RAMCube: Exploiting Network Proximity for RAM-Based Key-Value Store

Yiming Zhang[†*], *Chuanxiong Guo*[‡], *Rui Chu*[†], *Yongqiang Xiong*[‡], *Haitao Wu*[‡], *Guohan Lu*[‡]
[†]*National University of Defense Technology*
[‡]*Microsoft Research Asia*
[*]*University of Cambridge*
{*ymzhang, rchu*}*@nudt.edu.cn*, {*chguo, yqx, hwu, lguohan*}*@microsoft.com*

Abstract

Disk-based storage is becoming increasingly problematic in meeting the needs of large-scale cloud applications. Recently RAM-based storage is proposed by aggregating the RAM of thousands of commodity servers in data center networks (DCN). These studies focus on improving performance with high throughput I/O, low latency RPC and fast failure recovery. RAM-based storage brings great DCN-related challenges, for example, false server failure detection due to network problems, traffic congestion during failure recovery, and top-of-rack (ToR) switch failures.

This paper presents *RAMCube*, a DCN-oriented design for RAM-based key-value store based on the BCube network [24]. RAMCube exploits network proximity to restrict all failure detection and recovery traffic within one-hop neighborhood, and leverages BCube's multiple paths to handle switch failures. Prototype implementation and experimental evaluation demonstrate that RAMCube is promising to achieve reliable, high performance I/O and fast failure recovery in large-scale data centers.

1 Introduction

Disk-based storage is becoming more and more problematic in meeting the needs of large-scale cloud systems in terms of I/O latency and bandwidth. As a result, in recent years we see an increasing trend of migration of data from disks to random access memory (RAM) in storage systems. For example, memcached [3] is an in-memory key-value store that has been widely used by a number of Web service providers, including Facebook, Twitter, and Youtube, to offload their storage servers. Google and Microsoft keep entire search indexes in RAM [28], and Google's Bigtable keeps certain columns (or even entire column family) in RAM [15].

Keeping data in RAM brings great challenges to reliable data access. Cache-based approaches (like memcached) cause difficulties for applications to effectively utilize RAM. E.g., it is the responsibility of applications to manage consistency between caches and storage, making it vulnerable to consistency problems. Most recently, RAMCloud [28, 29] is proposed as a RAM-based key-value store where data is kept entirely in the RAM of storage servers. It achieves fast server failure recovery by scattering backup data across a large number of disks and reconstructing lost data in parallel across high-bandwidth (but expensive) InfiniBand networks.

RAM-based storage has a number of benefits such as low latency RPC and high-throughput I/O. For example, a RAM-based storage system can provide 100-1000x greater throughput than disk-based systems [29]. Moreover, the applications need no longer manage the consistency between RAM and a separate backing store. To achieve practical RAM-based storage in data centers, however, many realistic DCN-related issues need to be addressed. (i) It is difficult to quickly distinguish temporary network problems from server failures across a large-scale network. (ii) The large number (up to thousands) of parallel unarranged recovery flows is likely to bring traffic congestion, resulting in unexpected recovery delay. (iii) Top-of-rack (ToR) switch failures bring great difficulty to fast failure recovery.

This work describes *RAMCube*, a DCN-oriented design for RAM-based key-value store that supports thousands or tens of thousands of servers to offer up to hundreds of terabytes of RAM storage. In this paper, we follow the technical trend that large data centers are constructed using commodity Ethernet switches, and use Ethernet-based BCube [24] as the underlying network of RAMCube.

The key idea of RAMCube is to have a codesign in which the storage system and the the underlying network can make joint efforts to achieve efficient RAM-based storage. Specifically, RAMCube exploits the proximity of BCube network to construct a symmetric *Multi-Ring* structure, restricting all failure detection and recovery traffic within one-hop neighborhood, which addresses the aforementioned problems including false failure detection and recovery traffic congestion. In addition, RAMCube leverages BCube's multiple paths between any pairs of servers to handle switch failures.

Note that although our RAMCube is designed based on BCube, a well-known network architecture for data centers [24], we believe that the *general* idea of **one-hop** failure discovery and recovery is promising and could be explored in many other DCN topologies such as hypercube [13], MDCube [32], and *k*-ary *n*-cube [34].

The rest of the paper is organized as follows. Section 2 discusses background and challenges. Section 3 presents RAMCube structure. Section 4 introduces failure detection and recovery. Section 5 addresses other design issues. Section 6 introduces prototype implementation and experiments. Section 8 introduces related work. Finally, Section 9 concludes the paper.

2 Preliminaries

In this section, we first discuss the challenges of the RAMCube design, and then briefly introduce the BCube network on which RAMCube is based.

2.1 Challenges

Fast failure recovery is crucial to improve availability [21] and durability [28] in RAM-based storage systems. Current studies, like RAMCloud, realize fast failure recovery by using aggressive data partitioning [15, 23], a distributed approach that scatters backup data across hundreds or thousands of disks on backup servers, and quickly reconstructs lost data in the RAM of hundreds of servers. In large-scale data centers, however, many network related issues make it challenging to recover a failed storage server with several tens of GB RAM in a short period of time (e.g., a few seconds). These challenges mainly include false failure detection due to temporary network problems, traffic congestion during recovery, and ToR switch failures.

False failure detection. In order to quickly recover from a storage server failure (e.g., in a few seconds), the timeout of heartbeat messages should be relatively short (at most a few hundred milliseconds). However, various circumstances like incast and temporary network partitions may make heartbeats be discarded (in Ethernet) or suspended (in InfiniBand), making it difficult to be distinguished from real server failures. Although false failure detection is not fatal (which is discussed in Section 4), the recovery of several tens of GB data is definitely very "expensive". Since network problems cannot be completely avoided in any large-scale systems, our solution to this problem is to shorten the paths that heartbeat messages have to traverse, reducing the chances of encountering network problems. Ideally, if the servers only need to inspect the status of directly-connected neighbors, then we can minimize the possibility of network-induced false server failure detection.

Recovery traffic congestion. Recovering several tens of GB data in a short period of time (e.g., 1$\sim$2 seconds) requires an aggregate recovery bandwidth of tens of GB/sec both for disks and for networks. This means that hundreds or even thousands of servers will be involved in the recovery of a failed server. If the distributed recovery takes place in a random and unarranged manner and the recovery flows traverse long paths, it will bring hot spots in the network and result in unexpected long recovery delay. Even in multi-rooted trees with the highest-end switches, this can also bring severe congestion because per-flow ECMP generates collisions due to mapping multiple flows into one path [9]. To address this problem, our solution is to restrict the recovery traffic within an area as small as possible. Ideally, if the recovered data is only sent to directly-connected neighbors, then the possibility of congestion will be minimized, and it will also be easy to control the recovery traffic.

ToR switch failures. In data centers a rack usually contains several tens of servers that are connected to a ToR switch. In previous studies [28] when a ToR switch fails, all the servers connected to it are considered failed and several TB data has to be recovered. The recovery storm takes much more time than recovering a single server failure does, and this would become worse if we consider the congestion among the tens of thousands of recovery flows. Since all the servers connected to a failed switch are actually "alive", our solution to this problem is to construct the RAM-based storage system with a multi-homed topology, i.e., each server connects to multiple switches. When one switch fails, the servers can utilize other paths to remain connected and thus no urgent failure recovery is needed.

Based on the above discussion, we design RAMCube on top of the BCube network [24] (which will be introduced in the next subsection), so that we can exploit the proximity of BCube to restrict all failure detection and recovery traffic within one-hop neighborhood, and use BCube's multiple paths to handle switch failures.

2.2 BCube

BCube [24] is a server-centric network architecture, where servers with multiple network ports connect to multiple layers of switches. Servers act as not only end hosts, but also relay nodes for each other. BCube supports various bandwidth-intensive applications, and exhibits graceful performance degradation as the servers and/or switches fail.

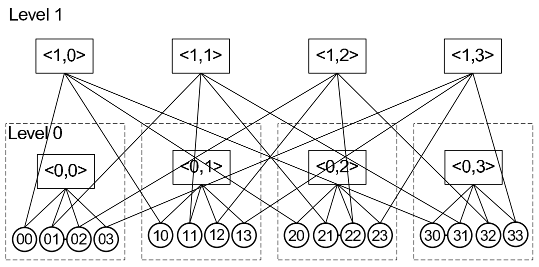

Figure 1: An example of BCube$(4,1)$ [24].

BCube is recursively defined. A BCube$(n,0)$ is simply n servers connecting to an n-port switch. A BCube$(n,1)$ is constructed from n BCube$(n,0)$ and n n-port switches. More generically, a BCube(n,k) ($k \geq 1$) is constructed from n BCube$(n,k-1)$ and n^k n-port switches. Each server in a BCube(n,k) has $k+1$ ports, which are numbered from level-0 to level-k. BCube(n,k) has $N = n^{k+1}$ servers and $k+1$ levels of switches, with each level having n^k n-port switches. Fig. 1 shows an example of BCube$(4,1)$, which is constructed from four BCube$(4,0)$ and four level-1 4-port switches.

BCube's software-based routing approach suffers from high CPU overhead and processing latency. To address this problem, most recently Lu et al. design and implement *ServerSwitch* [27], a programmable commodity switching chip that supports high-performance BCube routing and achieves very low CPU overhead, high throughput and low processing latency.

3 The RAMCube Structure

This section first introduces the basics of RAMCube, and then presents the design of RAMCube structure.

3.1 Basics

Large-scale cloud systems usually contains thousands of servers that can be divided into two categories: application servers implementing application logic, and storage servers providing long-term shared storage for the application servers. RAMCube is designed to aggregate the RAM of the storage servers, each with tens of GB RAM, into a single RAM-based key-value store with totally hundreds of TB RAM. This subsection briefly discusses some design choices of RAMCube in network hardware, data model, and structure.

Network hardware. Network hardware is an important factor that influences the performance of a RAM-based storage system. InfiniBand is featured by its high bandwidth, low latency, as well as high price. For example, in a small-scale InfiniBand testbed, RAMCloud claims 400$\sim$800 MB/sec recovery bandwidth per NIC [28]. However, in this paper we choose to follow the technical trend that almost all large-scale data centers are constructed using commodity Ethernet switches. Our vision is that high-performance Ethernet is more promising and cost-effective than InfiniBand for data centers. Recent technology trends show that Ethernet switches with 40 Gbps bandwidth [6] and sub-μs latency [5] are practical in the near future. Therefore, We design RAMCube on top of Ethernet-based BCube [24].

Data model. The current data model in RAMCube is a simple key-value store that supports tables containing arbitrary number of key-value pairs. A key-value pair consists of a variable-length (up to 1 KB) key and a variable-length (up to 1 MB) value. RAMCube provides a simple set of operations ("*set* key value", "*get* key" and "*delete* key") for writing/updating, reading and deleting data. In the future RAMCube will extend the data model with support of more powerful features such as indexes and super columns [15].

Primary-recovery-backup. For durability, RAM-based storage system has multiple copies for each key-value pair. There are two choices, namely *symmetric replication* [17] and *primary-backup* [14], to maintain consistency in normal read/write operations. In symmetric replication all copies of a key-value pair have to be kept in the RAM of different servers and a quorum-like technique [20] is used for conflict resolution. In contrast, in primary-backup only one primary copy is needed to be stored in RAM with redundant backup copies being stored in disks, and all read/write operations are through the primary copy. Considering the relatively high cost and energy usage per bit of RAM, primary-backup is preferred for RAM-based storage [29].

We refer to the servers storing the primary copies in RAM as *primary servers*, and the servers storing the backup copies in the disks as *backup servers*. Considering the typical bandwidth of disks (100$\sim$200 MB/sec), if we want to recover a primary server failure in a short period of time (a few seconds), one primary server with tens of GB RAM needs at least several hundred backup disks. Once having been read from disks of backup servers, the backup data should be recovered to as *few* as possible healthy servers since fragmentation changes the locality of original data and might degrade application performance after recovery. The healthy servers that accommodate the recovered data are called *recovery servers*. Considering the currently available commodity NIC bandwidth (10 Gbps), at least several tens of recovery servers are needed for recovering a failed primary server with tens of GB RAM in a few seconds. This "primary-recovery-backup" structure [15, 28] is depicted in Fig. 2(a). Note that each server symmetrically acts as

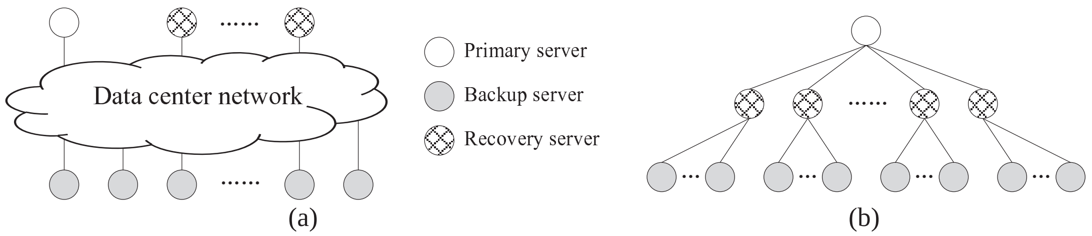

Figure 2: (a) Primary-recovery-backup structure [28]. (b) Directly connected tree in RAMCube.

all the three roles at the same time.

"Primary-recovery-backup" requires **fast failure recovery**. First, fast failure recovery is crucial to improve *availability*, which is defined as MTTF / (MTTF + MTTR) [21] where MTTF and MTTR respectively refer to "mean time to failure" and "mean time to recovery". Second, fast failure recovery is also important to improve *durability*. Previous studies [28] show that assuming two backup copies for each primary copy and two failures per year per server with a Poisson arrival distribution in a 10,000-server RAM-based storage system, the probability of data loss in one year is about 10^{-6} if the recovery is finished in 1 second; and the probability is about 10^{-4} when the recovery delay is 10 seconds.

3.2 RAMCube MuitiRing Construction

The basic idea of RAMCube for addressing the challenges discussed in Section 2.1 is to leverage network *proximity* to restrict all failure detection and recovery traffic within one-hop neighborhood. We improve the primary-recovery-backup structure (shown in Fig. 2(a)) with a *directly* connected tree (shown in Fig. 2(b)), where a primary server has multiple directly connected recovery servers, each of which corresponding to multiple directly connected backup servers. Clearly, Fig. 2(b) can be viewed as a special case of Fig. 2(a).

In Fig. 2(b), the primary server periodically sends heartbeat messages to all its recovery servers, and once the recovery servers detect (with certain mechanisms described in the next section) the primary server fails, they will start a recovery procedure reading backup data from their directly connected backup servers. Clearly, here the recovery servers play an important role both in failure detection and in failure recovery: since the recovery servers directly connect to the primary server, they can eliminate much of the possibility of false failure detection; and since the recovery servers also directly connect to the backup servers, the recovery traffic is guaranteed to have little overlap or congestion.

The directly connected tree provides great benefit for accurate failure detection and fast failure recovery. In order to apply it to RAMCube, we need *symmetrically* map the tree onto the entire network, i.e., each server equally plays all the three roles of primary server, recovery server and backup server. Our insight is that for BCube if we replace each switch and its n links with an $n \times (n-1)$ full mesh that directly connects the servers, we will get a generalized hypercube [13]. Then, we can construct the multi-layer logical rings (*MultiRing* for short) for symmetrically mapping the tree as depicted in Fig. 3.

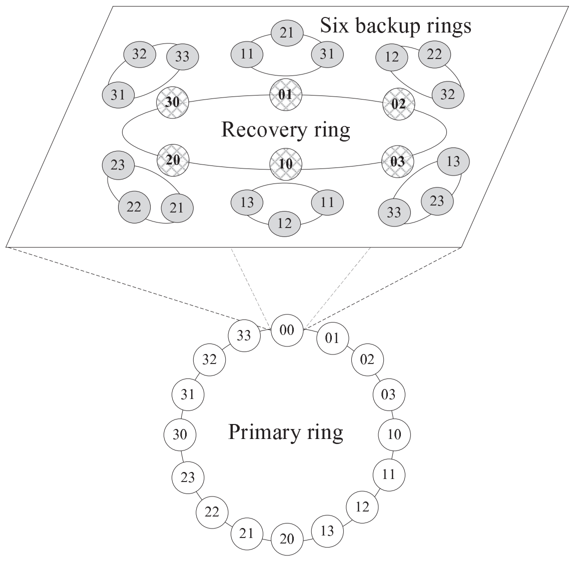

Figure 3: The primary ring (1st layer ring) of BCube$(4,1)$, and the recovery ring (2nd layer ring) and backup rings (3rd layer ring) of server 00.

- The first layer ring is called *primary ring*, which is composed of all servers in the BCube network. The entire key space is divided into sub key spaces and each server on the primary ring is responsible for one sub key space. Fig. 3 shows an example of the primary ring of BCube$(4,1)$ depicted in Fig. 1.
- Each primary server on the primary ring, say server P, has a second layer ring called *recovery ring*. P's recovery ring is composed of all one-hop neighbors of P, and when P fails its data should be recovered to the RAM of the servers on its recovery ring. Fig. 3 shows an example of the recovery ring (01, 02, 03, 10, 20, 30) of a primary server 00.

- Each recovery server, say server R, corresponds to a third layer ring called *backup ring*. The backup ring is composed of the servers that are one-hop to the recovery server R and two-hop to the primary server P. The backup copies of the key-value pairs of P are stored in the disks of servers on the backup rings. Fig. 3 shows an example of six backup rings.

In Fig. 3, all the 16 primary servers have the same primary-recovery-backup structure (i.e., a directly connected tree) with server 00. Note that the order of the primary/recovery/backup servers on the rings can be decided by any deterministic methods, and here we simply arrange them alphabetically which is known by all servers. Clearly in the symmetric MultiRing structure, if a server A is a primary/recovery/backup server of another server B, then B is also a primary/recovery/backup server of A. For MultiRing, we obtain the following theorem.

Theorem 1 *Consider a RAMCube constructed based on BCube(n,k). There are n^{k+1}, $(n-1)(k+1)$, and $(n-1)k$ servers on the primary ring, recovery ring, and backup ring, respectively. And a primary server has totally $\frac{(n-1)^2k(k+1)}{2}$ backup servers.*

The formal proof of Theorem 1 is given in Appendix A. For BCube$(16,2)$, for example, there are 4096 primary servers, each of which has 45 recovery servers (each having a 30-server backup ring) and 675 backup servers. Note that it is not mandatory for a primary server to employ all its recovery/backup servers. For example, a primary server in BCube$(16,2)$ may employ 30 (instead of all the 45) recovery servers on its recovery ring to reduce fragmentation, at the cost of longer recovery delay and lower aggregate backup bandwidth.

Similarly, a slightly different design for backup rings is to include the recovery server itself in its own backup ring, in which the number of servers on a backup ring increases by $\frac{1}{(n-1)k}$. RAMCube does not adopt this asymmetric design because it adds the complexity of management while brings only insignificant improvement. For example, by including the recovery server in its own backup ring in BCube$(16,2)$ the number of servers on a backup ring increases from 30 to 31.

4 Failure Detection and Recovery

This section first introduces failure detection in RAMCube, and then discusses the recovery of single server failures, multiple server failures and switch failures.

4.1 Failure Detection

In order to quickly detect server/switch failures, RAMCube uses a lightweight heartbeat mechanism as a gauge of server liveness. A primary server periodically sends heartbeat messages to each of its recovery servers. The timeouts should be relatively short (e.g., a few hundred milliseconds) for fast failure detection.

In RAMCube for each primary server P there is a *local coordinator*, which is usually one of P's recovery servers. The ZooKeeper service [25] can be used to achieve high availability and durability of the coordinators. If a recovery server (say R) does not receive heartbeats from its primary server (say P) for a certain period, then R will report this suspicious server failure to P's local coordinator, which would verify the problem by immediately asking all P's recovery servers. This would test all the $k+1$ paths through the $k+1$ switches directly connected to P. If P does fail, then all the recovery servers should report the failure to the local coordinator and the coordinator will initiate the recovery. Otherwise if some recovery servers report P is still alive, then the coordinator will notify the available paths to the recovery servers that lose connections to P so that they can temporarily connect to P by using BCube source routing (BSR) [24]. If the primary server keeps being unreachable through a switch for a period of time (e.g., 10 seconds), then the switch will be considered failed.

For example, in the network depicted in Fig. 1, if server 01 suspects 00 fails since it cannot receive heartbeats from 00, it would report to the local coordinator (say, 02) and then all paths to 00 (through switches $\langle 0,0\rangle$ and $\langle 1,0\rangle$) will be tested. If all the tests fail the coordinator 02 would make a final decision to initiate recovery.

In case of false failure detection due to rare conditions, e.g., all $k+1$ switches directly connected to primary server P are *simultaneously* temporarily congested, for each recovery server R after the recovery starts, the backup servers connected to R would reject any further backup requests from P and indicate P to stop servicing the corresponding sub key space. By this means, false failure detection in RAMCube is NOT *fatal* but *expensive*. Our local detection eliminates much of the possibility of false positives induced by network problems and thus effectively reduces unnecessary recoveries.

4.2 Single Server Failure Recovery

RAMCube uses a *global coordinator* to assign the entire key space to the primary servers. For each primary server P, a *local coordinator* is used to assign P's sub key space to P's recovery servers, and to assign the sub sub space of each recovery server to the backup servers. Recovering a server failure includes restoring data and maintaining the MultiRing structure. Three cases have to be considered corresponding to the three roles of the failed server. For simplicity in this subsection we assume that a primary server employs all its recovery servers and each key-value pair has one backup copy.

Primary server failures. A primary server failure should be recovered as fast as possible since it greatly affects system availability and durability. After a primary server fails, the recovery servers would fetch backup copies from their directly connected backup servers.

In this process each backup server services at most *two* recovery servers because it has two digits different from the failed primary server. For example, in Fig. 3 if a primary server (say 00) fails, a backup server (11) services two recovery servers (01 and 10). Thus there is little traffic overlap or congestion either in the network or in the disks of backup servers. Therefore, given the normal configuration with 10 Gbps network bandwidth and 100 MB/sec disk I/O bandwidth, a RAMCube constructed based on BCube$(16,2)$ with 4096 servers can easily recover several tens of GB data in a few seconds.

After this process, these recovery servers become the new primary servers of their newly assigned sub key space. We refer to this urgent data transfer process as "backup-to-primary" process.

In order to maintain the MultiRing structure, the backup key-value pairs also need to be transferred to the new backup rings. For example, after a primary server (say 00) fails in Fig. 3, server 01 will become a new primary server, and 11, 21, 31 become the new recovery servers and transfer the backup data to their backup rings (11 to 10, 12, 13; 21 to 20, 22, 23; 31 to 30, 32, 33). Compared to the "backup-to-primary" process, this "backup-to-backup" process is not urgent and could be accomplished offline in a relatively longer period of time.

Recovery server failures. If a recovery server fails, for each affected primary server P, the affected sub key space would be reassigned to some other servers on P's recovery ring. In normal cases a backup server B directly connects to two recovery servers R_1 and R_2, then after one recovery server (say, R_1) fails B can simply register to R_2, i.e., tells the primary server P and P's local coordinator to redirect R_1's sub key space to R_2. This process can be accomplished instantaneously.

In Fig. 3, for example, if the recovery server (01) of a primary server (00) fails, RAMCube can register a new recovery server for each affected backup server (11, 21, 31): new recovery server 10 for backup server 11, 20 for 21, and 30 for 31.

If the backup server B has no more directly connected recovery servers for the primary server P (e.g., due to a previous failure), then P will designate a new recovery server R' for B's sub key space, and transfer B's backup data to a server B' on the backup ring of R'. Here B has a big chance to find a directly connected B', because B is two-hop away from P while R' is one-hop away from P, which means initially in BCube(n,k) there are $2(n-2)$ recovery servers of P that are two-hop away from B and each have a backup server one-hop away from B.

Continue with the previous example. After the recovery server (01) of a primary server (00) fails, the backup server B (11) finds a new recovery server (10). And if 10 also fails, 11 could find another recovery server R' (e.g., 02) and transfer backup data to a new server B' (12) that is on the backup ring of R' (02) and one-hop away from 11. This "backup-to-backup" data transfer process is not urgent and could be done offline.

Backup server failures. The recovery of backup server failures is straightforward and not critical: after the backup server B fails, each affected primary server P will transfer B's backup data to another healthy server on the same backup ring with B. The "primary-to-backup" data transfer process is not urgent and could be done offline.

Generally speaking, if initially each RAMCube server stores β GB data in RAM, then after a server failure, on average: (i) β GB data needs to be transferred one-hop as fast as possible (within a few seconds) and β GB data needs to be transferred one-hop in a relatively long period (e.g., several minutes) due to the primary server role; (ii) $0 \sim \beta$ GB data needs to be transferred one-hop (with high probability) or two-hop in a relatively long period due to the recovery server role; and (iii) β GB data needs to be transferred one-hop in a relatively long period due to the backup server role. The data transfer to backup servers is not urgent because (i) backup data has little affect on I/O performance, and (ii) even if another failure happens before the transfer is over it still can be recovered at acceptable price.

Maintaining the mappings. RAMCube uses two kinds of coordinators to maintain the mappings between the key space and the multi-layer logical rings.

First, there is a *global coordinator* that maintains the mapping between the entire key space and the primary ring. Any changes of the mapping should be notified to the global coordinator. The clients have a local cache of (part of) the mapping, and normally they can directly send requests to primary servers without querying the coordinator. If a client cannot locate a key either because the mapping has not been cached or because the cached information stales due to server failures, it will fetch up-to-date information from the global coordinator.

Second, for each primary server P there is a *local coordinator* that maintains the mappings between P's sub key space and P's recovery ring/backup rings. Similar to the global coordinator, any changes of the mappings should be notified to the local coordinator. As discussed in Section 4.1, the local coordinators are also responsible for aggregating failure reports from recovery servers and identifying switch failures.

Both kinds of coordinators can use the ZooKeeper service [25] to provide high availability and durability.

4.3 Handling Multiple Server Failures

RAMCube treats multiple server failures as separated single server failures. If the multiple failures take place one by one, i.e., one failure happens after the previous failure has been recovered, two factors limit the number of failures that can be tolerated while keeping the MultiRing structure. First, if a server is disconnected from RAMCube or all servers on a recovery/backup ring fail, then the MultiRing structure fails. Second, recovery servers must have enough spare RAM to accommodate the recovered data. We obtain the following theorem for this one-by-one failure pattern.

Theorem 2 *Consider a RAMCube constructed based on BCube(n,k). Suppose that each server installs α GB RAM and stores β GB data. Then in the worst case at least* $\min\{nk-k-1, \frac{(\alpha-\beta)(nk+n-k)}{\alpha}\}$ *one-by-one failures can be tolerated before the MultiRing structure fails.*

The formal proof of Theorem 2 is given in Appendix B. In BCube$(16,2)$, for example, if $\alpha = 64$ and $\beta = 48$ then in the worst case the MultiRing structure can tolerate at least 11 server failures; and if $\alpha = 64$ and $\beta = 56$ then at least 5 failures can be tolerated in the worst case. Note that here a *MultiRing structure failure* does NOT necessarily mean any data loss. This is because even if the data cannot be backuped/recovered in the backup/recovery ring after the MultiRing structure fails, we can still maintain durability by utilizing other servers in RAMCube for backup/recovery, in which case RAMCube performs similar to RAMCloud [28].

Another multiple failures pattern is simultaneous failures. We handle simultaneous failures in the same way as the one-by-one failure pattern, unless they are recovery/backup servers of each other. If two directly-connected neighbors fail at the same time, RAMCube will exclude them from each other's recovery ring; and if a primary server P and its backup server B (and vice versa) fail simultaneously, RAMCube will ask the local coordinators of P and B to locate the redundant backup copies as discussed below.

For durability each key-value pair has multiple backup copies distributed in the backup servers' disks. We assign one of the backup copies for each key-value pair as the *dominant* backup copy (dominant copy for short). Normally during recovery only the dominant copies are read from disks and transferred to the recovery servers. Non-dominant copies are requested only if the dominant copies are failed. Several factors should be considered for the placement of the backup copies. E.g., the copies should reside in different racks in case of rack failure, and different disk I/O bandwidths should also be considered for balancing the load. Currently RAMCube simply uses a straightforward mapping method: the key space held by a primary server is partitioned and each sub space is assigned to a recovery server, and the sub space of a recovery server is divided into b shares each being assigned to its f backup servers, where b and f are the number of backup servers and the replication factor (i.e., f backup copies on disks in addition to one primary copy in RAM), respectively. For simultaneous failures pattern, clearly in the worst case at least f failures can be tolerated.

4.4 Handling Switch Failures

RAMCube can easily handle switch failures by leveraging the multiple paths of the BCube network. For example, in the network depicted in Fig. 1, if servers 01, 02, 03 all find server 00 is unreachable through switch $\langle 0,0\rangle$ but they can receive ping acknowledgements through switch $\langle 1,0\rangle$, then RAMCube considers switch $\langle 0,0\rangle$ failed. Since a switch failure in BCube results in only graceful performance degradation [24] but no data loss or unavailability, it is not critical and we can replace the failed switch in a relatively longer period of time (compared to the primary server failure recovery).

Since there are $k+1$ paths between any pair of servers, RAMCube is very unlikely to have a network partition that may result in serious unavailability like the service disruption that Amazon EC2 suffered in April 2011 [1].

5 Other Design Issues

5.1 Servicing During Recovery

Since RAMCube requires the involved recovery/backup servers to be one-hop/two-hop neighbors of the primary server, the recovery is guaranteed to be restricted within a region at most two-hop away from the failed server. Therefore, despite the failure recovery all the servers and switches out of that region are free to service I/O requests. The ability to "servicing during recovery" improves the overall availability of the system. In contrast, in previous studies such as RAMCloud [28], since a failure recovery involves all the servers and switches, when a server fails, either the whole system has to stop storage services during failure recovery, or the recovery and the services would affect each other unpredictably both on servers and in network.

Servicing during failure recovery is very similar to servicing under normal circumstances, except that the primary server should not backup its data to the servers involved in the recovery. We illustrate the ability of RAMCube to servicing during recovery by utilizing BCube$(4,2)$ with 64 servers (because the diameter of

BCube$(4,1)$ is only 2). Suppose that a primary server, say server 000, fails. Then the 36 servers 001∼110, 120, 130, 200∼210, 220, 230, 300∼310, 320, 330 are recovery servers or backup servers of the failed server 000, and the remaining 27 servers can service read/write requests as usual, except that they should not employ their recovery/backup servers involved in the recovery. For example, primary server 111 can still service requests, but it cannot employ three servers 110, 101 and 011 as its recovery servers, and it also cannot use the backup rings corresponding to those three servers.

To describe the ability of RAMCube to servicing during recovery, we define the "availability ratio of storage servers during recovery" (*availability ratio* for short), as the ratio of the number of available storage servers that can still service requests without interfering with the recovery to the number of all storage servers. For availability ratio, we obtain the following theorem.

Theorem 3 *Consider a RAMCube constructed based on BCube(n,k). During the recovery of a single server failure, the availability ratio (denoted as γ) satisfies $\gamma = 1 - \frac{(k^2+k)n^2-2(k^2-1)n+(k^2-k)}{2n^{k+1}}$.*

The formal proof of Theorem 3 is given in Appendix C. When using relatively large n and k RAMCube has a high availability ratio, which improves overall system availability. For example, when $n = 16$ and $k = 2$ we have $\gamma \approx 0.824$. Clearly in multiple one-by-one server failures RAMCube with $k \geq 2$ still has a high availability ratio due to the high proportion of the more-than-two-hop-away neighbors. Section 6.6 will show through simulations that the availability ratio of RAMCube in simultaneous failures is also high. In contrast, the availability ratio of RAMCloud [28] is always 0.

5.2 Handling Heterogeneity

This subsection considers two kinds of heterogeneity in RAMCube.

Different number of recovery servers per backup server. As discussed in Section 4.2, a backup server may service 1 or 2 recovery servers during recovery. This is because initially in RAMCube a backup server has two recovery servers, but as more failures happen it may lose one recovery server due to previous failures. For example, in Fig. 3 if a recovery server (01) of a primary server (00) first fails, the backup servers (11, 21, 31) register to other recovery servers (respectively 10, 20, 30). If 00 also fails, then in the recovery of 00, backup servers 11, 21, 31 will service one recovery server, while other backup servers service two.

To fully utilize the network/disks bandwidth in recovery, RAMCube should try to finish the recovery processes of all backup-recovery server pairs at the same time, meaning all backup servers have similar recovery bandwidth despite the number of their recovery servers.

Consider a RAMCube constructed based on BCube(n,k) and we use BCube$(16,2)$ as an example. First suppose that a recovery server (001) of a primary server (000) fails. Then 001's $(n-1)k$ backup servers (011, 021, ...) would respectively register to the other recovery server (010, 020, ...). Then the primary server (000) fails. In this case the $(n-1)k$ backup servers (011, 021, ...) have one recovery server, while others have two. We call the backup servers with only one recovery server as "lonely backup servers". Now among the $(n-1)(k+1)-1$ recovery servers, $(n-1)k$ servers (010, 020, ...) have a lonely backup server and we call them "affected recovery servers", while the remaining $(n-1)(k+1)-1-(n-1)k = n-2$ servers have no lonely backup server and we call them "free recovery servers" (002, 003, ...). The backup servers on a backup ring can be divided into three types: (i) 1 lonely backup servers (e.g., 011), (ii) $n-2$ "free backup servers" (012, 013, ...) that connect to one free recovery servers and one affected recovery servers, and (iii) $(n-1)k-1-(n-2) = (n-1)(k-1)$ "affected backup servers" (110, 210, ...) that connect to two affected recovery servers.

Let α, β, γ be the optimal bandwidth shares assigned to the lonely/free/affected backup servers on the backup ring of the affected recovery server with per-port λ Gbps bandwidth, respectively. To finish all the recovery at the same time, we have $\alpha = 2\gamma$, $\alpha = \beta + \frac{\lambda}{(n-1)}$, and $\alpha + (n-2)\beta = \lambda$. Therefore we have $\alpha = \frac{2n-3}{(n-1)^2}\lambda$, $\beta = \frac{n-2}{(n-1)^2}\lambda$, and $\gamma = \frac{n-1.5}{(n-1)^2}\lambda$. When $n \gg 1$ we have $\alpha \approx 2\beta \approx 2\gamma$. Therefore, the lonely backup servers that have only one recovery server R should use approximately twice the bandwidth of R used by R's other backup servers. E.g., for BCube$(16,2)$ with $\lambda = 10$ we have $\alpha \approx \frac{4}{3}$ Gbps and $\beta \approx \gamma \approx \frac{2}{3}$ Gbps. Clearly for multiple failures as long as $n \gg 1$ and the failures happen randomly we can get similar results for α, β, γ. Therefore, as discussed in Orchestra [16], we can get nearly-optimal bandwidth allocation by simply creating two separate TCP connections for the backup servers connecting to only one recovery server.

Different disk bandwidths of backup servers. The backup servers may have different number of disks, different disk parameters, or even different kinds of storage media (e.g., flash memory). If RAMCube handles failure recovery despite these differences, the slowest backup server may degrade the recovery performance. This kind of heterogeneity can be handled in a way similar to RAMCloud [28]: The sub key spaces could be assigned to the backup servers according to their disk band-

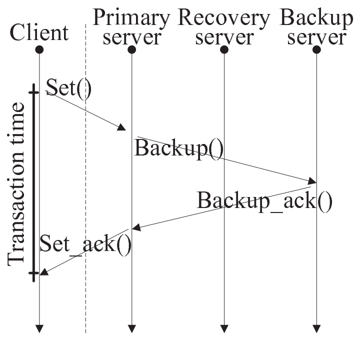

Figure 4: Data path in processing a write request in RAMCube.

widths, so that they can finish the recovery processes with similar delays.

5.3 Defragmentation and Cold Start

After a primary server fails RAMCube recovers its data to multiple recovery servers, on which the recovered fragmented data may lost locality. Although locality has no effects on our current simple key-value data model, this issue might become important if RAMCube supports richer data models in the future. A simple method for defragmentation is to restart or reinstall the failed primary server at the same place in the BCube network where it had been. With the coordinator's help, the revived primary server can easily restore its data from relevant servers offline.

A dangerous situation for RAMCube is that the entire system loses power at once. A simple way to addressing this problem is to install on each server a backup battery that extends the power long enough to flush all the primary data to disks. Then the cold start after power returns is simple: each server restores its primary data to RAM and reconstructs connections with its backup servers. If the battery cannot support flushing all the data, at least it needs to ensure the buffered backup data (not yet written to disks) to be flushed. When power returns the cold start is performed as many simultaneous defragmentations for all the servers (with the difference that data is fetched from backup disks). The primary servers should be reconstructed exactly the same as they had been before the power failure.

6 Implementation and Evaluation

6.1 Implementation Architecture

We have prototyped RAMCube by designing and implementing a service in Windows Server 2008 R2. The

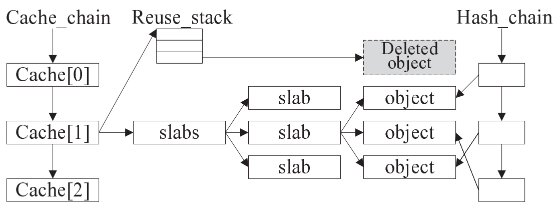

Figure 5: Slab-based memory management.

RAMCube service is multi-threaded and has the following components: the *connection manager* (containing 3K lines of C++ code) which maintains the status of directly connected neighbors and interacts with other servers, and the *memory manager* (having 2K lines of C++ code) which handles *set/get/delete* requests inside a server.

The *connection manager* uses libevent [2] to handle events and buffer data. Since in RAMCube all servers symmetrically act as primary/backup/recovery servers, the connection manager handles multiple network events including receiving data from clients and sending/receiving backup/recovery data. Besides, the connection manager is also responsible for sending/receiving heartbeat messages.

When the primary server receives a key-value pair, it stores the data in its RAM (handled by the *memory manager* discussed later) and sends a copy to the relevant backup server. The backup server acknowledges after the copy is written to the RAM (not disks), and then the primary server acknowledges to its client. The process is depicted in Fig 4. This approach provides efficiency both in normal operations and in recovery.

The *memory manager* uses a slab-based mechanism [3, 4], to alleviate fragmentation problems caused by frequent alloc/free operations in memory management inside a server. As shown in Fig. 5, it pre-allocates a large amount of memory during initialization, and uses fixed-size memory chunks with a series of predetermined size classes. The memory manager uses a hash table to quickly locate the keys, and implements a reusable stack to efficiently collect and reuse obsolete memory.

The memory manager uses a simple log-structured approach similar to previous logging file systems [30] for asynchronously writing backup data into the disks of backup servers. When a backup server receives a write request, it stores the key-value pair in its in-memory buffer. When the buffer fills, it writes the buffered data to its disk in one single large transfer and then frees the buffer. This is achieved by appending the new key-values to the log in its disk, which is divided into 8 MB segments.

We also implement a simple coordinator that can manage configuration information, maintain the mapping between the key space and the rings, aggregate failure reports from recovery servers, and identify switch failures. For simplicity in current prototype we only implement the *global coordinator* which takes charge of all the responsibilities of *local coordinators* discussed in Section 4. This should not become a bottleneck since our testbed is relatively small (introduced in the next subsection). Our prototype also has not implemented a mechanism for handling coordinator failures. We will incorporate the ZooKeeper service [25], which provides high availability and durability for coordinators, into RAMCube in our future work.

In order to minimize the recovery delay, RAMCube pipelines the recovery procedure [28], including reading logs from disks, transmitting logs to recovery servers, and replying logs. The order in which the segments are read is decided in advance so that the recovery server can reply the logs immediately after receiving them. Currently we have not implemented a log cleaner as well as a cold start mechanism (discussed in Section 5.3) in our prototype, which is orthogonal to the design of RAMCube and will be implemented in our future work.

6.2 Testbed

We have built a RAMCube testbed with 16 Dell R610 servers and 2 48-port Gigabit Ethernet switches (Pronto 3290 and Quanta LB4G) constructing a 1 Gbps BCube$(4,1)$ network. Each server has one Intel Xeon 2.27 GHz quad core CPU and 16 GB RAM, and installs one Seagate ST9500430SS 7200 RPM, 1 TB disk. In the future we would replace the 1 Gbps network with the 10 Gbps BCube by using the ServerSwitch 10 Gbps NIC (which is still under development and will be available soon).

In our experiments each key-value pair has only one backup copy, because our current testbed is relatively small (only three servers on a backup ring) and each server has only one disk installed. In practice a simple way to effectively supporting larger replication factors is to install multiple disks on each server.

We in turn study the performance of normal I/O operations, the recovery of server failures, the performance after a switch failure, and the availability ratio during multiple server failures. Each experiment is repeated at least 100 times.

6.3 Throughput

Our first experiment evaluates the write throughput of a RAMCube server, measured by the number of requests handled per second. Like the RAMCube service, we implement a multi-threaded benchmark. We run *m* service threads on a RAMCube server and *m* benchmark threads on a client machine. Each thread has a busy loop with 150 connections (to the RAMCube service) that asynchronously perform write operations with the form of "*set* key value". The size of the key-value pairs is 100 bytes and the key is a random 15-byte string. For simplicity we map the entire key space onto the tested server. For comparison, we also emulated memcached [3] (an in-memory key-value store on Linux) by discarding the backup operations of RAMCube and letting the primary server acknowledge to clients as soon as the data is written to its RAM. We measure the number of *set* requests handled per second as a function of the number of threads (*m*) running on the server.

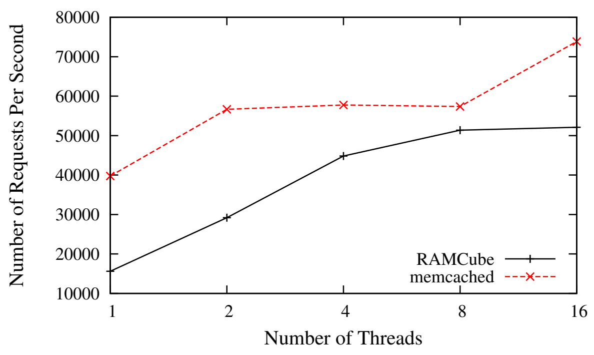

Figure 6: RAMCube server throughput.

The result (depicted in Fig. 6) shows that RAMCube and (emulated) memcached have comparable throughputs, both of which increase as the number of threads increases. The difference is because memcached does not need to send data to backup servers. The throughput of RAMCube increases little when there are more than 8 threads because the CPU is quad-core. Clearly, the write throughput is bounded by the CPU (while our experiments use a relatively slow CPU) and we can expect a higher throughput by running more threads on a RAMCube server with more cores and higher frequency.

6.4 Server Failure Recovery

One important ability of RAMCube is fast recovery from server failures. In our second experiment we first fill a primary server with 12 GB data (each key-value pair having 100 bytes), and then cause a failure of that server and measure the size of the aggregate transferred backup data over time. The heartbeat timeout is set to 200 ms.

The result is depicted in Fig. 7. RAMCube recovers 12 GB data in less than 17.5 seconds. The aggregate recovery bandwidth keeps almost always about 699.5 MB/sec during the entire recovery, which is very close to the optimal recovery bandwidth bounded by the NIC bandwidth (1 Gbps) and the number (6) of recovery servers. Note that in our experiments the bandwidth of the NIC is comparable to that of disks. We can expect a much faster

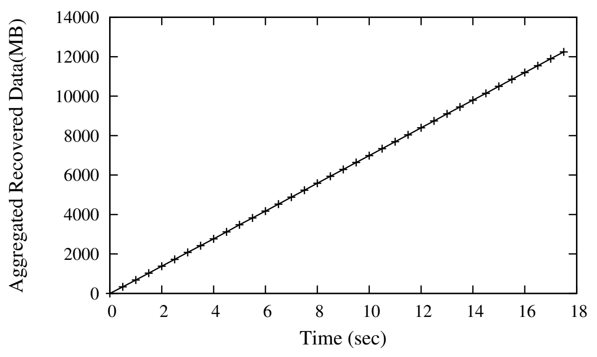

Figure 7: Single server failure recovery.

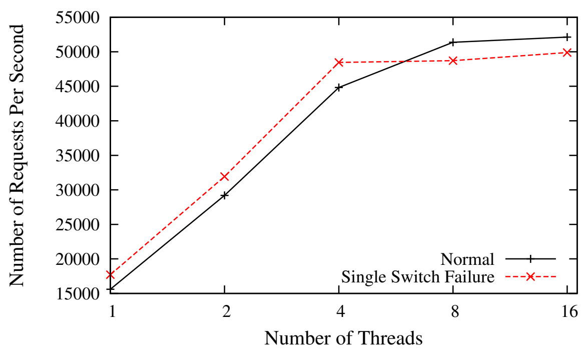

Figure 9: RAMCube throughput with a switch failure.

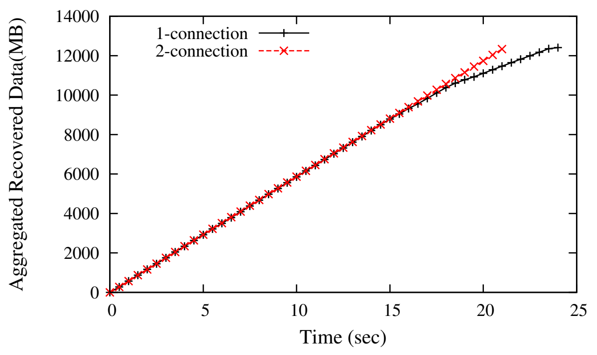

Figure 8: Multiple server failures recovery.

recovery speed by using ServerSwitch 10 Gbps NIC and more recovery/backup servers in a larger RAMCube with more levels (meaning more NIC ports per server).

RAMCube also supports multiple server failures recovery. Here we show how RAMCube recovers from a primary server failure right after a recovery server failure. In the network depicted in Fig. 1, we first cause a failure of the recovery server (01) of a primary server (00), which can be recovered instantaneously by registering the affected backup servers (11, 21, 31) to new recovery servers (respectively 10, 20, 30). We then make 00 also fail.

The recovery process is depicted in Fig. 8, where "1-connection" represents that each backup-recovery pair has exactly one TCP connection, while "2-connection" represents that if a backup server has only one recovery server R then it has two TCP connections to R. At the beginning both "1-connection" recovery and "2-connection" recovery have a high aggregate recovery bandwidth of about 587 MB/sec, which is bounded by the NIC bandwidth (1 Gbps) and the number (5) of recovery servers. After 16 seconds, however, only "2-connection" can keep the recovery speed while "1-connection" receives an obvious degradation. This is because in "1-connection" recovery after this point some backup servers finish recovery, resulting in a waste of the network bandwidth. This figure shows that our simple method (discussed in Section 5.2) for handling backup servers heterogeneity has a near-optimal result.

6.5 Switch Failures

Switch failures are a critical problem that prevents RAM-based storage systems from being practical. We turn off a switch and evaluate the write throughput of a RAMCube server, as a function of the number of threads (m).

The result is depicted in Fig. 9, where the throughput without switch failures is also included for comparison. The throughput of RAMCube with a switch failure is very similar to that without a failure. This is because RAMCube handles switch failures by using BCube source routing [24] which can remain all the connections between the primary server and its backup servers. In both cases it is the CPU, not the network, that bounds the throughput. Therefore, in theory a RAMCube server should have the same throughput with/without a switch failure, and the small variance depicted in Fig. 9 mainly comes from hardware differences between the switches. When the network is heavy-loaded, which is not shown in this figure, RAMCube may have a more obvious performance degradation with switch failures.

6.6 Availability Ratio

This subsection studies the *availability ratio* of RAMCube under simultaneous failures through simulations. Recall that availability ratio is the ratio of the number of available storage servers to the number of all servers (Section 5.1), which reflects the capability of servicing during server failure recovery that improves the overall availability of the storage system. We simulate two RAMCube topologies constructed based on BCube$(16, 2)$ (with 4K servers) and BCube$(16, 3)$ (with 64K servers). For each topology, we simulate various number of simultaneous failures randomly chose from all servers. The result is plotted in Fig. 10.

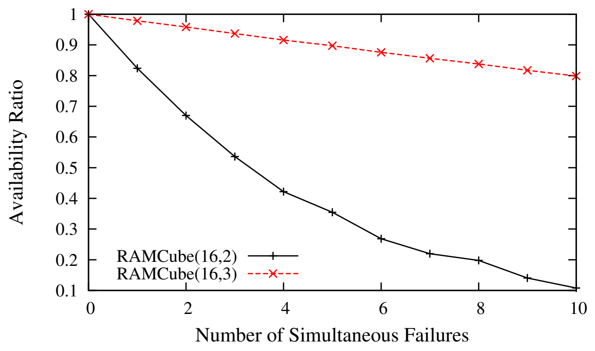

Figure 10: RAMCube availability ratio

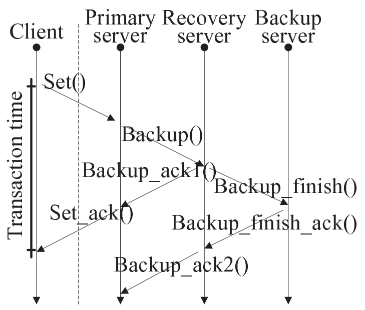

Figure 11: Alternative data path in processing a write request in RAMCube.

The result shows that when there is one failure both topologies provide high availability ratio, 0.824 in BCube$(16,2)$ and 0.978 in BCube$(16,3)$. As the number of failures increases, the ratios degrade gracefully. E.g., when there are 5 failures in BCube$(16,2)$ there are still 35.4% available servers. A potential impact not shown in Fig. 10 is that with the same number of servers a higher availability ratio usually means lower aggregate recovery bandwidth. However, this should not be a problem in a relatively large network. E.g., given the normal configuration with 10 Gbps network bandwidth and 100 MB/sec disk I/O bandwidth, in BCube$(16,2)$ with 4K servers the available aggregate network/disk bandwidths for recovery are respectively 112.5 GB/sec and 67.5 GB/sec, which are sufficient to recover several tens of GB data in 1$\sim$2 seconds even with the presence of multiple failures.

7 Discussion and Future Work

This section discusses several challenging issues that will be considered in the future RAMCube design.

First of all, since low latency [31] is one primary advantage of RAM-based storage over other storages, in the future RAMCube may require a low-latency Ethernet infrastructure of 10-μs level RTT, with the presence of various traffic patterns like shuffle and incast. This may be achieved by leveraging the programmability of ServerSwitch to realize in-network traffic control and congestion avoidance. We also need to support user-level applications to communicate directly with the NICs to send and receive data bypassing the kernel.

Second, as introduced in Section 6.1, the backup servers are two hops away from the primary servers and all backup data passes intermediate recovery servers. We can have an alternative data path depicted in Fig. 11. Here the different backup copies for a key-value pair are sent to different recovery servers, and the primary server returns to the client as soon as it receives acknowledgements from them. The recovery server acts as a *backup surrogate* and ensures backup data to be eventually written to the disk of one of its backup servers. This approach reduces the response time of write operations which may become non-trivial after we realize the aforementioned low-latency Ethernet, while of course introducing more complexity.

Third, our prototype only implements a simple data model of key-value pairs with variable lengths and *set*/*get*/*delete* operations. We plan to implement richer data model and more complex operations in RAMCube, such as column families [15], graphs, indexes, atomic updates to multiple key-value paris, conditional updates, and transactional operations [8].

Some practical issues orthogonal to the network design also need to be considered, for example, designing a segment cleaner for the logging system, implementing the ZooKeeper service [25] for handling coordinator failures, replacing replication with erasure coding for better durability, and incorporating flash memory into RAMCube.

8 Related Work

8.1 Large-Scale Cache

In recent years, large-scale web applications have been moving more and more data to RAM-based caches to meet their performance goals. E.g., memcached [3] is an in-memory key-value store for small chunks of data, widely used by a number of Web service providers to offload their storage servers. Google and Microsoft keep their entire Web search indexes in RAM [28].

Using RAM as a cache of disk-based storage systems has many limitations on consistency and efficiency. For example, it is the responsibility of applications to manage consistency between caches and storage, making it vulnerable to consistency problems. E.g., the software bug of the automated consistency verifying system made Facebook flush all its cache and stop all storage services [7]. The slowly refilling process made Facebook unreachable for approximately 2.5 hours.

8.2 RAM-Based Storage

Disk performance has not been improved as rapid as its capacity, making it increasingly difficult to meet the needs of more and more large-scale applications even with the help of cache [29]. This makes people consider new solutions of permanently storing data in RAM.

The idea of permanently storing data in RAM is not new. For example, main-memory databases [22, 26] keep entire databases in the RAM of one or more servers and support full RDBMS semantics. However, these systems cannot survive coordinated server failures and do not provide enough durability for large-scale systems.

Most recently, RAMCloud [28] is proposed as a RAM-based key-value store in data centers, where data is kept entirely in RAM. RAMCloud realizes low-latency RPC by using expensive InfiniBand networks and supporting user-level applications to send/receive data directly through the NICs bypassing the kernel. RAMCloud realizes fast server failure recovery by using aggressive data partitioning [15, 23]. They scatter backup data across hundreds or thousands of disks on backup servers. RAMCloud employs randomized techniques [12], mainly including random replica placement (with refinement) for load balance and random ping for server failure detection, to manage the system in a decentralized and scalable fashion.

Compared with RAMCloud, we improve RAM-based storage by addressing several critical DCN-related issues, including false failure detection due to temporary network problems, traffic congestion during failure recovery, and ToR switch failures.

8.3 Flash-Based Storage

Recently, flash memory is receiving increasing attention for high-performance storage [11, 19, 10, 18].

FAWN [11] couples low-power embedded CPUs to small amounts of flash storage, and balances computation and I/O capabilities to enable efficient, massively parallel access to data. FlashStore [18] uses flash memory as a non-volatile cache between RAM and disks. It organizes key-value pairs in a log-structure on flash to exploit faster sequential write performance and uses an in-memory hash table for indexing.

SkimpyStash [19] is a RAM space skimpy key-value store on flash-based storage, which uses a hash table directory in RAM to index key-value pairs stored on flash. HashCache [10] is also targeted at DRAM and flash combined storage system, leveraging an efficient flash-oriented data-structure to lower the amortized cost of insertions and updates.

As discussed in FAWN [11], for high query rates and smaller sized storage RAM is the most efficient; for low query rates and large sized storage disk is the most efficient; and flash memory lies in the middle ground. From a research standpoint, since the cost/bit of RAM improves rapidly, we want to be more aggressive and prefer RAM to flash memory as the primary storage to achieve higher throughput and lower latency. In our future work we will study replacing backup disks with flash memory, which may improve I/O performance (e.g., in a cold start) and adapt to emerging 40 Gbps Ethernet.

9 Conclusion

We have presented the design of RAMCube as a novel RAM-based key-value store for high-performance I/O in data centers. By exploiting the proximity of BCube network, RAMCube restricts all failure detection and recovery traffic within one-hop neighborhood. This eliminates much of the possibility of false failure detection and recovery traffic congestion. RAMCube uses multiple paths of BCube to handle switch failures. Our prototype implementation and evaluation show that RAMCube is promising to meet the requirements of RAM-based storage in high performance, availability, durability and scalability.

Acknowledgement

This work was supported in part by the National Basic Research Program of China (973) under Grant No. 2011CB302601, the National Natural Science Foundation of China (NSFC) under Grant No. 60903205, 61222205 and 61003076, and the Doctoral Program of Higher Education under Grant No. 20094307110008. We thank Yuxing Peng, Yijie Wang, Zhenbang Chen, Huiba Li, Dongsheng Li, the anonymous NSDI reviewers, and the iVCE Group for their insightful comments. Some preliminary results of this work were presented at the HotCloud'12 workshop [33].

References

[1] `http://aws.amazon.com/message/65648/`.

[2] `http://libevent.org/`.

[3] `http://memcached.org/`.

[4] `http://redis.io/`.

[5] `http://www.aristanetworks.com/en/products/7100series`.

[6] `http://www.broadcom.com/press/release.php?id=s634491`.

[7] `http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919/`.

[8] Aguilera, M. K., Merchant, A., Shah, M. A., Veitch, A. C., and Karamanolis, C. T. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst. 27*, 3 (2009).

[9] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *NSDI* (2010), pp. 281–296.

[10] Anand, A., Muthukrishnan, C., Kappes, S., Akella, A., and Nath, S. Cheap and large cams for high performance data-intensive networked systems. In *NSDI* (2010), USENIX Association, pp. 433–448.

[11] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. Fawn: a fast array of wimpy nodes. In *SOSP* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 1–14.

[12] Azar, Y., Broder, A. Z., Karlin, A. R., and Upfal, E. Balanced allocations. In *STOC* (1994), pp. 593–602.

[13] Bhuyan, L. N., and Agrawal, D. P. Generalized hypercube and hyperbus structures for a computer network. *IEEE Trans. Computers 33*, 4 (1984), 323–333.

[14] Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. The primary-backup approach, 1993.

[15] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A distributed storage system for structured data. In *OSDI* (2006), pp. 205–218.

[16] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. Managing data transfers in computer clusters with orchestra. In *SIGCOMM* (2011), pp. 98–109.

[17] Daniels, D., Doo, L. B., Downing, A., Elsbernd, C., Hallmark, G., Jain, S., Jenkins, B., Lim, P., Smith, G., Souder, B., and Stamos, J. Oracle's symmetric replication technology and implications for application design. In *SIGMOD* (1994), ACM Press, p. 467.

[18] Debnath, B. K., Sengupta, S., and Li, J. Flashstore: High throughput persistent key-value store. *PVLDB 3*, 2 (2010), 1414–1425.

[19] Debnath, B. K., Sengupta, S., and Li, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *SIGMOD Conference* (2011), T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds., ACM, pp. 25–36.

[20] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *SOSP* (2007), pp. 205–220.

[21] Fox, A. Toward recovery-oriented computing. In *VLDB* (2002), pp. 873–876.

[22] Garcia-Molina, H., and Salem, K. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng. 4*, 6 (1992), 509–516.

[23] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[24] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. Bcube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM* (2009), pp. 63–74.

[25] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC* (2010), pp. 1–14.

[26] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S. B., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB 1*, 2 (2008), 1496–1499.

[27] Lu, G., Guo, C., Li, Y., Zhou, Z., Yuan, T., Wu, H., Xiong, Y., Gao, R., and Zhang, Y. Serverswitch: A programmable and high performance platform for data center networks. In *NSDI* (2011).

[28] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J. K., and Rosenblum, M. Fast crash recovery in ramcloud. In *SOSP* (2011), pp. 29–41.

[29] Ousterhout, J. K., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G. M., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for ramclouds: scalable high-performance storage entirely in dram. *Operating Systems Review 43*, 4 (2009), 92–105.

[30] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. In *SOSP* (1991), pp. 1–15.

[31] Rumble, S. M., Ongaro, D., Stutsman, R., Rosenblum, M., and Ousterhout, J. K. It's time for low latency. In *HotOS* (2011).

[32] Wu, H., Lu, G., Li, D., Guo, C., and Zhang, Y. Mdcube: a high performance network structure for modular data center interconnection. In *CoNEXT* (2009), J. Liebeherr, G. Ventre, E. W. Biersack, and S. Keshav, Eds., ACM, pp. 25–36.

[33] Zhang, Y., Guo, C., Chu, R., Lu, G., Xiong, Y., and Wu, H. Ramcube: Exploiting network proximity for ram-based key-value store. In *HotCloud* (2012).

[34] Zhang, Y., and Liu, L. Distributed line graphs: A universal technique for designing dhts based on arbitrary regular graphs. *IEEE Trans. Knowl. Data Eng. 24*, 9 (2012), 1556–1569.

Appendix

A. Proof of Theorem 1

BCube(n,k) is equal to an n-tuple, $k+1$ dimensional generalized hypercube. Therefore there are n^{k+1} servers on the primary ring. For each primary server, there are $k+1$ directly connected switches, each with $n-1$ directly connected recovery servers. So there are $(n-1)(k+1)$ servers on the recovery ring. For each recovery server, there are k directly connected switches (except the one it uses to connect to its primary server), each with $n-1$ directly connected backup servers. Therefore, there are $(n-1)k$ servers on the backup ring.

The backup servers is two hops away from their primary server, and thus they have exactly two digits different from their primary server. Thus each backup server services 2 recovery servers irrespective of n and k. Therefore each primary server has totally $(n-1)(k+1)\times(n-1)k/2 = \frac{(n-1)^2k(k+1)}{2}$ backup servers.

B. Proof of Theorem 2

The number of one-by-one failures that can be tolerated is the smaller one of the following two cases.

First, if a server is disconnected from RAMCube or all servers on a recovery/backup ring fails, then the

RAMCube structure fails. From Theorem 1, there are $(n-1)(k+1)$ and $(n-1)k$ servers on the recovery ring and backup ring, respectively. Therefore, in the worst case $(n-1)k-1 = nk-k-1$ server failures (on the same backup ring) can be tolerated.

Second, recovery servers must have enough spare RAM to accommodate the recovered data. Let $m = (n-1)(k+1)$. After the first server fails, RAMCube must satisfy $\beta + \frac{\beta}{m} = \beta(1+\frac{1}{m}) \leq \alpha$; after the second server fails, which in the worst case may be a recovery server of the first failed server, RAMCube should satisfy $\beta + \frac{\beta}{m} + \frac{1}{m-1}(\beta + \frac{\beta}{m}) < \beta(1+\frac{1}{m-1})^2 \approx \beta(1+\frac{2}{m-1}) \leq \alpha$; $\ldots$; and after the r^{th} failure ($m-r >> 1$), in the worst case RAMCube should satisfy $\beta(1+\frac{r}{m-r+1}) \leq \alpha$, or $r \leq \frac{(m+1)(\alpha-\beta)}{\alpha} = \frac{(\alpha-\beta)(nk+n-k)}{\alpha}$.

C. Proof of Theorem 3

By Theorem 1, for each primary server in a RAMCube based on BCube(n,k), there are $(n-1)(k+1)$ recovery servers and $\frac{(n-1)^2k(k+1)}{2}$ backup servers, and the number of all storage servers is n^{k+1}. Therefore, the availability ratio equals to the number of servers more than two hops away from the failed server divided by the number of all servers, i.e., $\gamma = \frac{n^{k+1}-1-(n-1)(k+1)-(n-1)^2k(k+1)/2}{n^{k+1}} = 1 - \frac{(k^2+k)n^2-2(k^2-1)n+(k^2-k)}{2n^{k+1}}$.