations for recovery master replay. To improve scalability, coordinator broadcasts will need to be faster to allow more recovery masters to participate efficiently in recovery. Since network latency is unlikely to improve significantly in the future, the coordinator will have to employ more efficient broadcast patterns.

What would be needed to make RAMCloud cost-effective for small clusters? (§6.5.2)

A cluster of 6 machines can only recover 3 GB of data in 1 to 2 seconds, and smaller clusters recover even less. This makes RAMCloud expensive in small clusters, since only a small fraction of each server's DRAM can be used without threatening availability. For small clusters, it will be more cost-effective to keep redundant copies of objects in DRAM.

What is the fastest possible recovery for a 64 GB server with a large cluster? (§6.5.3)

It may be possible to recover a 64 GB server in about 330 ms using 2,000 machines. With more machines, coordination overhead begins to degrade recovery times.

6.1 Experimental Configuration

To evaluate crash recovery we use an 80-node cluster consisting of standard off-the-shelf components (see Table 6.1) with the exception of its networking equipment, which is based

CPU	Intel Xeon X3470 (4×2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
SSDs	2× 128 GB Crucial M4 (CT128M4SSD2)
NIC	Mellanox ConnectX-2 Infiniband HCA
Switches	4× 36-port Mellanox SX6036 (4X FDR) 1× 36-port Mellanox IS5030 (4X QDR) 1× 36-port Mellanox IS5025 (4X QDR)

Table 6.1: Experimental cluster configuration. All 80 nodes have identical hardware. Each node's network adapter provides 32 Gbps of network bandwidth in each direction, but nodes are limited to about 25 Gbps by PCI Express. The switching fabric is arranged in a two layer fat-tree topology [32] that provides full bisection bandwidth to the cluster. Two SX6036 switches form the core of the fabric with the remaining switches acting as leaf switches.

on Infiniband; with it our end hosts achieve both high bandwidth (25 Gbps) and low latency (5 µs round-trip RPCs).

The default experimental configuration (Figure 6.1) uses each node as both a master and a backup. In most experiments each master recovers a partition of a failed master. One additional machine runs the coordinator, a client application, and an extra master filled with data that is crashed to induce recovery. The CPUs, disks, and network are idle and entirely dedicated to recovery. In practice, recovery would have to compete for resources with application workloads, though we would argue for giving priority to recovery. The backup on each node manages two solid-state flash drives (SSDs). All experiments use a disk replication factor of three: three replicas on disk in addition to one copy in DRAM.

At the start of each experiment, the client fills a master with objects of a single size (1,024 bytes by default). It then sends an RPC to the master that causes it to crash. The cluster detects the failure and reports it to the coordinator, which verifies the failure and starts recovery. The client waits until all partitions have been successfully recovered, then reads a value from one of those partitions and reports the end-to-end recovery time.

All reported recovery times factor out failure detection time and only include the client-observed unavailability due to recovery time. Failure detection time has not been optimized, and the current implementation would add variability to recovery time measurements. The median failure detection time over 100 recoveries spread across 80 recovery masters is 380 ms, with times ranging from 310 ms up to 930 ms.

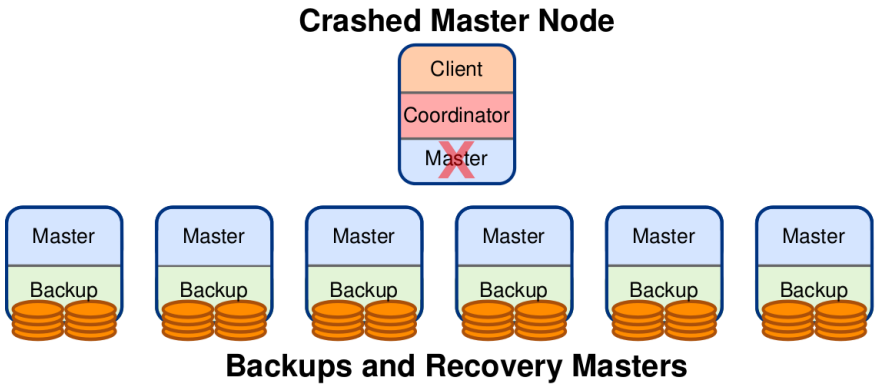


Figure 6.1: In most experiments, each node runs as a recovery master and a backup. Each recovery master is idle and empty at the start of recovery, and each backup manages two solid-state flash drives (SSDs). Some noted experiments replace the SSDs with traditional disks. Each recovery master recovers a partition of a failed master. One additional machine runs the coordinator, a client application, and an extra master filled with data that is crashed to induce recovery (indicated with an 'X' in the figure).

6.2 How Large Should Segments Be?

To achieve efficient storage performance, RAMCloud servers must be configured with an appropriate segment size. Each access to disk has a high fixed cost due to seek and rotational latency, so minimizing the number of disk accesses increases effective disk bandwidth. Flash SSDs help, but disk controller latency still mandates large accesses to achieve maximum disk bandwidth. Consequently, backups buffer objects into full segments and write each segment replica out to disk with a single contiguous operation. This also makes reading replicas from disk efficient, since backups read each replica from disk in a single access from a contiguous region of storage.

However, segments cannot be arbitrarily large. Backups must be able to buffer entire segments in non-volatile DRAM, so using large segments could substantially increase the cost of the system if larger battery backups were needed. Furthermore, gains to storage bandwidth diminish as segment size increases, so segments only need to be so large. Finally, large segments hinder load balancing, since they limit choice in replica placement. A good segment size is one that provides high disk bandwidth utilization and limits the buffering required on backups.

Configuration 1	2× Flash Disks	128 GB Crucial M4 SSDs (CT128M4SSD2) Access Latency: 93 µs read, 222 µs write
Configuration 2	Disk 1	Western Digital WD 2503ABYX (7200 RPM, 250 GB) Access Latency: 9-12 ms read, 4-5 ms write
	Disk 2	Seagate ST3500418AS (7200 RPM, 500 GB) Access Latency: 10-14 ms read, 6-8 ms write

Table 6.2: Storage devices used in experiments. Most experiments use the first configuration, which consists of two identical flash SSDs in each node. The high bandwidth of the SSDs allows high write throughput during normal operation and speeds recovery after failures. The second configuration, used in a few experiments, uses two disks in each node. The first disk is an enterprise class disk and the second is a desktop class disk. The disk-based configuration gives up some write throughput and recovery bandwidth for more cost-effective durable storage. Access latency is determined by averaging the duration of 1,000 random 1 KB accesses. Access latency is given as a range for traditional disks, since it depends on which portion of the disk is accessed. The shorter time is when accesses are limited to the first 25% of the disk; the longer time is when accesses span the entire disk. The bandwidth of each of these devices is characterized in Figure 6.2.

Also, RAMCloud must use a segment size that works well for a variety of storage devices. Some applications may wish to use traditional disks for low-cost durability. Other applications may use solid-state flash drives to provide higher write throughput and faster recovery. Table 6.2 shows two storage configurations used in our experiments. In the first (and default) configuration, each node uses two flash SSDs. In the second configuration, each node uses two traditional hard disk drives, each of the two is a different model. The first model is an enterprise-class disk and the second model is a desktop-class disk. These configurations have significantly different bandwidth and latency characteristics.

A segment size of 8 MB provides high effective disk bandwidth for both experimental configurations. Figure 6.2 shows the effective disk bandwidth of each of these devices as block access size varies. For both traditional disk devices, reading replicas in 8 MB blocks achieves about 85% of each device's maximum bandwidth. For each flash SSD, 8 MB segments achieve 98% of the device's maximum bandwidth. All of the remaining experiments in this chapter use 8 MB segments.

Using an 8 MB segment size also sufficiently limits the buffering needed on backups. During normal operation, each master keeps three replicas of its head segment buffered on

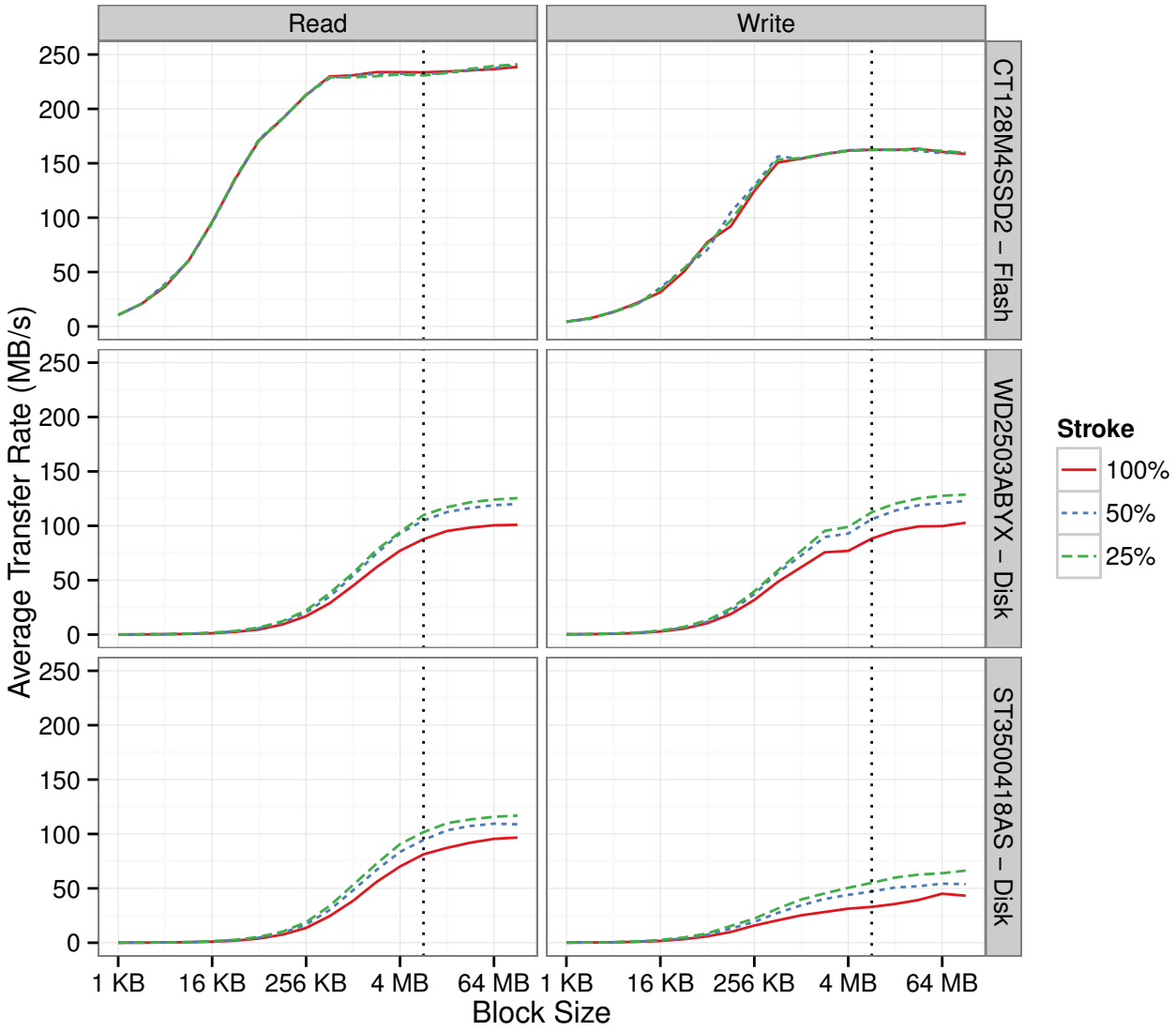


Figure 6.2: Average transfer rate of three different storage devices when randomly reading and writing data in various block sizes. With 8 MB block accesses, flash SSDs operate at about 98% of their maximum transfer rate. Each of the disk models operates at about 85% of its maximum transfer rate. For traditional disk devices, transfer rate is non-uniform across the surface of the disk. Individual lines on each graph show the average transfer rate depending on how much of the device is randomly accessed (starting from the beginning of the disk, where transfer rates are highest). The "50%" line shows that if random block accesses are limited to the first half of the drive, then average transfer rate is improved by about 15 to 20% for the traditional disk-based storage devices. Limiting accesses to the first quarter of each drive provides a further small improvement.

backups. Correspondingly, 8 MB segments will require about 24 MB to 48 MB of non-volatile buffering on each backup. Thus, backup replica buffers typically require a tiny fraction of the total DRAM capacity of each node and can be flushed to storage on power failure in a few hundred milliseconds.

6.3 How Large Should Partitions Be?

Before the coordinator can perform partitioning during recovery, it must know how much data a recovery master can process in 1 to 2 seconds. Figure 6.3 measures how quickly a single recovery master can process backup data, assuming enough backups to keep the recovery master fully occupied. The top graph of the figure shows that, depending on the object size, a recovery master can replay log data at a rate of 75-800 MB/s. This includes the overhead for reading the data from backups and writing new backup copies. With small objects the speed of recovery is limited by the per-object costs of updating the hash table. With large objects the speed of recovery is limited by memory bandwidth while re-replicating segments. For example, with 800 MB partitions and a disk replication factor of 3, each recovery master would need to write 2.4 GB of data to backups, and each replica written to a backup crosses the memory controller multiple times.

For one-second recovery Figure 6.3 suggests that partitions should be limited to no more than 800 MB and no more than 2 million log records (with 128-byte objects a recovery master can process 300 MB of data per second, which is roughly 2 million log records). We limit partition size to 500 MB for the remaining experiments in order to target a 1 to 2 second recovery time. If 10 Gbps Ethernet is used instead of Infiniband, partitions would need to be limited to 300 MB due to the network bandwidth requirements for re-replicating the data.

In our measurements we filled the log with live objects, but the presence of deleted versions will, if anything, make recovery faster. Replicas may be replayed in any order during recovery, and if a newer version of an object is replayed before an older version, then the older version is discarded without adding it to the recovery master's log. So, deleted versions may not need to be re-replicated, depending on the order that segments are replayed.

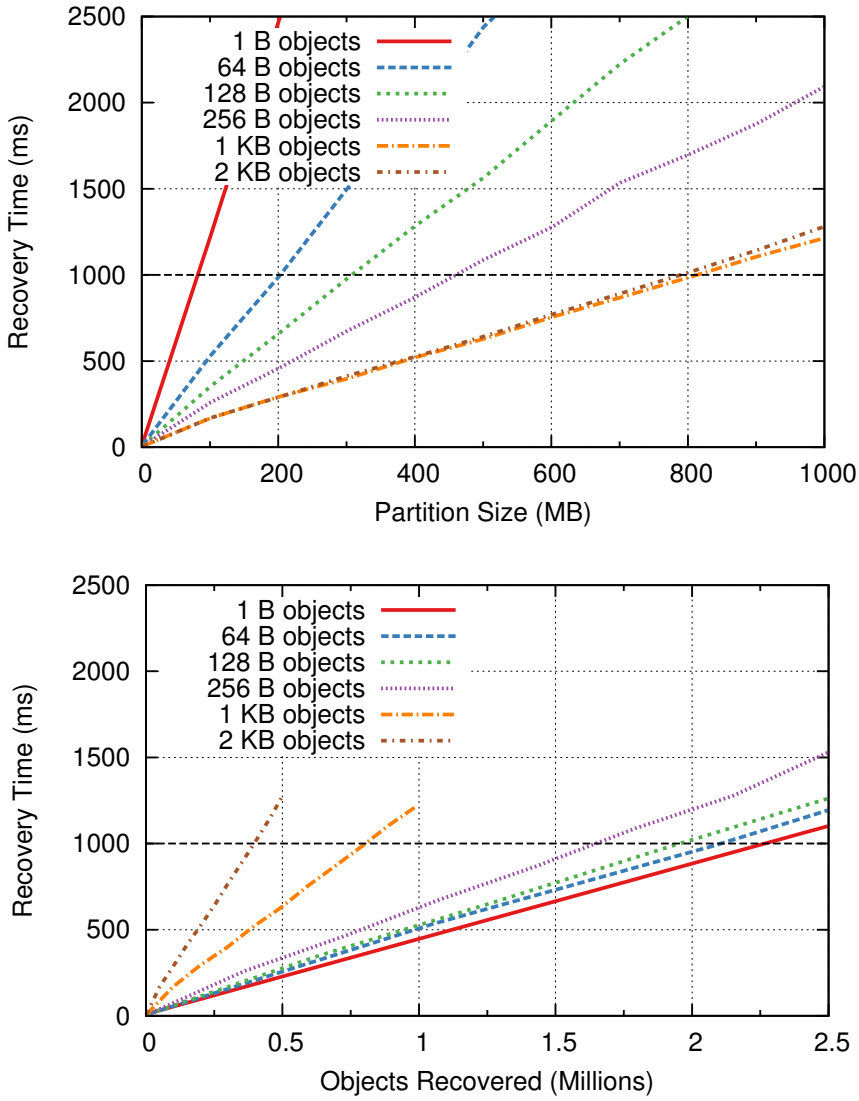


Figure 6.3: Recovery time as a function of partition size with a single recovery master and 80 backups. Using large set of backups with a single recovery master, means that the recovery master is the bottleneck in recovery. The top graph shows recovery time as a function of the amount of data recovered; the bottom graph shows recovery time as a function of the number of objects recovered. Each line uses objects of a single uniform size. Recovery time grows proportionally as partition size increases, but the rate of increase depends on the number of the objects in the partition. Among equal-sized partitions, those containing many small objects take longer to recover than those containing fewer larger objects. Depending on the size of objects in a crashed master's log, a single recovery master can recover between 75 and 800 MB per second. We limit partition size to 500 MB for the remaining experiments in order to target a 1 to 2 second recovery time.

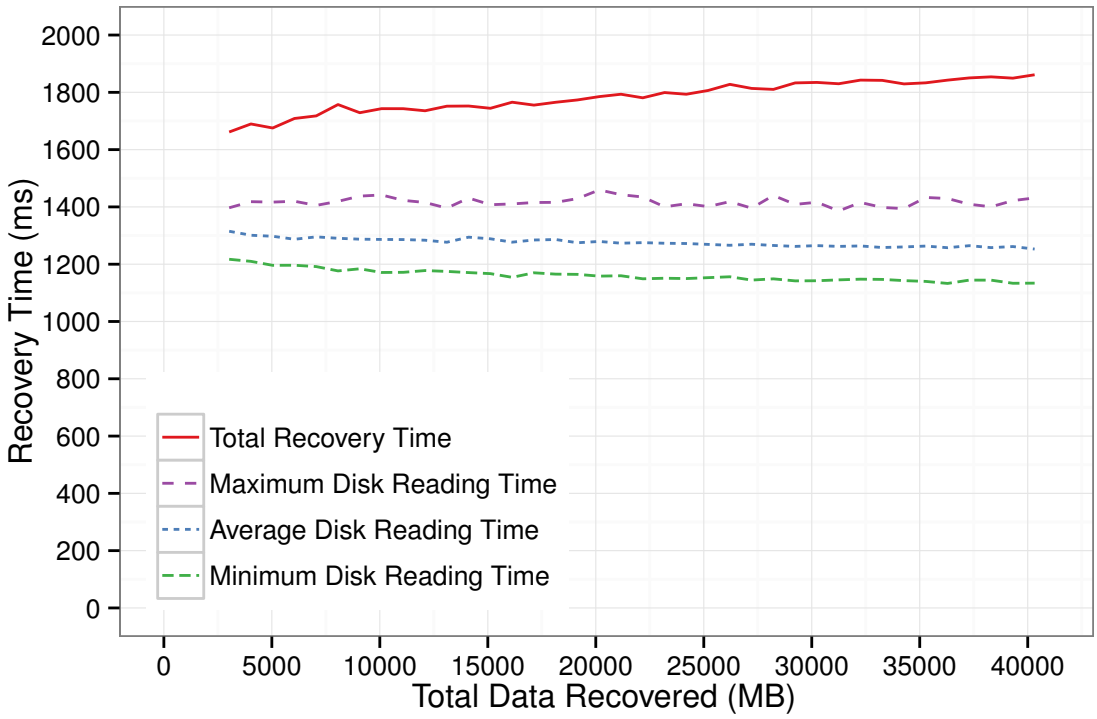


Figure 6.4: Recovery time as the size of the crashed master is scaled proportionally to the number of nodes in the cluster. A master is filled with 500 MB for each node in the cluster then crashed and recovered. Each node also acts as a backup with replicas from the crashed master stored on two flash disks, providing about 460 MB/s of storage read bandwidth per node. The minimum, average, and maximum amount of time backups spend reading data from disk are also shown. Each point is an average of 5 runs (Figure 6.6 shows the variance of master recoveries). A horizontal line would indicate perfect scalability.

6.4 How Well Does Recovery Scale?

The most important issue in recovery for RAMCloud is scalability; specifically, does recovery bandwidth grow as cluster size grows? Ultimately, recovery bandwidth determines how much data can be stored on each master and how quickly a master can be recovered when it fails. If one recovery master can recover 500 MB of data in one second, can 10 recovery masters recover 5 GB in the same time, and can 100 recovery masters recover 50 GB?

Figure 6.4 seeks to answer this question. It plots recovery time as the amount of lost data is increased and the cluster size is increased to match. For each additional 500 MB

Figure 6.5: Same experiment as Figure 6.4, except each recovery master only recovers 300 MB of data. Recovery times for 300 MB partitions are significant, since they are a rough estimate of the maximum partition size that could be used with 10 Gbps Ethernet for fast recovery. With smaller partitions, the 80 recovery masters only recover 24 GB of data, but recovery finishes in 1.0 seconds. Recovery times improve slightly better than in proportion to the reduction in partition size compared to the times in Figure 6.4. Though the amount of recovered data is less overall, the effective recovery bandwidth improves to 24 GB/s compared the 21 GB/s achieved using 500 MB partitions.

partition of lost data, the cluster includes one additional node that acts as both a recovery master and a backup. The additional node should be able to replay the additional 500 MB of data in less than a second, and it should be able to help read an additional 460 MB/s of data from disk. Consequently, recovery times should remain constant as the amount of data recovered scales with cluster size.

Figure 6.4 shows that larger clusters scale to recover larger failures while retaining acceptable recovery times. With 80 recovery masters and 160 flash SSDs, RAMCloud recovers 40 GB of data in 1.86 seconds, which is only 12% longer than it takes to recover

3 GB with 6 recovery masters using 12 SSDs. Recovery times are consistently within the 2 second time budget. Extrapolating, it appears that recovery will scale to recover 64 GB in 1 to 2 seconds with clusters of 120 machines or more. Figure 6.5 repeats the same experiment, but limits partition size to 300 MB. In that configuration, the 80 recovery masters recover 24 GB in 1.0 seconds. The overall amount of data that can be recovered is limited, but the effective recovery bandwidth improves to 24 GB/s compared the 21 GB/s achieved using 500 MB partitions.

To measure the variability in recovery time, Figure 6.6 gives a cumulative distribution of recovery time for three-hundred 40 GB recoveries with 80 recovery masters and 500 MB partitions. The results show that recovery times are consistent across recoveries; in the experiments recovery time varied less than 200 ms across all 300 recoveries.

Table 6.3 and Table 6.4 give a detailed breakdown of recovery time, network utilization, and storage performance for an 80-node 40 GB recovery. The breakdown of recovery master time shows each recovery master spends about half of recovery time replaying objects. Most of the rest of the time is spent waiting on data from backups. Of replay time, more than one-third is spent checking object checksums (about 20% of recovery time on each recovery master). RAMCloud uses the highly optimized Intel 64 `CRC32` instruction to minimize checksum overhead, but it still takes the largest fraction of time for object replay on recovery masters.

6.4.1 Bottlenecks in the Experimental Configuration

It appears the current implementation scales well enough to recover an entire server's DRAM, but back-of-the-envelope calculations imply that it should be possible to recover more quickly. Each recovery master should be able to recover 500 MB in less than one second, and each backup should be able to read its share of data in about 1.1 seconds (500 MB at 460 MB/s). If recovery masters overlapped work perfectly with backups reading data from storage, then recovery times should be about 1.1 seconds, even with an 80 node recovery for 40 GB of data. Figure 6.4 shows that the full 80-node cluster takes 1.86 seconds to recover 40 GB. So, why is recovery 760 ms slower than expected?

Figure 6.4 and 6.6 give some suggestions. First, both of these graphs show that the flash

Recovery Summary	
End-to-end recovery time	1,890.8 ms
Nodes (one recovery master and backup on each)	80
On-storage replicas per segment	3
Object size	1,024 bytes
Objects per partition	498,078
Total objects recovered	39,846,240
	38,912 MB
Total log data recovered	40,328 MB
Metadata overhead (in-memory and on-storage)	3.5 %

Coordinator Time		
Starting recovery on backups (two broadcasts)	9.7 ms	0.5%
Starting recovery on masters	4.0 ms	0.2%
Servicing recovery master completed RPCs	4.0 ms	0.2%
Waiting for recovery masters	1,872.5 ms	99.1%
Total master recovery time on coordinator	1,890.3 ms	100.0%

Recovery Master Time	Average		Min/Distribution/Max
Idling, waiting for objects from backups	696.7 ms	38.9%	495 816
Replaying objects into log and hash table	901.2 ms	50.3%	741 1,080
Verifying object checksums	333.0 ms		255 457
Appending objects to in-memory log	270.3 ms		232 315
Managing replication RPCs	23.5 ms		14 60
Replicating objects after replay completes	26.1 ms	1.5%	11 53
Other time	167.9 ms	9.4%	118 358
Recovery time for assigned partition	1,791.9 ms	100.0%	1,700 1,880

Backup Time	Average	Min/Distribution/Max
Servicing request for replica list	0.2 ms	0.0 0.8
Servicing replication RPCs	802.2 ms	576 1,007
Copying replication data to buffers	796.8 ms	572 1,000
Servicing requests for partitioned objects	6.4 ms	5.6 10.1
Reading replicas from disk	1,268.7 ms	1,115 1,465
Reading replicas from disk and partitioning	1,320.8 ms	1,148 1,504

Table 6.3: Breakdown of recovery time among the steps of recovery. All metrics are taken from one recovery with 80 recovery masters using 500 MB partitions, which recovers 40 GB of data. Metrics taken from backups and recovery masters report average times along with the minimum, maximum, and a distribution. Table 6.4 includes additional statistics taken from the same recovery.

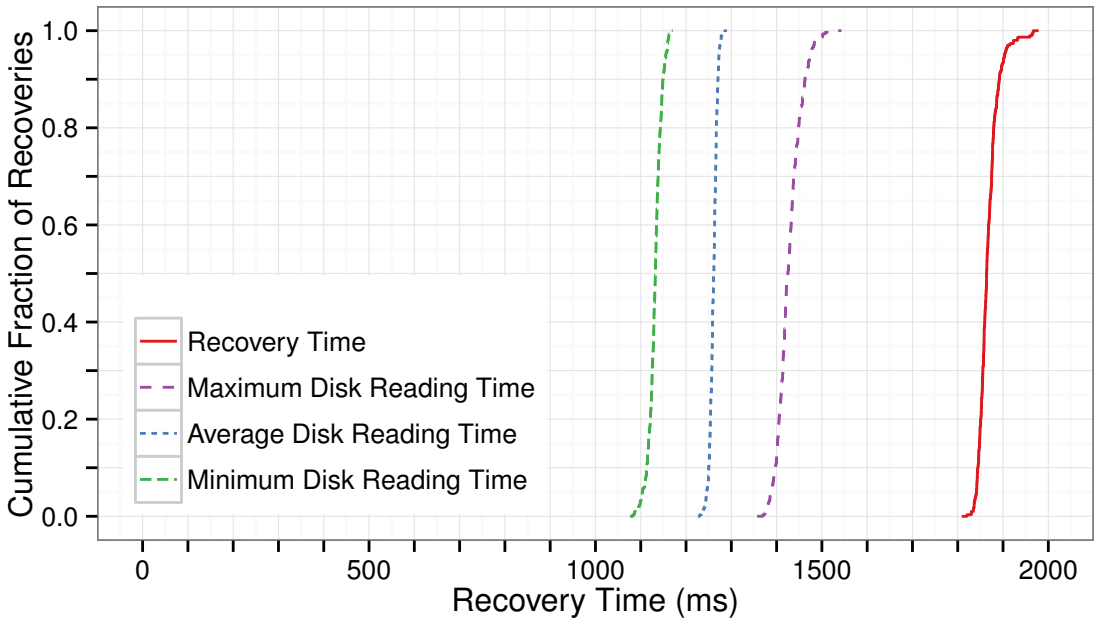
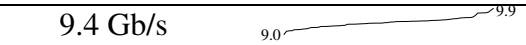


Figure 6.6: Distribution of recovery time for an 80-node cluster recovering 40 GB. Nodes act both as a recovery master and a backup; each node recovers one 500 MB partition and manages two flash SSDs (160 total SSDs cluster-wide). Each line is a cumulative distribution of 300 recoveries showing the fraction of recoveries that completed within a given time. The minimum, average, and maximum amount of time backups spent reading data from disk is also shown. The worst recovery time of all the runs is 1967 ms. The median recovery time is 1864 ms; recovery time ranges less than 200 ms across all of the recoveries.

SSDs perform more slowly during recovery than they do when benchmarked in Figure 6.2. During recovery the slowest SSD takes about 1.4 seconds to read its share of the segments rather than the expected 1.1 seconds. Second, even once all of the data is loaded from storage, recovery takes another 450 ms to complete, which suggests that recovery masters cannot replay data as fast as backups can load and filter it. In fact, recovery masters seem to be replaying log data at about 270 MB/s as compared to the expected 800 MB/s for 1 KB objects reported by the benchmark in Figure 6.3.

What causes both storage I/O performance to decrease and replay performance to decrease? Backups and masters are combined on a single server, so contention for CPU

Network Utilization	Average	Min/Distribution/Max
Average transmit rate per host	9.4 Gb/s	
Aggregate average network utilization	715.5 Gb/s	36% of full bisection

Disk Utilization	Average	Min/Distribution/Max
Active host read bandwidth	398.5 MB/s	
Aggregate	31,878.9 MB/s	
Average host read bandwidth	266.6 MB/s	
Aggregate	21,329.0 MB/s	
Segments read from storage per host	63	

Table 6.4: Network and storage statistics for a recovery. All statistics are taken from a single recovery with 80 recovery masters using 500 MB partitions, which recovers a total of 40 GB of data. Statistics are reported as averages along with the minimum, maximum, and a distribution when appropriate. Table 6.3 includes a breakdown of recovery time for the same recovery. Average transmit rates are total bytes transmitted divided by end-to-end recovery time; peak rates are higher, since transmit rates vary significantly throughout recovery. Similarly, average storage read bandwidth is total data read divided by end-to-end recovery time; active storage read bandwidth reports storage bandwidth over just the intervals of recovery when storage is actively being read.

resources could be a problem. However, that alone would not explain the loss of I/O performance, since it should be mostly independent of CPU utilization. As a result, we focused on the shared resource between these two tasks: memory bandwidth.

To determine whether memory bandwidth utilization is limiting recovery time, two questions must be answered. First, what is the maximum memory bandwidth that each node can sustain? Second, what is the actual memory bandwidth utilization during recovery? To answer the first question, we first ran an experiment that sequentially accesses cache lines and measures memory bandwidth utilization. During the experiment, workload is gradually varied between a pure read-only workload and a pure write-only workload. Figure 6.7 shows the results. The mix of read and write operations has some impact on the amount of data that can be transferred to and from DRAM, but, for the most part, a node's memory bandwidth saturates at about 10 to 11 GB/s.

To address the second question, we measured memory bandwidth utilization during recovery. In order to understand the impact combining a backup and a recovery master on each node, experiments were run in two process configurations. The first configuration

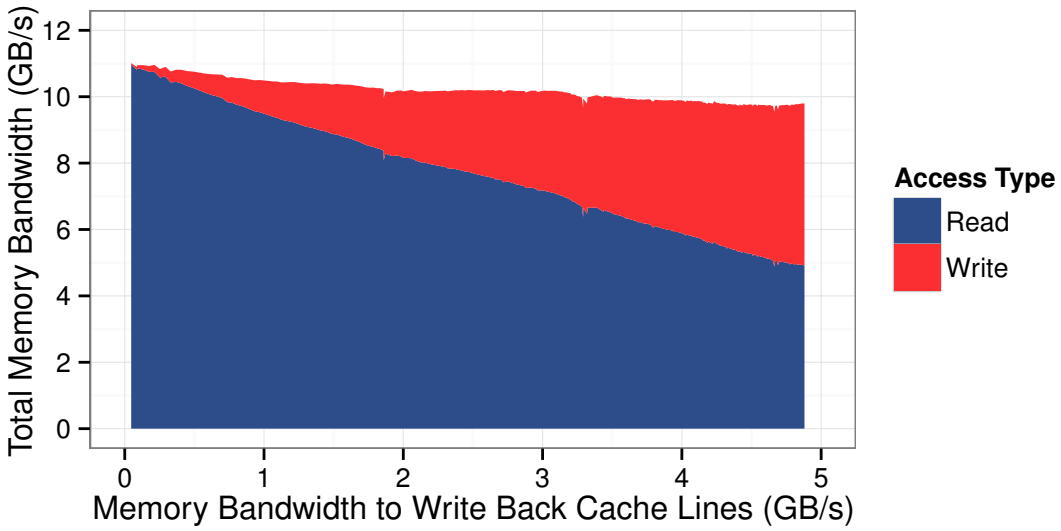


Figure 6.7: Maximum achievable memory bandwidth across a node's DRAM controller as a function of DRAM write traffic. A benchmark sequentially accesses cache lines for a large region of memory to infer the maximum bandwidth to DRAM. The benchmark varies the ratio of read accesses to write accesses to show the total memory bandwidth for varying mixes of traffic. At the left end of the graph, the benchmark only reads cache lines. At the right end of the graph, the benchmark only modifies cache lines; 50% of memory controller traffic is due to cache line reads even when the benchmark is only modifying cache lines, since each cache line must be fetched into cache, modified, and written back. Each node has about 10 GB/s of total memory bandwidth to DRAM.

was the default "combined" configuration, where a master and backup are run together on each node. The second configuration split the role of the master and backup onto separate nodes in order to reduce the memory bandwidth utilization on each node. Furthermore, each configuration was tested both for a smaller recovery including only 6 partitions of data and 6 hosts (12 hosts when backups and masters are split), and a larger recovery with 40 partitions of data and 40 hosts (80 hosts when backups and masters are split). Recoveries using more than 40 partitions were not possible in the split configuration; all 80 nodes were in use, each running either a recovery master or a backup. These recovery sizes correspond to the size of the recoveries of the leftmost and middle data points in Figure 6.4.

Figure 6.8 plots the results, and shows that memory bandwidth is saturated during recovery, mostly due to the load on backups. A recovery master running alone on a node

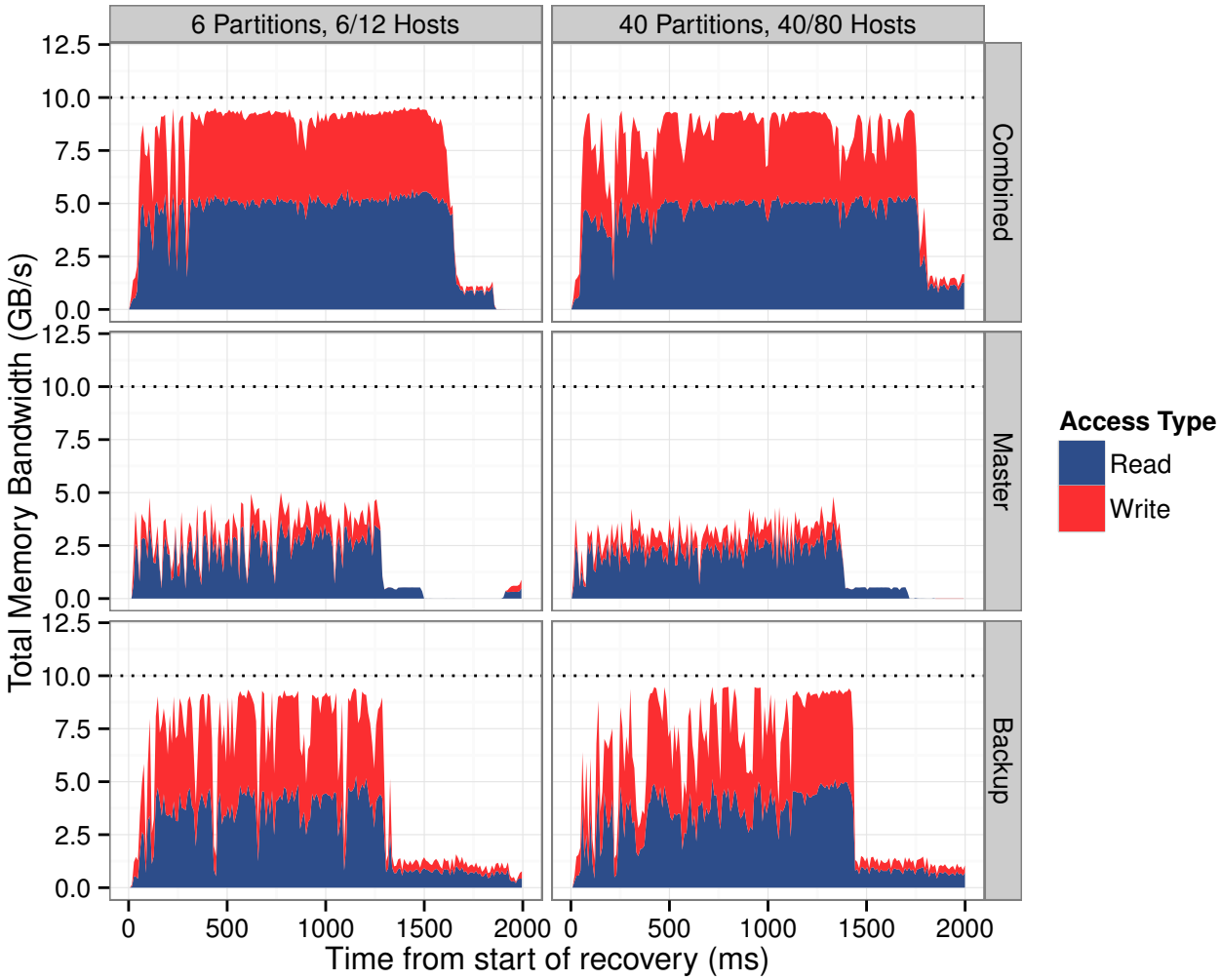


Figure 6.8: Memory bandwidth utilization during recovery. "Combined" shows memory bandwidth utilization during recovery when each node acts as both a recovery master and a backup as in Figure 6.4. The "Master" and "Backup" graphs show memory bandwidth utilization throughout recovery when recovery masters and backups are run on separate nodes as in Figure 6.9. Recoveries with masters and backups on separate nodes use twice as many nodes as the corresponding recoveries with masters and backups combined on the same node. Recoveries labeled "6 Partitions, 6/12 Hosts" correspond to the leftmost data point in Figure 6.4 and recover 3 GB of data total. Recoveries labeled "40 Partitions, 40/80 Hosts" correspond to the middle point of Figure 6.4 and recover 20 GB of data total. Nodes running a recovery master alone do not saturate memory bandwidth. Nodes running a backup alone saturate memory bandwidth during most of recovery. When a recovery master and a backup run on the same node memory bandwidth is saturated for the duration of recovery, and recovery time increases.

uses about 2 to 4 GB/s of memory bandwidth, and a backup alone uses about 9 GB/s of memory bandwidth. Since memory bandwidth saturates at about 10 GB/s, it appears that a backup alone may already be memory bandwidth constrained. When recovery masters and backups are placed on a single server, it becomes overloaded; memory bandwidth remains saturated at about 9 GB/s for the entire recovery, and recovery slows down. Overall, the results show that the backup component on each node is much more memory bandwidth intensive than the recovery master component.

Figure 6.9 shows how using the alternate configuration where masters and backups are run on separate nodes improves overall recovery times. Compared to Figure 6.4, recovery times improve by about 20% for both large and small recoveries. Mitigating memory bandwidth overload also improves the scalability of recovery; the slope in recovery time is about 35% lower overall when backups and recovery masters run on separate servers.

The reason memory bandwidth is saturated is an artifact of the size of our experimental cluster; the work of recovery is concentrated on a small number of nodes, which become overloaded. In an expected RAMCloud deployment of 1,000 machines or more, recovery would be spread across more backups. To recover 64 GB from a crashed master, each backup would only load and process 8 segments and would only expect to receive about 24 new segment replicas. In our 80-node cluster, each backup must process 64 replicas and backup 192 new replicas. As a result, each backup does eight times the work in nearly the same period than it would in a realistic deployment. Furthermore, each node in this configuration also acts as a recovery master, whereas in a full scale cluster, less than 10% of nodes would have to perform both roles at the same time. Thus, recovery should be both faster and more scalable in a larger production cluster than in our experimental cluster.

Newer CPUs provide substantially higher memory bandwidth and additional cores compared to the nodes of our experimental cluster. On newer machines, placing recovery masters together with backups in a small cluster should have a less drastic effect on recovery time. For example, benchmarking a machine that is two years newer showed that its memory bandwidth is about 47 GB/s: more than four times higher than the machines used in these experiments. On the other hand, newer machines also have higher DRAM capacity, so newer hardware may still require a large cluster to recover an entire server's DRAM quickly.

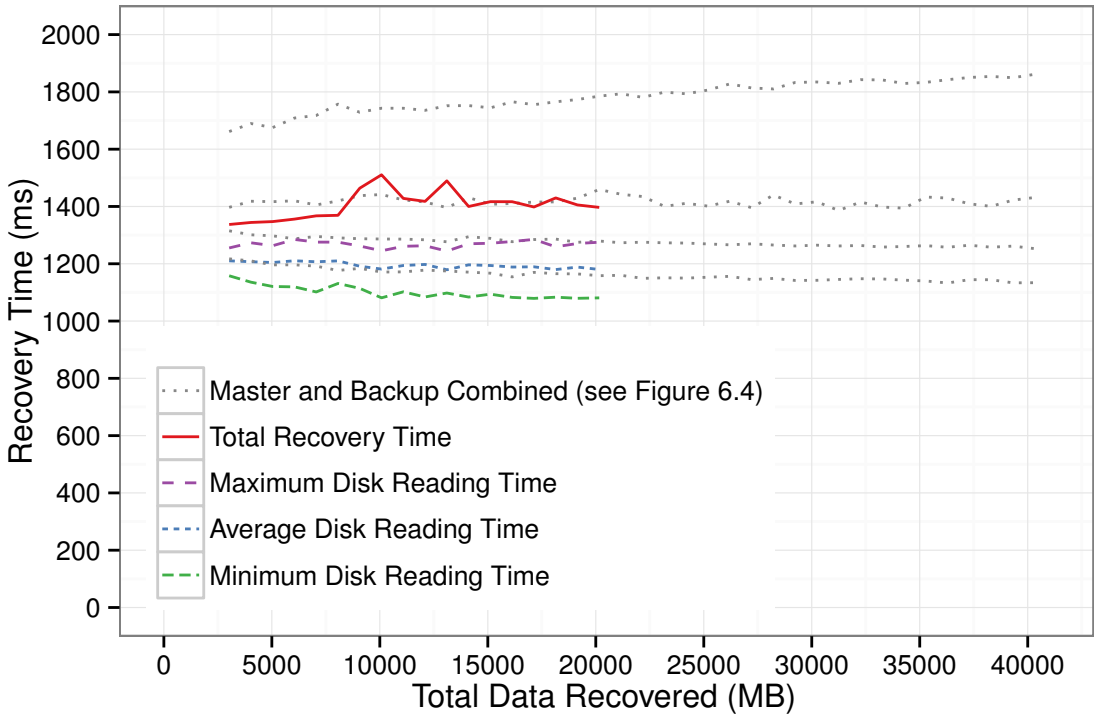


Figure 6.9: Recovery time as crashed master size and cluster size are scaled when recovery masters and backups are run on separate nodes. Just as in Figure 6.4, a master is filled with 500 MB for each recovery master in the cluster. The master is then crashed and recovered. In these recoveries, two nodes are added for each partition recovered: one operating as a recovery master and one operating as a backup. Lines from Figure 6.4 are shown as dotted gray lines for comparison. Separating the work of backups and masters improves overall recovery time. The flash SSDs perform more consistently and finish reading replicas more quickly. Also, the time between when backups finish loading replicas into DRAM and when recovery masters finish replaying data is shorter. Each point is an average of 5 runs.

6.4.2 Recovery Coordination Overhead

It is important to keep the overhead for additional masters and backups small, so that recovery can span hundreds of hosts in large clusters. For example, the coordinator performs three rounds of broadcast during recovery; if the time for these broadcasts grows to a substantial fraction of recovery time in a large cluster, then large recoveries may become impractical. As another example, recovery masters must perform more RPCs per byte of data recovered as the number of partitions used during recovery increases. A full failed master always leaves behind the same number of replicas, and each recovery master always recovers the same amount of data regardless of the number of partitions. However, as more partitions/recovery masters participate in recovery, each replica is broken down into more buckets, each of which is smaller. Recovery masters fetch data from backups one bucket at a time, so they must make more requests to collect the same amount of data as the number of partitions increases. If it becomes impossible to keep recovery masters busy as a result, then recovery time will suffer.

In order to isolate the overhead of the recovery coordination broadcasts, we measured recovery time with artificially small segments that each held a single 1 KB object and kept all segment replicas in DRAM to eliminate disk overheads. The bottom series of points in Figure 6.10 shows the recovery time using trivial partitions containing just a single 1 KB object in a single segment; this measures the cost for the coordinator to contact all the backups and masters during the recovery setup phase, since the time to replay the objects on recovery masters is negligible. Our cluster scales to 80 recovery masters and backups with only about a 10 ms increase in total recovery time thanks to our low-latency RPC system.

The upper series of points in Figure 6.10 exposes the overhead for recovery masters to perform the increased communication with backups required as the number of partitions scales. It plots recovery time for a crashed master with 63 segments each containing one object. In this configuration, the cluster performs roughly the same number of RPCs (shown as the dashed line) to fetch data from backups and re-replicate data to backups as it does when recovering 500 MB partitions using 8 MB segments, but very little data is processed. As a result, the results expose RPC overheads as recovery scales. As the

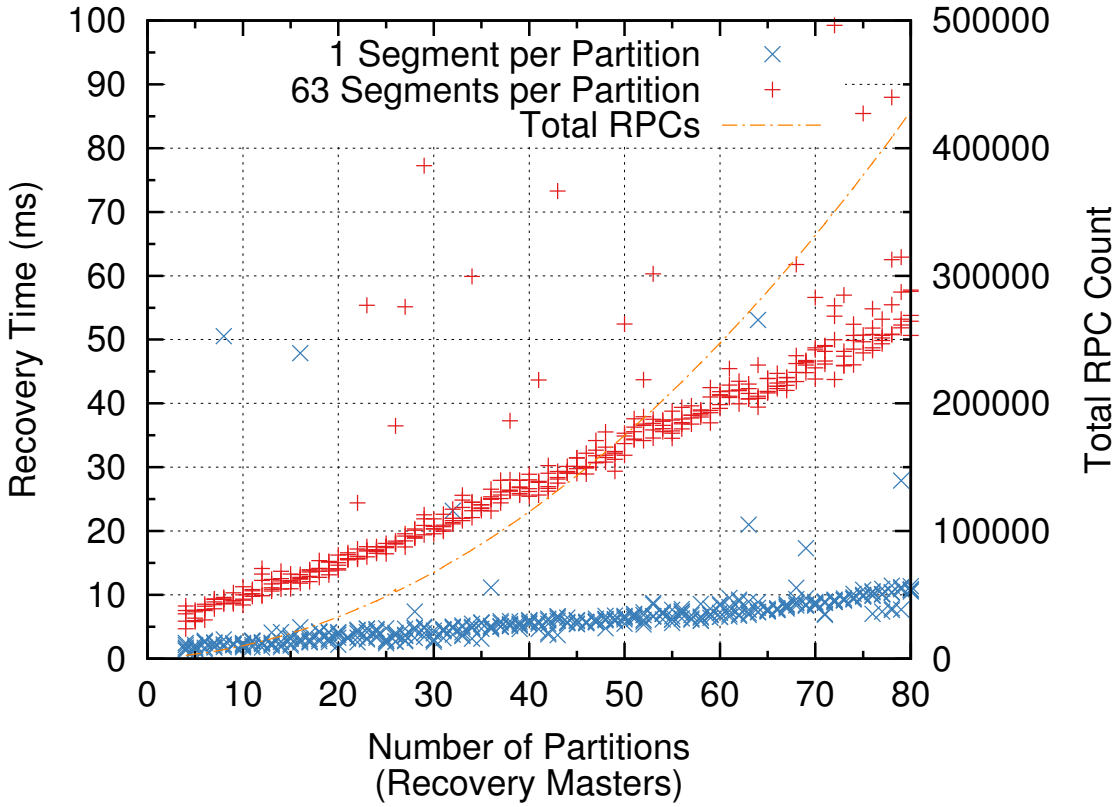


Figure 6.10: Recovery management overhead as a function of system scale. Segment size is limited to single 1 KB object and segment replicas are stored in DRAM in order to eliminate overheads related to data size or disk. Recoveries with "1 Segment per Partition" recover a single object per recovery master. This measures the coordinator's overheads for contacting masters and backups. With 80 nodes, the fastest possible recovery time is about 10 ms. Recoveries with "63 Segments per Partition" recover the same number of segments as the full 500 MB partitions used in Figure 6.4, but each segment only contains a single 1 KB object. In this configuration, each node performs roughly the same number of RPCs (shown as the dashed line) as it would in a full recovery but without the transfer and processing overheads for full 8 MB segments. It measures the overhead of the increased communication required between masters and backups to split recovery across more recovery masters. Each node acted as both a recovery master and a backup. Each data point represents a single recovery; recoveries are repeated five times each for all numbers of partitions from 6 to 80 to show variability.

system scale increases, each master must contact more backups, retrieving less data from each individual backup. Each additional recovery master adds about 600 µs of overhead, so work can be split across the 120 recovery masters needed to recover a full 64 GB server while accounting for less than 7% of total recovery time. At the same time, this overhead represents about 12% (120 ms) of a one-second recovery time using 200 recovery masters, so it may constitute a noticeable delay for larger recoveries.

6.5 The Limits of Fast Crash Recovery

The experiments of this chapter raise questions about the limits of fast crash recovery as cluster size is scaled up or down. Experiments indicate that recovery times will begin to suffer noticeable overheads when scaling beyond a few hundred recovery masters. What is the largest server that can be recovered in 1 to 2 seconds today? Will recovery be able to retain 1 to 2 second recovery times as server hardware evolves, particularly as DRAM capacity grows? What changes will be needed for fast recovery to be effective for future clusters? At the same time, what is the smallest cluster for which fast crash recovery is effective? Our 80 node cluster is already heavily overloaded in providing 2 second recovery times for 40 GB of data. Will long recovery times in small clusters make RAMCloud impractical for small datasets? Finally, what is the fastest possible recovery for a full 64 GB master in a large cluster, and what would be the bottleneck in sub-second recoveries?

6.5.1 Scaling Up

The key question for the long-term viability of fast recovery is, "how much data can a sufficiently large RAMCloud cluster be expected to recover in 1 to 2 seconds?" To answer this question, we use the measurements from earlier experiments to project the amount of data a cluster could recover if recovery time were fixed to either 1 or 2 seconds. Figure 6.9 showed that when servers are not overloaded they recover data at about 360 MB/s. Analyzing the slope of that same figure, recovery time increases by about 1.8 ms for each server added to recovery. Using these numbers we can project how much data can be recovered as cluster size grows. The top graph of Figure 6.11 shows the results. The second line from

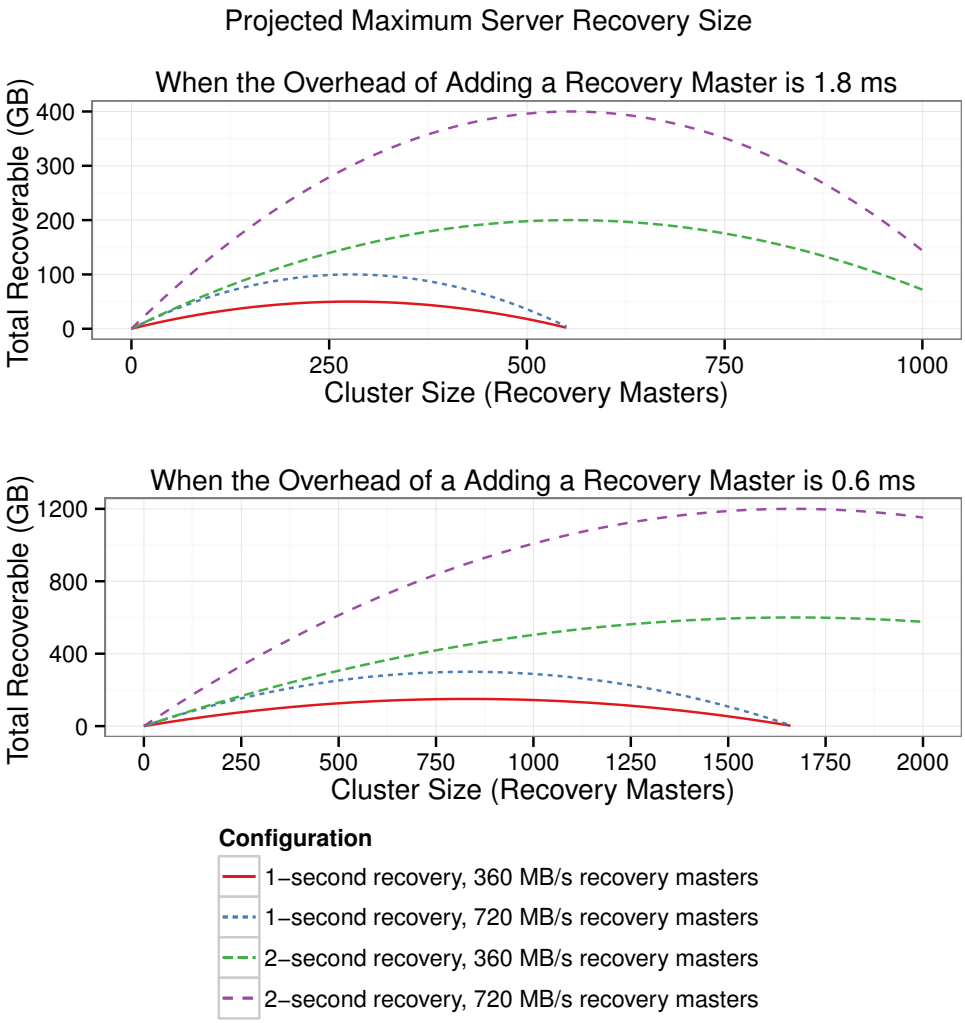


Figure 6.11: Projection of the maximum size of crashed server recoverable in 1 or 2 seconds. Each line shows how much data can be recovered depending on how quickly each recovery master replays data and whether recovery time is fixed at 1 or 2 seconds. The top graph assumes each additional recovery master adds 1.8 ms of wasted time due to coordination overhead (derived from the slope of Figure 6.9). The bottom graph assumes each additional recovery master adds 600 µs of wasted time to recovery (derived from the more optimistic slope of Figure 6.10). Note, the scales do not match between the two graphs. Today's implementation should scale to recover a failed master with 200 GB of data in 2 seconds with about 560 recovery masters (top graph, second line from the top). If the overhead of adding a recovery master drops to 600 µs, recovery scales to 3 times as many machines and should recover 600 GB servers in 2 seconds (bottom graph, second line from the top). Lines labeled "720 MB/s recovery masters" project how much could be recovered if recovery master performance is doubled (for example, by spreading replay across multiple cores). With the higher recovery master performance, servers as large as 1.2 TB could be recovered in 2 seconds.

the top shows that RAMCloud should be able to recover 200 GB of data using 560 recovery masters when recovery time is limited to 2 seconds. When recovery time is limited to 1 second (the solid line), server capacity must be limited to 50 GB, even in large clusters. These estimates are likely to be optimistic. There will be unexpected super-linear costs hidden in the implementation that will be exposed as RAMCloud is run on large clusters; however, these bottlenecks should not be fundamental to the design.

Recovery bandwidth can be improved in two ways to recover larger servers. The first way is by making each server faster. If backups can load data more quickly and recovery masters can replay data more quickly, then more data can be recovered in 1 to 2 seconds. The second way is to reduce the overhead of including more recovery masters in recovery, which allows recovery to be split efficiently across more servers. Figure 6.11 suggests that both optimizations will be needed in order to scale to recover the growing DRAM of machines; each recovery master will need to perform more work, and RAMCloud will need to eliminate overheads to incorporate more recovery masters.

Improving recovery master performance will require additional server-localized optimizations to take advantage of multicore and non-uniform memory access (NUMA). Recently, CPU performance has seen little single-core performance gain. Instead, newer CPUs provide more cores and more sockets. Server memory controllers have seen a similar trend; NUMA is now ubiquitous in server architectures. To achieve the best performance, developers must partition data across DRAM banks attached to different cores and make an effort to co-locate computation on the core nearest to the desired data. To recover more data more quickly, recovery masters will have to be adapted to spread the work of recovery across all of the cores and all of the memory controllers of each machine.

Some effort will be required to determine the most efficient way to divide the work of recovery across many cores and memory controllers on each recovery master. Recovery masters could break up the work of recovery in a fine-grained way by replaying different data from segment replicas on different cores in parallel. However, cores would need to share access to a single hash table, in-memory log, and replica manager. All of these structures are thread-safe and provide fine-grained locking where possible, but parallel recovery threads would contend for access to these structures. The in-memory log may be a particularly serious bottleneck since all appends to the head of the log are serialized.

A more coarse-grained approach would be to run logically separate masters on each server. This would limit contention on shared data structures, but it may not provide an overall improvement if overhead of including each of these logical recovery masters in recovery is just as high as it is for a standard recovery master today.

Servers with 8 cores are ubiquitous today and 16 cores are common, so obtaining a two-fold recovery master performance over the 4-core machines used in evaluation seems attainable. Multicore and NUMA optimizations would have provided little benefit in our experimental cluster; each server only provided a single memory controller and all cores were fully utilized even with a single core replaying recovery data. Additional lines in Figure 6.11 show recovery capacity when recovery master performance is doubled to 720 MB/s; recovery capacity is doubled, so failed servers with 400 GB of DRAM could be recovered in 2 seconds. If recovery master performance were improved to more than 1 GB/s, then network bandwidth would start to become a limitation, since each recovery master must triplicate all the data it processes; servers would need to move to 56 Gbps Infiniband or 100 Gbps Ethernet if recovery master performance quadrupled.

Reducing the overhead for including additional recovery masters in recovery is just as essential to improving recovery capacity as improving recovery master performance. Figure 6.10 measured "data-less" recoveries that perform the same number of RPCs as a full recovery. The results showed that recovery time increases by about 600 µs for each additional recovery master. In a large-scale recovery, each RPC contains very little data, similar to these data-less recoveries. Optimistically, if full recoveries could be tuned to scale with equally low overheads, then the amount of data that can be recovered in 1 to 2 seconds would improve by a factor of 3. The bottom graph of Figure 6.11 shows the same recovery configurations as the upper graph except with a 600 µs overhead to add a recovery master instead of 1.8 ms. With this change, 600 GB masters could be recovered in 2 seconds. Taken together with multicore optimizations, recovering a failed 1.2 TB master should be possible in 2 seconds with about 1600 machines.

6.5.2 Scaling Down

Not only do our experiments raise questions about scaling up recovery, but they also raise questions about how well recovery will work in small RAMCloud clusters. Figure 6.5 shows that RAMCloud recovers the full capacity of one of our (24 GB) nodes in 1.0 seconds using our entire 80-node cluster, but less and less data can be recovered in the same time period as cluster size scales down. With fewer nodes there are fewer recovery masters, which limits the amount of data that can be recovered in a few seconds. For example, the same figure shows that a cluster of 6 nodes only recovers 1.8 GB of data in about 900 ms. Using 500 MB partitions (Figure 6.4), 6 nodes can recover 3 GB of data in about 1.65 seconds. Thus, a 6-node cluster can only support about 3 GB of data per node while retaining a 2-second recovery time.

Severely limiting per-server capacity hurts RAMCloud's cost-effectiveness at small scale. This is problem is compounded, since servers with 64 GB to 256 GB of DRAM capacity are most cost-effective today. For small clusters, a different or supplementary approach may be more cost-effective. For example, a small cluster could duplicate each object in the DRAM of two servers in addition to logging data on disk as usual. Since failures are infrequent in a small cluster, two-way replication is likely to be sufficient. Online redundancy would eliminate lapses in availability and provide additional time to reconstruct the lost object replicas. Returning to the 6-node cluster, DRAM utilization could be safely improved to 50%, a ten-fold increase in per-server capacity over using fast recovery alone. Buffered logging would still be necessary to protect against data loss during correlated failures and power loss.

Adding servers to a small cluster causes a quadratic improvement on the overall capacity of a RAMCloud cluster. This is because each added server brings along two benefits. First, it adds more DRAM capacity. Second, it can act as a recovery master, which allows more DRAM capacity of each of the existing masters to be used safely. Thus, adding servers is particularly cost-effective way to increase cluster capacity up until the amount of data that can be recovered in 1 to 2 seconds exceeds the DRAM capacity of an individual server. Thus, our 2 TB 80-node cluster with 24 GB servers and a 1.0 second recovery takes full advantage of this quadratic effect; we can use all of the DRAM of the cluster safely.

6.5.3 What Is the Fastest Possible Recovery?

Assuming a large cluster, it should be possible to recover an entire server in less than 1 second by using more backups, more recovery masters, and smaller partitions. Given a cluster unbounded in size, what is the fastest a full server with 64 GB of DRAM can be recovered? The key constraint is how quickly the coordinator can contact servers; the faster the coordinator can broadcast, the wider it can spread the work of recovery, which boosts recovery bandwidth.

Using an estimation of coordinator broadcast times, recovery master replay bandwidth, and flash I/O performance yields a best-case lower-bound of 330 ms to recover a 64 GB server using a 2,000 node cluster with today's implementation. The bottom set of points in Figure 6.10 measure the cost of coordinator broadcast during recovery. It shows that an 80-node recovery can perform the three broadcasts required for recovery (along with small amount of other work) in about 10 ms. Assuming each of the three 80-node broadcasts takes about one-third of that (3 ms), each coordinator broadcast should scale to take about 80 ms in a 2,000 node cluster. Each flash disk can read a segment replica in about 30 ms, so if a 64 GB server is divided into 8,000 segments scattered across 2,000 servers with 2 flash disks each, then each server should be able to load its primary replicas in about 60 ms. Factoring in the time for the initial broadcast, all the required data for recovery should be completely loaded from storage within 140 ms after the start of recovery. The coordinator would start its third broadcast to start recovery masters 160 ms into recovery (after the first two rounds of broadcast). Assuming each recovery master can replay data at 360 MB/s and accounting for broadcast time, 2,000 recovery masters could replay the data in about 170 ms. As a result, 330 ms would be a near-optimal recovery time for a failed 64 GB server, and a 2,000 machine cluster would be required. Using a larger cluster is slower due to longer broadcast times, and using less machines is slower due to reduced I/O and replay bandwidth.

If coordinator broadcast times were made nearly instantaneous, then recovery times as low as 50 ms should be possible. Each server would read a single replica from storage in about 30 ms, all replicas would become available in DRAM at the same time, and the

recovery masters would replay them in about 20 ms. Though broadcasts cannot be instantaneous, recovery could use a tree-like pattern to spread the work quickly. Recovery would then be able to spread work across a number of nodes exponential to broadcast time (instead of linear, as it is now). In the time of about 12 back-to-back RPCs by the coordinator, recovery could be spread across 4,000 nodes; in this case, each SSD would only need to load a single segment, which would provide very near minimal recovery times.

6.6 How Well Does Segment Scattering Work?

During normal operation, each master works to optimize recovery time by balancing the amount time each disk will have to spend reading data during recovery. Balancing disk reading time eliminates stragglers and improves recovery time, since the cluster never has to wait idly on a few slow disks to finish loading data. Is this optimization necessary, and is it effective? In particular, how does it impact recovery time when disk performance is relatively uniform and when disk performance is highly variable? Can it mask degraded performance due to defective disks, and will it be able to hide performance differences between disk models?

To answer these questions, we ran an experiment that varied the algorithm used to distribute segment replicas and measured the impact on recovery time. In these experiments, the two flash SSDs in each node were replaced with two traditional hard disks, one each of two different models (Configuration 2 in Table 6.2). Our disks have greater performance variance than our SSDs, and RAMCloud must effectively compensate for this additional variance in order to recover quickly. We measured three different variations of the placement algorithm: the full algorithm, which considered both disk speed and number of primary replicas already present on each backup; a version that used purely random placement; and an in-between version that attempted to even out the number of primary replicas on each backup but did not consider disk speed. The top graph in Figure 6.12 shows that the full algorithm improved recovery time by about 33% over a purely random placement mechanism. Much of the improvement came from evening out the number of segments on each backup; considering disk speed improved recovery time by only 12% over the even-segment approach because the disks did not vary much in speed.

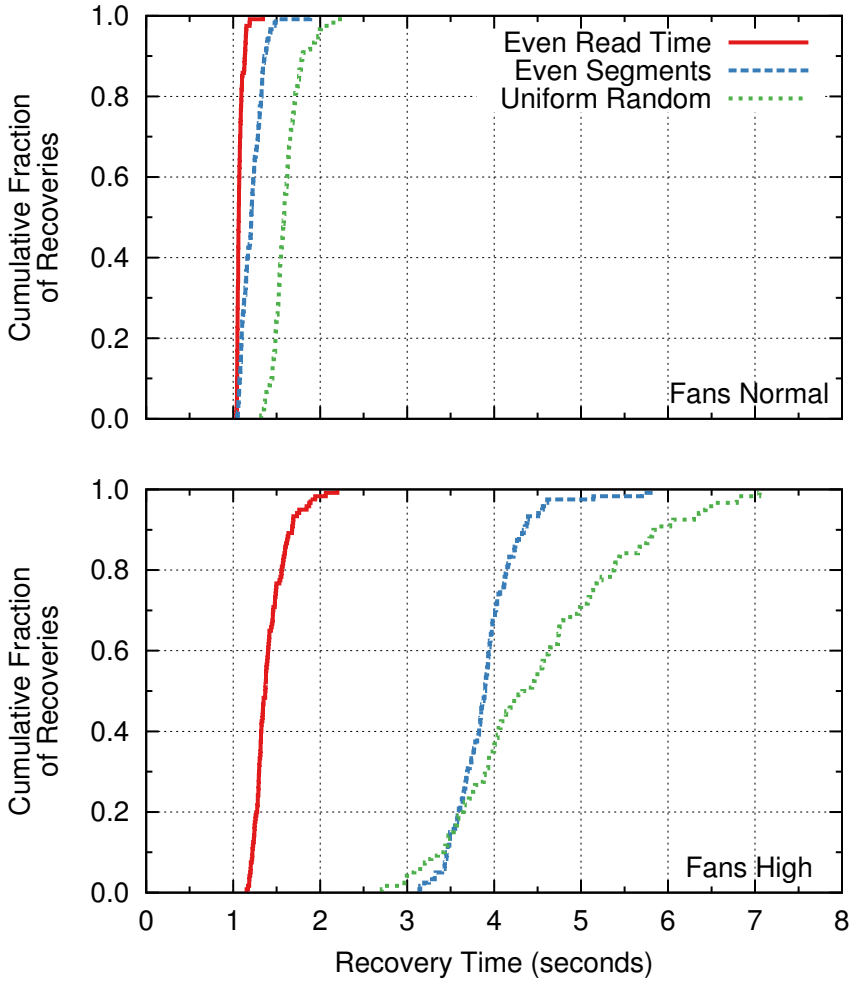


Figure 6.12: Impact of segment replica placement and variance in storage performance on recovery time. In this experiment, the two flash SSDs in each node are replaced with two traditional hard disks, one each of two different models (Configuration 2 in Table 6.2). In each recovery, 20 recovery masters each recover 600 MB of a 12,000 MB crashed master from 60 backups (120 total disks). Each line is a cumulative distribution of 120 recoveries showing the fraction of recoveries that completed within a given time. The disks provide about 12,000 MB of combined read bandwidth, so ideally recovery should take about 1 second. "Even Read Time" uses the placement algorithm described in Section 4.3; "Uniform Random" uses a purely random approach; and "Even Segments" attempts to spread segment replicas evenly across backups without considering disk speed. 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 that degraded performance significantly for some disks). With fans at normal speed, "Even Read Time" and "Even Segments" perform nearly the same since there is little variation in disk speed.

To test how the algorithm compensates for degraded disk performance we also took measurements using the configuration of our cluster when it first arrived. The fans were shipped in a "max speed" debugging setting, and the resulting vibration caused large variations in speed among the disks (as much as a factor of 4×). The performance degradation disproportionately affected one model of disk, so about half of the disks in the cluster operated in a severely degraded state. In this environment, the full algorithm provided an even larger benefit over purely random placement, but there was relatively little benefit from considering replica counts without also considering disk speed (Figure 6.12, bottom graph). RAMCloud's placement algorithm compensates effectively for variations in the speed of disks, allowing recovery times almost as fast with highly variable disks as with uniform disks. Disk speed variations may not be significant in our current cluster, but we think they will be important in large data centers where there are likely to be different generations of hardware.

6.7 Impact of Fast Crash Recovery on Data Loss

RAMCloud replicates each segment across backups of the cluster to protect against data loss. By default, each master stores three copies of data on backups in addition to the copy of data stored in its DRAM. As a result, RAMCloud will never lose data until at least four nodes are inaccessible. However, the threat of data loss rises when too many machines are simultaneously unavailable.

This suggests two ways to improve durability: increased redundancy and fast recovery. Increasing redundancy increases the number of simultaneous failures a cluster can sustain without threat of data loss. Fast recovery reduces the length of each individual failure, which reduces the probability of overlapping failures. Fast recovery has an advantage in that it does not require additional storage capacity; however, additional redundancy is also beneficial, because it improves resilience against bursts of failures. Given these two approaches how should a system trade-off between them? Should a system just keep a two copies of data on disk and recreate lost replicas in 1 second? Is it better to keep three replicas of every segment and recreate lost replicas in 100 seconds?

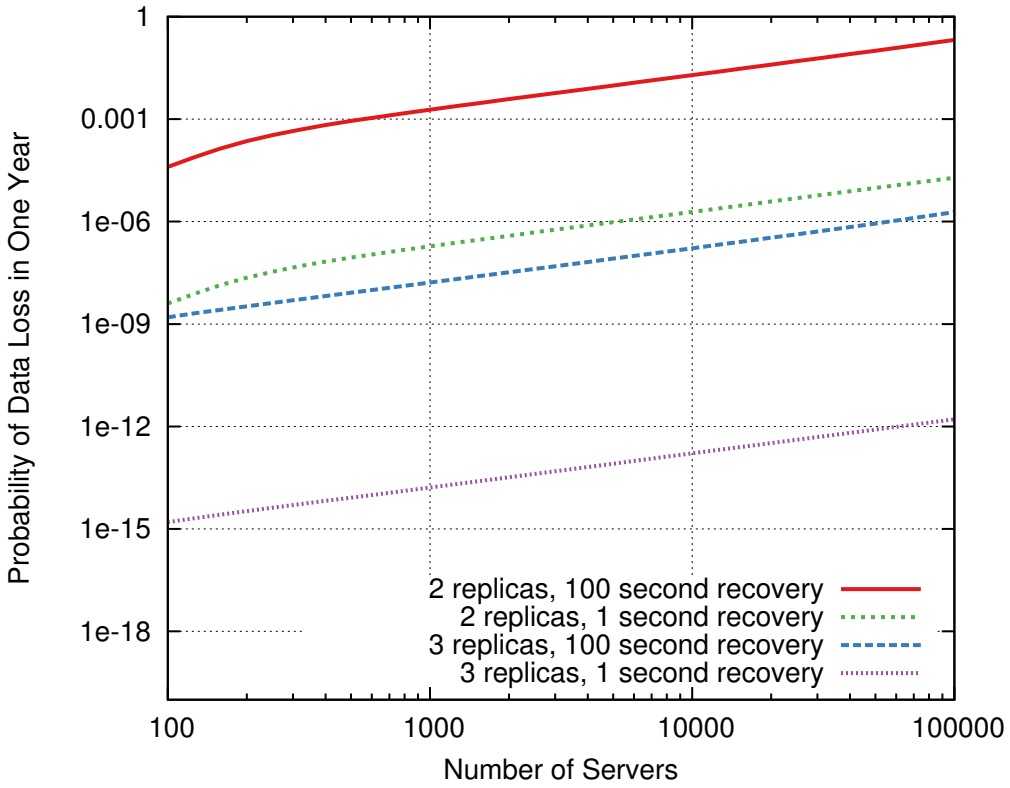


Figure 6.13: Probability of data loss in one year as a function of cluster size, assuming 8,000 segments per master and two crashes per year per server with a Poisson arrival distribution. Different lines represent different recovery times and levels of replication. "2 replicas" indicates two disk-based replicas with one additional replica in DRAM; "3 replicas" keeps a third replica on disk. When keeping two on-disk replicas, reducing recovery time by a factor of 100 reduces the probability of data loss by a factor of 10,000. If recovery time is fixed at 100 seconds and three copies are stored on disk rather than two, then the probability of data loss improves by a factor of 25,000. The gap between 1 second recovery with 2 and 3 replicas shows that fast recovery reduces the probability of data loss even more effectively as the replication level goes up. Many real systems have recovery times on the order of 100 seconds (like GFS [18]), and improving their recovery times would increase their fault-tolerance in larger clusters.

To answer these questions, we simulated the probability of data loss with varying recovery times and replication configurations. For these simulations, we assume that a single server will fail about twice per year and that server failures are independent. Data provided by Ford et. al. [15] suggest that servers will fail with about this frequency. For a 3-month period they recorded between about four to ten failures per 1,000 servers per day on most days. This translates to about two to four failures per server per year. Simulations were run with 1-second recovery and with 100 second recovery, which is the recovery time of some other data center storage systems (for example, GFS [18]). Simulations are repeated with two and three on-disk replicas. Note that two on-disk replicas corresponds to the typical triple replication level of many disk-based systems (like GFS), since RAMCloud stores an additional replica in DRAM.

Figure 6.13 plots the probability of data loss as cluster size grows for each of these configurations. The top line shows that with long recovery times two replicas on disk is insufficient; a large cluster has a 20% chance of losing data each year. Adding an additional replica reduces the probability of data loss by about a factor of 25,000 (the third line from the top). If two replicas are retained and recovery time is decreased to 1 second instead, then the probability of data loss each year drops by a factor of about 10,000. Overall, the key observation is that improving recovery time by a factor of 100 (as RAMCloud does over typical systems) is nearly as effective at preventing data loss as adding an additional replica. Corroborating this further, adding a third replica on disk decreases the risk of data loss more significantly when recovery time is 1 second than when it is 100 seconds. That is, fast recovery makes adding a third replica more effective in preventing data loss.

Although we made all of the performance measurements in this chapter with 3× on-disk replication to be conservative, Figure 6.13 suggests that the combination of two copies on disk and one copy in DRAM may be enough protection against independent server failures. The main argument for 3× disk replication is to ensure 3-way redundancy even in the event of a data center power outage, which would eliminate the DRAM copies. With 3× disk replication in addition to the DRAM copy, the likelihood of data loss is extremely small: less than 1×10^{-12} per year with 1-second recovery times, even with 100,000 nodes.

6.7.1 Vulnerability to Correlated Failures

One risk with Figure 6.13 is that it assumes server failures are independent. There is considerable evidence that this is not the case in data centers [10, 15, 46]; for example, it is not unusual for entire racks to become inaccessible at once. Thus, it is important for the segment scattering algorithm to compensate for sources of correlated failure, such as rack boundaries. For example, to recover from rack failures RAMCloud should ensure that no two replicas of a segment are stored on servers in the same rack. If there are unpredictable sources of correlated failure (for example, disks, CPUs, or DRAM that share a manufacturing flaw), they will result in longer periods of unavailability while RAMCloud waits for one or more of the backups to reboot or for defective hardware to be repaired; RAMCloud is no better or worse than other systems in this respect.

Cidon et. al. [8] show that RAMCloud's random scattering (as proposed in this dissertation) will result in data loss when a small percentage of machines fail simultaneously. Unfortunately, due to power failures these types of failures happen as frequently as a few times per year. The intuition for the problem is that each segment is scattered on three servers selected at random plus its master, and for a particular segment losing the four machines it is stored on simultaneously results in data loss. In a large cluster, tens-of-millions of segments must be scattered, which creates many four-combinations of server failures that will cause data loss when they fail together. The authors provide a solution, which has been implemented in RAMCloud, that preserves the throughput benefit of scattering replicas for fast recovery and has no impact on normal case performance. They also provide analysis on how the same mechanism can be used to mitigate data loss in large GFS and HDFS [46] clusters.

6.8 Summary

The evaluation of crash recovery shows that recovery bandwidth scales as cluster size scales. With 80 nodes, equipped with flash SSDs, RAMCloud recovers data from a crashed masters at about 21 GB/s. The size of our cluster limits our experiments; however, our results show that recovery can be spread across more nodes with little overhead. As a result,

we expect that the entire contents of a 64 GB crashed master could be recovered in about 2 seconds with a cluster of 120 machines similar to those used in our experiments.

Our experiments also show that recovery time is hampered by memory bandwidth in our small cluster. This would be mitigated in a full scale RAMCloud deployment, where recovery could be spread across hundreds or thousands of machines. Consequently, we expect recovery times to improve significantly in larger clusters.

Using 8 MB segments with RAMCloud's buffered logging allows backups to utilize at least 85% of the maximum bandwidth of their storage devices for flash SSDs and traditional hard disks. RAMCloud also compensates well for differences in storage performance by scattering segments intelligently based on disk speeds. With about half of our disks operating in a severely degraded state that resulted I/O bandwidth variance of 4 $\times$, recovery time only increased by about 30%.

Extrapolating from the experiments of this chapter, we can project the limits of the current recovery implementation. A cluster of about 560 machines should be able to recover about 200 GB of data in 2 seconds. In order to keep up with growing DRAM capacities, recovery will need two improvements. First, the broadcast overhead must be reduced in order to spread recovery across more machines. Second, each recovery master will need to recover more data. With improvements to broadcast times and recovery master performance 2-second recoveries for machines with more than 1 TB of data should be possible.

Small clusters can only recover a fraction of a server's DRAM, since there a limited number of recovery masters. In small clusters, leaving part DRAM idle on some machines may make RAMCloud impractically expensive for some applications. In the future, it may make sense for RAMCloud to use some form of redundancy in DRAM as a supplement to fast recovery, so that masters can safely use more of their DRAM for data storage.

Small clusters can only recover a fraction of a server's DRAM, since they have a limited number of servers to act as recovery masters. As a result, a small cluster would need to leave most of the DRAM of its servers idle to ensure fast recovery after failures. However, this may make RAMCloud impractically expensive for some applications. In the future, it may make sense for RAMCloud to use some form of in-memory redundancy in small clusters to supplement fast recovery. This would allow masters use more of their DRAM for data storage without jeopardizing availability.

In large clusters it should be possible to recover entire servers in less than one second. Based on measured broadcast and flash I/O performance, it may be possible to recover a 64 GB server in about 330 ms using 2,000 machines. With more machines, coordination overhead begins to degrade recovery times; with less machines I/O time hampers recovery time.

By modeling data loss due to server failures in larger clusters, we have determined that under independent failures shorter recovery times are more effective at preventing data loss than increased redundancy. Our analysis shows 1-second recovery times improve reliability by about a factor of 10,000.

Chapter 7

Related Work

Data center storage systems build on the well-studied areas of distributed file systems and databases. Consequently, RAMCloud uses ideas from each of these areas, while recombining them in a new context. It draws heavily on prior work on in-memory databases, log-structured file systems, striping filesystems, and database/file server crash recovery. At the same time, RAMCloud's push for unnoticeably-fast crash recovery along with its dependence on large-scale operation prevents the straightforward application of many existing ideas.

A variety of "NoSQL" storage systems have appeared recently, driven by the demands of large-scale Web applications and the inability of relational databases to meet their needs. Examples include Dynamo [11], Cassandra [30], and PNUTS [9]. All of these systems are fundamentally disk-based and none attempt to provide latencies in the same range as RAMCloud. These systems provide availability using symmetric replication and failover to redundant copies on disk, rather than relying on fast crash recovery.

7.1 DRAM in Storage Systems

7.1.1 Caching

DRAM has historically been a critical part of storage; early database systems and operating systems as early as Unix augmented storage performance with page/buffer caching

in DRAM. With buffer caching, rather than directly accessing disk, applications request data from a DRAM cache, which is filled from storage when the requested data is not in memory. The growing performance gap between disk and DRAM, along with increasing application working sets [12], has made buffer caches an indispensable component of all of operating systems. Work on file system caching resurged in the 1980s and 1990s as distributed file systems saw wide deployment. Distributed file system clients in Echo, DEcorum, SpriteFS, and Spritely NFS compensated for the low bandwidth of early workstation networks by caching blocks in memory [23, 29, 41, 47].

More recently, many web applications have found the caching and buffer management provided by their storage systems to be inadequate and instead rely on external application-managed look-aside DRAM caches like memcached [37]. Memcached instances are often federated across many servers and allow cache size to be tuned independently of the underlying stable storage system. Facebook offers an extreme example of this approach to large-scale DRAM caching; it offloads database servers with hundreds of terabytes DRAM cache spread across thousands of machines running memcached [39]. In short, caches are growing, and DRAM is playing an ever-increasing role in storage systems.

Unfortunately, even with aggressive caching, performance in these systems is still limited by disks. For example, due to the performance gap between disk and DRAM, an application with a cache hit rate of 99% would still result in a 10× slowdown compared to a 100% cache hit rate (that is, when all data is kept in DRAM). Furthermore, cache misses result in access times 1,000× worse than hits, which makes cache performance difficult to predict. RAMCloud avoids these problems, since it is not a cache; all operations are serviced from DRAM. Access times are more predictable and are only limited by networking latency.

Caching mechanisms like memcached appear to offer a simple mechanism for crash recovery; if a cache server crashes, its contents can simply be reloaded on demand, either on the crashed server (after it restarts) or elsewhere. However, in large-scale systems, caching approaches can cause large gaps in availability after crashes. Since these systems depend on high cache hit rates to meet their performance requirements, if caches are flushed, then the system may perform so poorly that it is essentially unusable until the cache has refilled. Unfortunately, these caches are large and take a long time to refill, particularly since

they are typically reloaded using small, random disk I/Os. This happened in an outage at Facebook in September 2010 [26]: a software error caused 28 TB of memcached data to be flushed, rendering the site unusable for 2.5 hours while the caches refilled from slower database servers. Nishtala et. al. [39] further describe these cascading failures due to overload when cache servers fail. They describe reserving 1% of their memcached servers as standbys that are swapped into service when a cache server fails. The result is a 10-25% reduction in requests to database servers due to server failures. RAMCloud avoids these issues by recovering full performance after failures in 1 to 2 seconds. It never operates in a degraded state.

Some primarily disk-based data center storage systems allow in-memory operations on parts of datasets. For example, Bigtable [7] is a data center storage system in wide use at Google that allows entire "column families" to be loaded into memory. By contrast, RAMCloud keeps all data in DRAM, so all design choices in RAMCloud are optimized for in-memory operation; RAMCloud provides no means to query data stored on disk.

7.1.2 Application-Specific Approaches

Recently, some large-scale web applications have found fast and predictable access important enough to build application-specific approaches to storing entire datasets in DRAM that span many machines. Web search is an example; search applications must sustain high throughput while delivering fast responses to users. A single user query may involve RPCs to hundreds or thousands of machines. Even if just 1% of RPCs resulted in disk access, nearly every response to a user query would be delayed. Consequently, both Google and Microsoft keep their entire Web search indexes in DRAM, since predictable low-latency access is critical in these applications. These systems use DRAM in adhoc ways; new services and data structures are developed to expose DRAM for each new application. In contrast, RAMCloud exposes DRAM as a general-purpose storage system. One of RAMCloud's goals is to simplify the development of these types of applications.

7.1.3 In-memory Databases

Designs for main-memory databases [13, 16] emerged in the 1980s and 1990s as DRAM grew large enough for some datasets (about 1 GB), though these designs were limited to a single machine. Recently, interest in main-memory databases has revived, as techniques for scaling storage systems have become commonplace. One example is H-Store [28], which keeps all data in DRAM, supports multiple servers, and is general-purpose. However, H-Store is focused more on achieving full RDBMS semantics and less on achieving large scale or low latency to the same degree as RAMCloud. H-Store keeps redundant data in DRAM and does not attempt to survive coordinated power failures.

Avoiding Disk-Limited Performance

Early single-machine in-memory databases pioneered techniques to leverage disks for durability without incurring high disk write latencies on the fast path. Like RAMCloud, a key goal of these systems was to expose the full performance of DRAM-storage to applications, and logging was a key component of the solution. Unlike RAMCloud, these logs were not the final home of data. Instead, the DRAM state was periodically checkpointed to disk, at which point these "commit" logs were truncated. As a result, these databases wrote all data to disk twice: once as part of the log and once as part of a checkpoint. This increased their consumption of disk bandwidth. In RAMCloud, the log is the location of record for data; requests for data are always serviced directly from the log. This has a substantial difference in how recovery is performed; recovery in RAMCloud consists purely of log replay. There is never a checkpoint to load. Also, RAMCloud reclaims log space using a log cleaner, rather than relying on checkpointing and truncation.

One key technique these systems used for taking full advantage of disk bandwidth was *group commit*, which batched updates and flushed them to disk in bulk [13, 17, 19, 33]. Group commit allowed these databases to operate at near-maximum disk bandwidth, but since transaction commit was delayed, update operations still incurred full disk latency. Group commit played a similar role to RAMCloud's segments, and allowed many operations to be flushed to disk in a single contiguous operation. Group commit is insufficient for RAMCloud, since synchronous disk writes are incompatible with its latency goals.

To hide the latency mismatch between disks and DRAM, several of these early in-memory databases also used or proposed non-volatile memory for log buffering [13, 14, 16, 31]. Baker suggested a similar solution in the context of the Sprite distributed filesystem [4], which used battery backed RAM to allow safe write caching. These earlier works found that even a modest amount of non-volatile storage (a megabyte or less) is enough to hide storage latency and achieve high effective throughput. RAMCloud draws a similar conclusion and only requires the most recently written segment replicas to be buffered for high normal-case performance. RAMCloud provides stronger fault-tolerance than these single-machine systems, since it replicates data in non-volatile buffers on other servers in the clusters, whereas these systems stored a single copy of data locally.

7.2 Crash Recovery

7.2.1 Fast Crash Recovery

Many of the advantages of fast crash recovery were outlined by Baker [3, 5] in the context of distributed file systems. Her work explored fast recovery from server failures in the Sprite distributed filesystem. Restarting after server failures in SpriteFS required recovering cache state distributed across clients. Baker noted that fast recovery

- may constitute continuous availability if recovery time is short enough,
- may be more cost-effective than redundancy or high-reliability hardware,
- may be simpler than engineering reliability into normal operation,
- and may have no impact on normal case performance.

RAMCloud's fast crash recovery is motivated precisely by these advantages, and its reliance on DRAM amplifies each of these benefits.

File system server cost was a key constraint for Baker's work in the 1990s, but data centers have made redundant and large scale use of hardware cost effective. Baker achieved recovery times of 30 seconds for failed distributed file system servers. RAMCloud is able to improve this by an order of magnitude even while performing more work by relying on

scale. Baker's recovery times were limited almost entirely by server reboot time; RAMCloud sidesteps this problem by storing data redundantly across the cluster and relocating data to new machines when servers fail. Such a solution would have been impractical in 1994, but is natural in today's data centers where machines are abundant.

7.2.2 Striping/Chunk Scattering

Striping data across multiple disks or nodes has been used in several contexts for improved reliability and throughput. Garcia-Molina and Salem [16] observe that the write throughput of a durable in-memory database is limited by local disk throughput, so they suggest striping commit logs across many disks. RAMCloud's primary motivation is high read bandwidth during recovery rather than high write bandwidth, but striping across all of the disks of the cluster also affords masters high burst write bandwidth as well. The Zebra [20] distributed filesystem recorded client operations in a log, which it striped across multiple storage servers.

More recently, striping has become ubiquitous in data center filesystems like the Google File System (GFS) [18], the Hadoop Distributed File System (HDFS) [46], and Windows Azure Storage [6]. These systems provide specialized support for applications to use them as a high-bandwidth high-reliability log. As applications log data, it is divided into chunks that are replicated and scattered across a cluster; these chunks are equivalent to RAMCloud masters' segments, which are also used as a unit of replication and scattering. These systems play a similar role to RAMCloud's backups, and they provide durability for data center storage systems like Bigtable [7], HBase [21], and Windows Azure Tables [6]. RAMCloud is even more dependent on striping than these disk-based database systems, since high read bandwidth is a prerequisite for fast crash recovery.

7.2.3 Partitioned Recovery

While discussing single-node in-memory database recovery, Garcia-Molina and Salem [16] observe that local DRAM bandwidth can be a barrier to fast recovery. They note that if a database checkpoint is striped across many disks, then each disk needs an independent path to memory to be most useful. This same insight drives RAMCloud to partition recovery

across all of the nodes of the cluster. Since each backup reads a small fraction of a crashed master's replicas, each of the disks has an independent path to memory. Similarly, each recovery master recovers a single partition of a crashed master, which divides data flowing from storage across the network, CPU, and memory bandwidth of many hosts.

Bigtable uses a partitioned approach to recovery that shares many similarities to RAMCloud's recovery. It distributes recovery by reassigning tablets across many other servers in the cluster. Tablets are kept small (100 to 200 MB) by splitting them as they grow, so the work of recovery can be distributed relatively evenly among all the nodes of the cluster. RAMCloud does not pre-divide or pre-partition masters' tablets; instead, it allows tablets to grow to the full size of DRAM, and it only decides how tablets will be divided at recovery time.

Similar to RAMCloud recovery, Bigtable servers must replay data from a commit log that contains objects intermingled from multiple tablets. During normal operation, each Bigtable server records update operations in a single commit log. Using a single log for all the tablets on a single server improves the effectiveness of group commit, which reduces disk seeks and improves throughput. During recovery, a failed server's commit log is first divided and cooperatively sorted by the cluster. This allows servers recovering a tablet to read just the relevant entries of the failed server's commit log with a single disk seek. Each RAMCloud master similarly intermingles log records for all of its assigned tablets during normal operation. Similar to Bigtable's sort step, RAMCloud backups bucket objects in each replica according to a partition scheme determined at the start of recovery; RAMCloud performs this bucketing in-memory, whereas Bigtable sorts commit log data on disk.

7.2.4 Reconstruct Versus Reload

Metadata and indexing structures pose an interesting question for recovery systems; should these structures be stored durably along with data and reloaded after a failure or is it better reconstruct these structures from the stored data itself? For example, Bigtable stores indexes explicitly on disk along with object data. Servers reload the indexes into DRAM when they are assigned a tablet for recovery. In RAMCloud, a master's index (that is,

its hash table) is only stored in DRAM and is lost when the master crashes. Instead, recovery masters rebuild hash table entries, since replaying data already in DRAM is inexpensive. Rebuilding such indexes would be prohibitive in most disk-based systems like Bigtable, since data left behind on storage is large (presumably, much larger than a machine's DRAM) and is only read on demand.

This same question arises for replicated segment/chunk locations; should chunk locations be explicitly stored somewhere or is it better to broadcast and reconstruct the chunk location mapping from responses? Bigtable and RAMCloud both reconstruct this mapping after failures. Bigtable uses GFS for durability. Similar to RAMCloud masters, each filesystem "master" in GFS and HDFS only tracks chunk replica locations in DRAM, which is lost when it crashes. When a new filesystem master is chosen, it rediscovers the replica locations as chunk servers detect the old master's failure and re-enlist with it. The RAMCloud coordinator broadcasts during recovery for the same reason.

7.3 Log-structured Storage

RAMCloud's log-structured approach to storage management is similar in many ways to log-structured file systems (LFS) [42]. However, log management in RAMCloud is simpler and more efficient than in LFS. RAMCloud is simpler because the log need not contain metadata to enable random-access reads as in LFS. Instead, each master uses a hash table to enable fast access to data in DRAM. During recovery, a master's lost hash table is reconstructed by replaying the entire log, rather than having the hash table explicitly stored in the log. RAMCloud does not require checkpoints as in LFS, because it replays the entire log during recovery. Log size is bounded by the DRAM capacity of a machine, which indirectly bounds recovery time as well. RAMCloud is more efficient than LFS because it need not read data from disk during cleaning: all live data is always in memory. The only I/O during cleaning is to rewrite live data at the head of the log; as a result, RAMCloud consumes less bandwidth for cleaning than LFS [43]. Cleaning cost was a controversial topic for LFS; see [45], for example.

7.4 Randomized Load Balancing

Mitzenmacher and others have studied the theoretical properties of randomization with refinement and have shown that it produces near-optimal results [2, 38]. RAMCloud uses similar techniques in two places. First, the coordinator uses randomness with refinement to achieve near-uniform partition balance across recovery masters. Second, masters randomly scatter segment replicas across the disks of the cluster with a refinement that weights selection according to storage bandwidth.

Chapter 8

Conclusion

This dissertation described and evaluated scalable fast crash recovery, which is key in providing durability and availability in the RAMCloud data center storage system. When servers fail in RAMCloud, the work of recovery is split across all of the nodes of the cluster in order to reconstitute the DRAM contents of the failed server as rapidly as possible. By fully recovering the contents of a failed server in a few seconds, fast crash recovery provides an illusion of continuous availability; client applications see server failures as brief storage system performance anomalies.

Several key techniques allow RAMCloud to recover quickly from failures while providing low-latency normal operation and cost-effective durability.

- Most importantly, it harnesses the large scale resources of data center clusters. When a master fails in RAMCloud, data is read from thousands of disks in parallel and processed by many recovery masters working in a tightly pipelined effort to reconstitute the contents of the failed server. RAMCloud takes advantage of all of the CPUs, NICs, and disks of the cluster to recover data as quickly as possible.

- It eliminates stragglers that limit recovery time by balancing the work of recovery evenly across all of the machines of the cluster. No backup or recovery master is saddled with a workload that takes significantly longer than the others.

- It avoids scalability-limiting centralization both during normal operation and during

recovery. Each backup and each recovery master work independently without coordination.

- It uses randomization techniques to allow localized decision making that results globally balanced work. In particular, it achieves improved balance for disk and log replay work by using random choice with refinement.

- It uses log-structured storage techniques that provide durability while avoiding disk-limited performance on the fast path.

Experiments show an 80-node cluster recovers 40 GB of data from a failed server in 1.86 seconds and that the approach scales to recover more data in the same time as cluster size grows. Consequently, we expect a cluster of 120 machines can recover the entire DRAM of a failed 64 GB node in 1 to 2 seconds.

Overall, RAMCloud's scalable fast crash recovery makes it easy for applications to take advantage of large scale DRAM-based storage by making the system failure transparent while retaining its low latency access times. Fast crash recovery provides RAMCloud with the reliability of today's disk-based storage systems for the cost of today's volatile distributed DRAM caches with performance 50-5000$\times$ greater than either.

8.1 Future Directions for Fast Crash Recovery

In order for fast recovery to reconstitute the full DRAM of future data center servers, recovery will need multicore and non-uniform memory access (NUMA) optimizations. Recent CPU architectures have only seen modest gains in single core performance. Applications now must intelligently divide work across cores and partition memory accesses among core-specific memory controllers for the best performance. At the same time, per-server DRAM capacity continues to grow, so for the same number of recovery masters to recover larger servers data processing will have to be split across cores. It may be that work can be divided in a rather coarse-grained way, with several rather independent recovery masters working within each server, or it may require more fine-grained optimization if, for example, the single hash table and single in-memory log on each master is to be preserved.

Improving per-server recovery performance alone will be insufficient to compensate for the growth in per-server DRAM capacity. To retain fast recovery times in the future, server communication overheads will have to be improved. Network latency is increasingly dominated by speed-of-light delays, so the latency of network hardware is unlikely to improve significantly in the future. This means recovery will need to restructure communication patterns to minimize network delays. As one example, the simple, iterative approach to broadcast that the coordinator uses during recovery may need to be converted to use a tree-like communication structure.

Using in-memory redundancy to avoid unavailability is not incompatible with fast recovery. One interesting direction for future work is to extend RAMCloud to support a minimal amount of in-memory redundancy to avoid the 1 to 2 second availability gaps caused by fast crash recovery. In such an approach, fast recovery would still be important, since aggressively limiting redundancy would make the system more vulnerable to availability gaps if failures occurred during recovery. Minimizing the capacity and performance overhead of redundancy would be critical, but recent coding techniques like Local Reconstruction Codes [24] achieve capacity overheads of 1.33× with an effect stronger than triplicating data. Significant rework to how data is organized on masters would be required to effectively take advantage of coding.

8.2 Lessons Learned About Research

From the outset, the goal of the RAMCloud project was to produce a usable and deployable software artifact rather than just a research reference design. As a result, a large fraction of student effort went into practical design, software engineering, documentation, and testing. As of November 2013, RAMCloud had accumulated 50,000 lines of source C++ code with an additional 35,000 lines of unit tests. Of the 85,000 lines about 30,000 were documentation and comments alone.

One key question raised by projects like RAMCloud is whether building polished software projects is worth the time cost. Is it more important for a student to mature as an engineer and to spend his efforts building systems, testing them, documenting them, supporting them, and making them generally useful? Or is it more important for a student to

mature creatively and to spend his efforts learning how to set and pursue a research agenda? Or perhaps this is a false choice; perhaps one can optimize for both.

Personally, the major benefit from extensive engineering on RAMCloud is that I have gained practical skill in building large-scale software and complex systems that work. Furthermore, this experience has helped me realize that even the most challenging problems can be tackled with a motivated and energetic team. Hopefully, this realization will encourage me to set into tough goals in the future. I certainly benefited greatly from collaborating closely on a large project.

Even retrospectively, it is hard to draw conclusions about RAMCloud's stringent development style. In the future, I will encourage students to build software that they can take pride in. At the same time, I believe a more lightweight approach to design and prototyping could have been beneficial at certain points in the project. Allowing students to "hack up" features could create energy behind new ideas and allow for fast exploration.

Developing RAMCloud was sometimes an arduous process. Three aspects of RAMCloud's recovery system made progress on developing it challenging; it is complex, distributed, and performance-sensitive. The combination of these three attributes meant regressions were common yet hard to track down. RAMCloud's recovery performance is intricately dependent on nearly every subsystem, including the RPC system, the hash table, the in-memory log, the distributed log replication system, the backup service, the I/O subsystem, the task management and threading model, server status broadcasts, and the failure detector. Even hardware was an adversary, with fans affecting disk performance, inexplicable losses in I/O performance, and unreliable networking due to faulty NIC firmware upgrades and PCI Express incompatibilities. Over the few years since recovery was prototyped, nearly every non-trivial change to any of RAMCloud's subsystems has caused a regression in either recovery performance or normal-case logging performance. Extending the prototype to improve modularity and cover all of the practical failure cases that arise in a real cluster was a slow process, requiring performance to be reconsidered at each step. Over time, experience helped to mitigate some of this difficulty, but it is clear that despite decades of systems research, building fast and reliable distributed systems is still an art.

8.3 Final Comments

Scalable fast crash recovery is the cornerstone that makes RAMCloud a viable data center storage system. It provides the durability, availability, and fault tolerance that make RAMCloud practical and simple for developers to use. It makes RAMCloud cost effective by eliminating the need for replication in DRAM. Finally, it does all this without impacting normal-case performance, which is key to enabling new data center applications that can interact with datasets more intensively than has been possible in the past.

Bibliography

[1] AGIGARAM. AgigA Tech AGIGARAM, October 2013. URL http://www.agigatech.com/agigaram.php. 15

[2] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations (extended abstract). In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 593–602, New York, NY, USA, 1994. ACM. ISBN 0-89791-663-8. 30, 120

[3] Mary Baker and John K. Ousterhout. Availability in the Sprite Distributed File System. *Operating Systems Review*, 25(2):95–98, 1991. 116

[4] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 10–22, New York, NY, USA, 1992. ACM. ISBN 0-89791-534-8. 116

[5] Mary Louise Gray Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1994. 116

[6] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure

Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. 22, 117

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. 114, 117

[8] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, USENIX ATC'13, pages 37–48, Berkeley, CA, USA, 2013. USENIX Association. 31, 109

[9] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment*, 1:1277–1288, August 2008. ISSN 2150-8097. 112

[10] Jeffrey Dean. Evolution and Future Directions of Large-Scale Storage and Computation Systems at Google. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 1–1, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. 109

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. 112

[12] Peter J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968. ISSN 0001-0782. 113

[13] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. Implementation Techniques For Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM. ISBN 0-89791-128-8. 115, 116

[14] Margaret H. Eich. Mars: The Design of a Main Memory Database Machine. In *Database Machines and Knowledge Base Machines*, volume 43 of *The Kluwer International Series in Engineering and Computer Science*, pages 325–338. Springer US, 1988. ISBN 978-1-4612-8948-7. 116

[15] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *OSDI*, pages 61–74. USENIX Association, 2010. ISBN 978-1-931971-79-9. 108, 109

[16] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4:509–516, December 1992. ISSN 1041-4347. 115, 116, 117

[17] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8(2):3–10, 1985. 115

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. 107, 108, 117

[19] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems*

Principles, SOSP '87, pages 155–162, New York, NY, USA, 1987. ACM. ISBN 0-89791-242-X. 115

[20] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 29–43, New York, NY, USA, 1993. ACM. ISBN 0-89791-632-8. 117

[21] HBase. Hbase, October 2013. http://hbase.apache.org/. 117

[22] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, July 1990. ISSN 0164-0925. 73

[23] Andy Hisgen, Andrew Birrell, Timothy Mann, Michael D. Schroeder, and Garret Swart. Availability and Consistency Tradeoffs in the Echo Distributed File System. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*, pages 49–54, 1989. 113

[24] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 15–26, Berkeley, CA, USA, 2012. USENIX Association. 123

[25] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIX ATC '10, pages 145–158, Berkeley, CA, USA, 2010. USENIX Association. 13, 71, 75

[26] Robert Johnson. More Details on Today's Outage - Facebook, September 2010. URL http://www.facebook.com/note.php?note_id=431441338919. 114

[27] Robert Johnson and J. Rothschild. Personal Communications, March 24 and August 20, 2009. 1

[28] 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. H-store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1:1496–1499, August 2008. ISSN 2150-8097. 115

[29] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas. DEcorum File System Architectural Overview. In *Proceedings of the Summer 1990 USENIX Conference*, pages 151–164, 1990. 113

[30] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010. ISSN 0163-5980. 112

[31] Tobin J. Lehman and Michael J. Carey. Query Processing in Main Memory Database Management Systems. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 239–250, New York, NY, USA, 1986. ACM. ISBN 0-89791-191-1. 116

[32] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, October 1985. ISSN 0018-9340. 80

[33] Kai Li and Jeffrey F. Naughton. Multiprocessor Main Memory Transaction Processing. In *Proceedings of the First International Symposium on Databases in Parallel and Distributed Systems*, DPDS '88, pages 177–187, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. ISBN 0-8186-0893-5. 115

[34] Bill Lynn and Ansh Gupta. Non-Volatile CACHE for Host-Based RAID Controllers, January 2011. URL http://www.dell.com/downloads/global/products/pvaul/en/NV-Cache-for-Host-Based-RAID-Controllers.pdf. 15

[35] Krishna T. Malladi, Benjamin C. Lee, Frank A. Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. Towards Energy-proportional Datacenter Memory with Mobile DRAM. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 37–48. IEEE Press, 2012. 7

[36] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A Coherent Distributed File Cache with Directory Write-Behind. *ACM Transactions on Computer Systems*, 12(2):123–164, 1994.

[37] memcached. memcached: a distributed memory object caching system, October 2013. http://www.memcached.org/. 1, 113

[38] Michael David Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1996. 30, 120

[39] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pages 385–398. USENIX Association, 2013. 113, 114

[40] 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. The Case for RAMCloud. *Communications of the ACM*, 54:121–130, July 2011. ISSN 0001-0782. 6, 11

[41] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21: 23–36, February 1988. ISSN 0018-9162. 113

[42] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10:26–52, February 1992. ISSN 0734-2071. 4, 16, 43, 119

[43] Stephen Mathew Rumble. *Memory and Object Management in RAMCloud*. PhD thesis, Stanford University, Stanford, CA, USA, 2014. 16, 44, 119

[44] Pierluigi Sarti. Battery Cabinet Hardware V1.0, July 2011. URL http://www.opencompute.org/wp/wp-content/uploads/2011/07/DataCenter-Battery-Cabinet-Specifications.pdf. 14, 15, 25

[45] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proceedings of the USENIX 1995 Technical Conference*, TCON'95, pages 249–264, Berkeley, CA, USA, 1995. USENIX Association. 119

[46] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. 31, 109, 117

[47] V. Srinivasan and Jeffrey C. Mogul. Spritely NFS: Experiments with Cache-Consistency Protocols. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 44–57, New York, NY, USA, 1989. ACM. ISBN 0-89791-338-8. 113