CubicRing: Enabling One-Hop Failure Detection and Recovery for Distributed In-Memory Storage Systems

Yiming Zhang, *National University of Defense Technology;* Chuanxiong Guo, *Microsoft;* Dongsheng Li and Rui Chu, *National University of Defense Technology;* Haitao Wu, *Microsoft;* Yongqiang Xiong, *Microsoft Research*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/zhang

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

CubicRing: Enabling One-Hop Failure Detection and Recovery for Distributed In-Memory Storage Systems

Yiming Zhang**, Chuanxiong Guo*†*, Dongsheng Li***, Rui Chu***, Haitao Wu*†*, Yongqiang Xiong*‡
**National University of Defense Technology,* †*Microsoft,* ‡*MSR*
{ymzhang, dsli, rchu}@nudt.edu.cn, {chguo, hwu, yqx}@microsoft.com

Abstract

In-memory storage has the benefits of low I/O latency and high I/O throughput. Fast failure recovery is crucial for large-scale in-memory storage systems, bringing network-related challenges including false detection due to transient network problems, traffic congestion during the recovery, and top-of-rack switch failures. This paper presents CubicRing, a distributed structure for cube-based networks which exploits network proximity to restrict failure detection and recovery within the smallest possible one-hop range. We leverage the CubicRing structure to address the aforementioned challenges and design a network-aware in-memory key-value store called MemCube. In a 64-node 10GbE testbed, MemCube recovers 48 GB of data for a single server failure in 3.1 seconds. The 14 recovery servers achieve 123.9 Gb/sec aggregate recovery throughput, which is 88.5% of the ideal aggregate bandwidth.

1 Introduction

Disk-based storage is becoming increasingly problematic in meeting the needs of large-scale cloud applications in terms of I/O latency, bandwidth and throughput. As a result, in recent years we see a trend of migrating data from disks to random access memory (RAM) in storage systems. In-memory storage is proposed to keep data entirely and permanently in the RAM of storage servers. E.g., Tenant's CMEM [2, 4] builds a storage cluster with thousands of servers to provide public in-memory key-value store service, which uses one *synchronous* backup server for each of the RAM storage servers. Thousands of latency-sensitive applications (e.g., online games) have stored several tens of TB of data in CMEM. Ousterhout et al. propose RAMCloud [45], an in-memory key-value store that keeps one copy of data in storage servers' RAM and stores redundant backup copies on backup servers' disks. RAMCloud uses InfiniBand networks to achieve low latency RPC.

In-memory storage has many advantages over disk-based storage including high I/O throughput, high bandwidth, and low latency. For instance, CMEM provides $1000\times$ greater throughput than disk-based systems [4]; RAMCloud boosts the performance of online data-intensive applications [46] which make a large number of sequential I/O requests in limited response time (e.g., generating dynamic HTML pages in Facebook [49]); and the applications need no longer maintain the consistency between the RAM and a separate backing store.

Fast failure recovery is crucial for large-scale in-memory storage systems to achieve high durability and availability. Previous studies [46] show for normal cases ($3\times$ replication, 2 failures/year/server with a Poisson distribution) in a 10,000-server in-memory storage system, the probability of data loss in 1 year is about 10^{-6} if the recovery is finished in 1 second; and it increases to 10^{-4} when the recovery time is 10 seconds. On the other hand, the relatively high failure rate of commodity servers requires a recovery time of no more than a few seconds to achieve continuous availability [46] in large-scale systems. According to [7], 1000+ server failures occur in one year of Google's 1800-server clusters. Since the recovery of in-memory storage server failures requires to fully utilize the resources of the cluster [45], a recovery time of a few seconds would result in an availability of about four nines (99.99%, 3,150 seconds downtime/year) if only server failures are considered, while a recovery time of several tens of seconds may degrade the availability to less than three nines, which could become the dominant factor for the overall availability.

RAMCloud realizes fast failure recovery by randomly scattering backup data on many backup servers' disks and reconstructing lost data in parallel through high-bandwidth InfiniBand networks. However, many realistic network-related challenges remain to be addressed for large-scale in-memory storage systems: (i) it is difficult to quickly distinguish transient network problems from server failures across a large-scale network; (ii) the large number (up to tens of thousands) of parallel recovery flows is likely to bring continuous traffic congestion which may result in a long recovery time; and (iii) top-of-rack (ToR) switch failures make fast failure recovery even more challenging.

To address these challenges this paper presents *CubicRing*, a distributed structure for cube-networks-based in-memory storage systems. CubicRing exploits network proximity to restrict failure detection and recovery within the smallest possible (i.e., one-hop) range, and cube networks could naturally handle switch failures with their multiple paths. We leverage the CubicRing structure to design a network-aware in-memory key-value store called *MemCube*, where the storage system and the network collaborate to achieve fast failure recovery. We implement a prototype of MemCube on BCube [35], and build a 64-node 10GbE testbed for MemCube evaluation. MemCube recovers 48 GB of data from a failed server in 3.1 seconds. In the recovery, the 14 recovery servers achieve 123.9 Gb/sec aggregate recovery throughput, which is 88.5% of the ideal aggregate bandwidth.

2 Preliminaries

Large-scale in-memory storage systems must provide a high level of durability and availability. One possible approach is to replicate all the data to the RAM of backup servers [4]. However, this approach would dramatically increase both the cost and the energy usage, and in-memory replicas are still vulnerable under power failures. On the other hand, although erasure coding can reduce some of the cost, it makes recovery considerably more expensive [46]. RAMCloud leverages high-bandwidth InfiniBand networks and utilizes aggressive data partitioning [19] for fast failure recovery [45]. It *randomly* scatters backup data across many backup servers' disks, and after a failure happens it quickly reconstructs lost data in parallel. However, it is difficult for RAMCloud's approach to scale to large clusters with thousands of servers because many network problems remain to be addressed [3]. We characterize the common network-related challenges for fast failure recovery in large-scale in-memory storage systems as follows.

False failure detection. To quickly recover from a failure, the timeout of heartbeats should be relatively short. However, various transient network problems [31] like incast and temporary hot spot 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 (as discussed in Section 5), the recovery of tens of GB of data is definitely very *expensive*. Since network problems cannot be completely avoided in any large-scale systems, our solution is to shorten the paths that heartbeats have to traverse, reducing the chances of encountering transient network problems. Ideally, if the servers only inspect the status of directly-connected neighbors, then we can minimize the possibility of false positives induced by transient network problems.

Recovery traffic congestion. Fast failure recovery requires an aggregate recovery bandwidth of at least 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. If the distributed recovery takes place in a random and unarranged manner and the recovery flows traverse long paths, it may bring hot spots in the network and result in unexpected long recovery time. Even on full bisection bandwidth networks like FatTree [34], severe congestion is still inevitable due to the problem of ECMP (equal-cost multi-path) routing [10]: large, long-lived recovery flows may collide on their hash and end up on the same output ports creating in-network bottlenecks. To address this problem, our solution is to restrict the recovery traffic within the smallest possible range. Ideally, if all the recovery flows are one-hop, then we can eliminate the possibility of in-network congestion.

ToR switch failures. A rack usually has tens of servers connected to a ToR switch. In previous work [4, 45] when a ToR switch fails, all its servers are considered failed and several TB of data may need to be recovered. The recovery storm takes much more time than a single recovery. Since the servers connected to a failed switch are actually "alive", our solution is to build the in-memory storage system on a multi-homed cubic topology, each server being connected to multiple switches. When one switch fails, the servers can use other paths to remain connected and thus no urgent recovery is needed.

3 Structure

3.1 Design Choices

Large-scale in-memory storage systems aggregate the RAM of a large number of servers (each with at least several tens of GB of RAM) into a single storage. This subsection discusses our choices of failure model, hardware, data model, and structure.

Failure model. For storage systems, (i) servers and switches may crash, which results in data loss (omission failures) [52]; and (ii) servers and switches may experience memory/disk corruption, software bugs, etc, modifying data and sending corrupted messages to other servers (commission failures) [43]. Like RAMCloud, in this paper we mainly focus on *omission* failures. Commission failures can be detected and handled using existing techniques like Merkle-tree based, end-to-end verification and replication [43, 52], but this falls beyond the scope of this paper and is orthogonal to our design.

Network hardware. The design of CubicRing is independent to network hardware and can be applied to both Ethernet and InfiniBand. For implementation, we follow the technical trend and focus on Ethernet, because most data centers are constructed using commodity Ethernet switches and high-performance Ethernet is more promising and cost-effective [34]. Recent advances show that Ethernet switches with 100 Gbps bandwidth [6] and sub-μs latency [5] are practical in the near future, and Ethernet NICs with RDMA support have reduced much of the latency of complex protocol stacks [27].

Data model. We focus on a simple key-value store that supports arbitrary number of key-value pairs, which consist of a 64-bit key, a variable-length value, and a 64-bit version number. Our prototype provides a simple set of operations ("*set* key value", "*get* key" and "*delete* key") for writing/updating, reading and deleting data.

Primary-recovery-backup. Storage systems have multiple copies for each piece of data. There are two choices, namely *symmetric replication* [22] and *primary-backup* [16], to maintain the durability and consistency. In symmetric replication all copies have to be kept in the RAM of servers and a quorum-like technique [25] is used for conflict resolution. In contrast, in primary-backup only one primary copy is needed to be stored in RAM while redundant backup copies could be stored on disks, and all read/write operations are through the primary copy. Considering the relatively high cost and energy usage per bit of RAM, we prefer *primary-backup*.

We refer to the servers storing the primary copies in RAM as **primary servers**, and the servers storing the backup copies on disks as **backup servers**. Once a primary server fails, the backup servers will recover the backup copies to some healthy servers that are called **recovery servers**. As discussed in Section 5, the number of recovery servers is a tradeoff between recovery time and recovered data locality: a larger number decreases the recovery time but results in higher fragmentation, and vice versa. The "primary-recovery-backup" structure (shown in Fig. 1(a)) is adopted by many storage systems like RAMCloud and BigTable [19], where each server symmetrically acts as all the three roles.

3.2 CubicRing

Our basic idea is to restrict failure detection and recovery traffic within the *smallest* possible (i.e., 1-hop) range. We improve the primary-recovery-backup structure (shown in Fig. 1(a)) with a directly-connected tree (shown in Fig. 1(b)), where a primary server has multiple directly-connected recovery servers, each of which has multiple directly-connected backup servers. Here two servers are "directly-connected" if they are connected to the same switch. Clearly, Fig. 1(b) can be viewed as a special case of Fig. 1(a).

In Fig. 1(b), the primary server P periodically sends heartbeats to its recovery servers, and once the recovery servers find P failed, they will recover the lost data from their backup servers. Since the recovery servers directly connect to the primary server, they can eliminate much of the possibility of false detection due to transient network problems (as discussed in Section 4.1); and since they also directly connect to their backup servers, the recovery traffic is guaranteed to have no in-network congestion.

The directly-connected tree provides great benefit for accurate failure detection and fast recovery. We *symmetrically* map the tree onto the entire network, i.e., each server equally plays all the three roles of primary/recovery/backup server. Our insight is that all cubic topologies are some variations of generalized hypercube (GHC) [15], each vertex of which can be viewed as the root of a tree shown in Fig. 1(b).

We take BCube [35] as an example. BCube$(n,0)$ is simply n servers connected to an n-port switch. 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, and has $N = n^{k+1}$ servers $a_k a_{k-1} \cdots a_0$ where $a_i \in [0,n-1], i \in [0,k]$. Fig. 2 shows a BCube$(4,1)$ constructed from 4 BCube$(4,0)$. If we replace each switch and its n links of BCube(n,k) with an $n \times (n-1)$ full mesh that directly connects the servers, we will get a $(k+1)$-dimension, n-tuple generalized hypercube.

We design the multi-layer cubic rings (*CubicRing*) as shown in Fig. 3 to map the key space onto a cube-based network (e.g., BCube), following the primary-recovery-backup structure depicted in Fig. 1.

- The first layer is the *primary ring*, which is composed of all the servers. The entire key space is divided and assigned to the servers on the primary ring. Fig. 3 shows an example of the primary ring.
- Each *primary server* on the primary ring, say server P, has a second layer ring called *recovery ring* that is composed of all its 1-hop neighbors (*recovery servers*). When P fails its data will be recovered to the RAM of its recovery servers. Fig. 3 shows an example of the recovery ring (01, 02, 03, 10, 20, 30) of a primary server 00.
- Each recovery server R corresponds to a third layer ring called *backup ring*, which is composed of the *backup servers* that are 1-hop to R and 2-hop to P. The backup copies of P's data are stored on the disks of backup servers. Fig. 3 shows an example of the (six) backup rings of a primary server 00.

In the *symmetric* CubicRing depicted in Fig. 3, all the 16 primary servers have the same primary-recovery-backup structure (i.e., a directly-connected tree) with server 00. We can easily obtain the following Theorem 1, the formal proof of which is given in Appendix A.

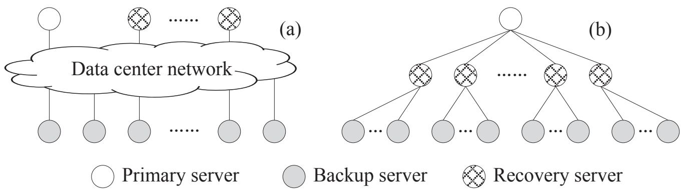

Figure 1: Primary-recovery-backup.

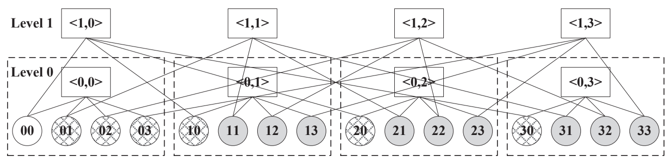

Figure 2: An example of BCube$(4,1)$.

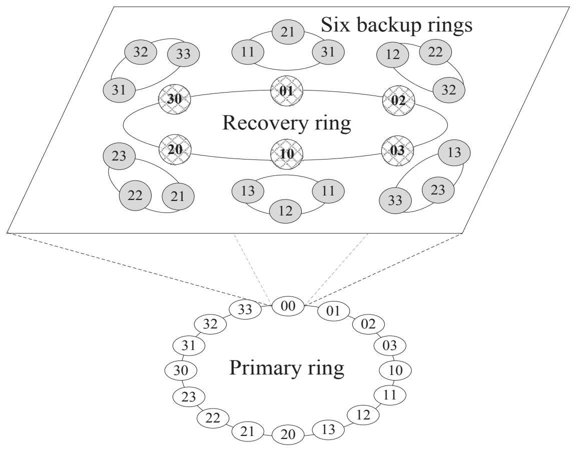

Figure 3: The CubicRing structure.

Theorem 1 *On BCube(n,k) there are n^{k+1}, $(n-1)(k+1)$, and $(n-1)k$ servers on the primary ring, recovery ring (of each primary server), and backup ring (of each recovery server), respectively. A backup server resides on two backup rings of a primary server, which has totally $\frac{(n-1)^2k(k+1)}{2}$ backup servers.*

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 CubicRing does not require a primary server to employ all its recovery/backup servers. E.g., a primary server in BCube$(16,2)$ may employ 30 (instead of all the 45) servers on its recovery ring to reduce fragmentation, at the cost of lower aggregate recovery bandwidth.

The construction of CubicRing is applicable to all cubic topologies such as MDCube [53], *k*-ary *n*-cube [55], and hyperbanyan [29], because they are all variations of the GHC [15] topology which consists of *r*-dimensions with m_i nodes in the i^{th} dimension. A server in a particular axis is connected to all other servers in the same axis, and thus CubicRing can be naturally constructed: all the servers in a GHC form the primary ring, and for each primary server its 1-hop neighbors form its recovery ring and 2-hop neighbors form backup rings. Next, we will focus on BCube [35] to design a network-aware in-memory key-value store called *MemCube*; extending for arbitrary GHC is straightforward.

3.3 Mapping Keys to Rings

MemCube uses a *global* coordinator for managing the mapping between the key space and the primary ring. The coordinator assigns the key space to the primary servers with consideration of load balance and locality: all the primary servers should store roughly-equal size of primary copies of data (called *primary data*), and adjacent keys are preferred to be stored in one server. Currently MemCube simplifies the load balancing problem by equally dividing the key space into consecutive sub spaces, each being assigned to one primary server. This design is flexible in *dynamically* reassigning the mapping when the load distribution changes (which has not yet been implemented in our current prototype).

The key space held by a primary server *P* is further mapped to *P*'s recovery servers; and for each recovery server *R*, its sub space is mapped to its backup servers. The mapping should make all the recovery servers be assigned equal size of data, because after *P* fails they will recover *P*'s data from their backup rings simultaneously. In order to avoid potential performance bottleneck at the global coordinator, the mapping from *P*'s key space to *P*'s recovery/backup rings is maintained by *P* itself instead of the coordinator, and the recovery servers have a cache of the mapping they are involved in. After *P* fails the global coordinator asks all the recovery servers of *P* to reconstruct the mapping.

Dominant/non-dominant backup data. For durability, each primary copy has *f* backup copies, among which there are 1 *dominant* copy stored on the backup server (according to the primary-recovery-backup mapping), and $f-1$ *non-dominant* copies stored on $f-1$ secondary backup servers (*secondary servers* for short) in different *failure domains* [44]. A failure domain is a set of servers that are likely to experience a correlated failure, e.g., servers in a rack sharing a single power source. The mapping from a primary server *P*'s key space to the secondary servers is also maintained by *P* and cached at *P*'s recovery servers. Normally only the dominant backup copy participates in the recovery. Non-dominant copies are used only if the primary copy and the dominant backup copy fail concurrently. In Fig. 2, e.g., suppose that the servers connected to the same level-0 switches are in one rack failure domain. Given primary server *P* and backup server *B* of a primary copy, any $f-1$ servers that reside in $f-1$ racks different from where *P* and *B* reside can serve as secondary servers. E.g., for primary server 00 and backup server 11, the secondary servers ($f = 3$) may be 21 in rack $< 0,2 >$ and 31 in $< 0,3 >$.

Normal read/write operations. When a primary server receives a read request it directly returns. When receiving a write, it stores the new data in its RAM, and transfers it to one (2-hop-away) backup server and $f-1$ secondary backup servers. When a backup server receives the data, it returns after storing it in its buffer to minimize write latency. The primary server returns after all the backup servers return. When the buffer fills, the backup servers use logging [48] to write buffered data to disks and free the buffer. The backup data's log is divided into *tracts* which are the unit of buffering and I/O.

4 Recovery

4.1 Failure Detection

Primary servers periodically send heartbeats to their recovery servers. If a recovery server R does not receive any heartbeats from its primary server P for a certain period, R will report this suspicious server failure to the coordinator, which would verify the problem by contacting P through all the $k+1$ paths on BCube(n,k). If P does fail, then all the tests should report the failure and the coordinator will initiate the recovery. Otherwise if some tests report P is still alive, then the coordinator will notify the available paths to the recovery servers that lose connections to P. If some (alive) servers keep being unreachable through a switch for a period of time, then the switch will be considered failed.

There are a high rate of transient network problems and a large number of small packets may be lost [31], which might result in large-scale unavailability and consequently severe disruptions. MemCube uses a relatively short timeout to achieve fast recovery. This introduces a risk that transient network problems make heartbeats get lost and thus may be incorrectly treated as server failures. Our solution is to involve as few as possible network devices/links on the paths that heartbeats traverse: MemCube achieves 1-hop failure detection which eliminates much of the possibility of network-induced false positives. In contrast, multi-hop detection (where, e.g., a heartbeat traverses $01 \rightarrow 11 \rightarrow 10 \rightarrow 00$ instead of $01 \rightarrow 00$ in Fig. 2) will considerably increase the risk.

In some uncommon cases, however, false positives are inevitable, e.g., a server is too busy to send heartbeats. MemCube uses *atomic recovery* to address this kind of problems, which will be discussed in Section 5.

4.2 Single Server Failure Recovery

A server failure means three types of failures corresponding to its three roles of primary/recovery/backup server. We sketch the major steps of recovering these failures in Pseudocode 1 (where for brevity no failure domain constraint is considered). During the recovery the backup data is read from disks of backup servers, divided into separate groups, transferred through the network, and received and processed (e.g., inserted into a hash table) by the new servers. Since most recovery flows are 1-hop, the in-network transfer is no long a bottleneck. And due to the relatively small number of recovery servers compared to other resources (as shown in our evaluation in Section 6), the recovery **bottleneck** is the *inbound* network bandwidth of recovery servers.

Pseudocode 1 Single server failure recovery

```
1: procedure RecoverFailures(FailedServer F)
2:     Pause relevant services
3:     Reconstruct key space mapping of F
4:     Recover primary data for primary server failure*
       ▷ All recoveries with * are performed concurrently
5:     Recover backup data for primary server failure*
6:     Resume relevant services
7:     Recover from recovery server failure*
8:     Recover from backup server failure
9: end procedure
```

Pause relevant services. After a server F's failure is confirmed by the coordinator, the key space held by F (as a primary server) will become unavailable. During the recovery all the relevant recovery/backup servers would pause servicing normal requests to avoid contention.

Reconstruct mapping. The coordinator asks all F's recovery servers to report their local cache of (part of) the mapping from F's key space to F's recovery/backup rings, and reconstructs an integrated view of the mapping previously maintained by F. Then the coordinator uses the mapping to initiate the recovery of primary/recovery/backup server failures for F.

Primary data recovery of *primary server failure*. After being notified the failure of a primary server F (say 00 in Fig. 2), F's backup servers (e.g., 11) will read backup data in tracts from disks, divide the data into separate groups for their 1-hop-away recovery servers (01 and 10), and transfer the groups of data to the recovery servers in parallel. To pipeline the processes of data transfer and storage reconstruction, as soon as the new primary server receives the first tract it begins to incorporate the new data into its in-memory storage. Keys are inserted to the hash table that maps from a key to the position where the KV resides. The new primary servers use version numbers to decide whether a key-value should be incorporated: only the highest version is retained and any older versions are discarded.

Backup data recovery of *primary server failure*. In addition to the primary data recovery, the (dominant) backup data previously stored on the old backup rings of the failed primary server F (00 in Fig. 2) needs to be recovered to the backup rings of the new primary servers R (i.e., F's recovery servers), for maintaining the CubicRing structure. To minimize the recovery time of the future failure of R (e.g., 01), the backup data of F (00) should be evenly reassigned from the old backup ring (11, 21, 31) of R (01) to the new backup rings of R. Suppose that each backup server B on the old backup ring (11, 21, 31) stores β GB of F's backup data that is previously mapped onto R (01). Since each backup server B (e.g., 11) is a new recovery server of R (01), B (11) only needs to recover its backup data to its 1-hop-away backup servers (B') on B's new backup ring (**10**, **12**, **13**), proportional to the number of recovery servers served by B': $\beta/5$ GB of backup data to 10, $2\beta/5$ GB to 12, and $2\beta/5$ GB to 13. Other old backup servers (21 and 31) have similar reassignment of their backup data, making each of the five new backup rings of R (01), namely, (**10**, **12**, **13**), (20, 22, 23), (30, 32, 33), (**12**, 22, 32) and (**13**, 23, 33), be assigned $3\beta/5$ GB of backup data from the old backup ring (11, 21, 31). For non-dominant backup data, since it does not participate in the normal recovery of primary data, MemCube does not reassign it unless the failure domain constraint is broken, as described in Section 4.3.

Resume services. After a new primary server P and P's backup servers complete the recovery of the new primary/backup data, P will update its mapping of the relevant data at its new recovery servers and the coordinator, and then P will notify the coordinator that its recovery is finished. P can choose (i) to wait for all the other new primary servers to finish their recoveries and then resume the services simultaneously (so that it will not affect others' recoveries), or (ii) to resume its service without waiting for others (so that its data can be accessed immediately). The two choices have no obvious difference in MemCube since by design all the recoveries are finished with similar time. Clients have a local cache of (part of) the mapping so that normally they can directly issue requests without querying the coordinator. If a client cannot locate a key, it fetches up-to-date information from the coordinator.

Recovery of *recovery server failure*. After a server F (e.g., 00 in Fig. 2) fails as a recovery server, for each of its primary servers P (e.g., 01), the (dominant) backup data on F's old backup ring (10, 20, 30) will be equally reassigned to the backup rings of P's remaining recovery servers R (11, 21, 31, 02, 03), in order to minimize the recovery time of P's future failure. Suppose that each backup server B on the old backup ring (10, 20, 30) stores β GB of P's (01) backup data that is previously mapped onto F (00). Then after F (00) fails, the backup data of B (10, 20, 30) will be reassigned to the 1-hop-away backup servers (B') on the backup rings of R (11, 21, 31, 02, 03), proportional to the number of recovery servers served by B': e.g., 10 will retain $\beta/5$ GB of backup data (for recovery server 11), transfer $2\beta/5$ GB to 12 (for 11 and 02), and transfer $2\beta/5$ GB to 13 (for 11 and 03). 20 and 30 have similar reassignment, making each of the five remaining backup rings be assigned $3\beta/5$ GB of backup data previously stored on 00's backup ring (10, 20, 30).

Recovery type	**Size**[1]	**From/to**[2]	**Length**	**# flows**[3]
Primary data of primary server	α	B→R	1-hop	br
Backup data of primary server	α	B→B B→R	1-hop	b^2r
Recovery server	$< \alpha$	R→B	1-hop	$(b-1)br$
Backup server	$f\alpha$	B→R	2-hop	$f(b-1)br$

[1] Total recovered size (assume a primary server stores α primary data).
[2] From the perspective of a failed *primary server*. R: recovery server. B: backup server. Bottleneck is R's inbound network bandwidth.
[3] # flows after the 1st failure. b: # backup servers on the backup ring. r: # recovery servers on the recovery ring. f: disk replication factor.

Table 1: Recovery summarization.

Recovery of *backup server failure*. After a server F (e.g., 00 in Fig. 2) fails as a backup server, its (dominant) backup data for each of its primary servers P (e.g., 11) is evenly divided and recovered from P (11) to P's 2-hop-away remaining backup servers (02, 03, 20, 30) on P's two backup rings where F (00) previously resided, to minimize the recovery time of P's future failure. Non-dominant backup data of F is recovered similarly.

Summarization. We summarize different types of recoveries in Table 1. (i) The primary/backup data recoveries of primary server failures are crucial to availability and performed concurrently. We note that there is contention between the two recoveries (B→R), but since the data size transferred to R in the backup data recovery ($S^{B\rightarrow R}_{Backup}$) is proportional to the number of new recovery servers served by R, it can be proved that $S^{B\rightarrow R}_{Backup}$ is between $\frac{1}{2b-1}$ and $\frac{1}{b}$ the size transferred to R in the primary data recovery, where b is the number of servers on the backup ring. Clearly it is negligible with relatively large b. (ii) The recovery of recovery server failures is not crucial but has no contention with primary server recovery, and thus could also be performed concurrently. (iii) The recovery of backup server failures has contention with primary server recovery (B→R), and thus should wait until the crucial recovery is finished. The deferred recovery has little affect on availability, and during this period the involved primary servers can service requests as usual, except that there is one less backup copy for the unrecovered backup data. The version numbers are used when multiple backup writes conflict (e.g., one from a new client write while another from the recovery of backup server failures).

4.3 Additional Failure Scenarios

Multiple failures. If multiple failures take place one by one, i.e., one failure happens after the previous failure has been *completely* recovered, MemCube recovers from each failure independently. Clearly the number of failures that the CubicRing structure can tolerate is bounded by the number of servers on each recovery ring: if all recovery servers of a primary server fail then the CubicRing structure fails. (If all backup servers of a recovery server R fail then it can be viewed as a failure of R.) In the worst case, e.g., the CubicRing structure on BCube$(16,2)$ can tolerate at least 44 failures, while the structure on BCube$(4,1)$ can tolerate at least 5 failures. Note that a *CubicRing* failure does NOT mean any data loss. This is because even though the data cannot be recovered to the recovery ring after the CubicRing structure fails, it still can be recovered to any healthy servers in MemCube.

Concurrent failures. When multiple failures happen concurrently, MemCube separately recovers from each failure, unless they are 1-hop or 2-hop neighbors. (i) If two directly-connected neighbors fail, e.g., during the recovery of a failed server F the coordinator cannot get responses from one of F's recovery servers, MemCube excludes them from each other's recovery ring, and recovers each failure as if it is a single failure. (ii) If a primary server and its backup server fail, the recovery server asks secondary servers for non-dominant copies.

Failure domain. MemCube guarantees after recovery none of the f backup copies are in the same domain. For instance, in the example of backup data recovery of a primary server (00) failure in Section 4.2, some dominant backup data may be reassigned to the same rack where the non-dominant data resides. E.g., the backup data previously stored on 11 is reassigned to (01, 21, 31) for the new primary server 10. In this case MemCube will reassign the affected non-dominant data to a new secondary server in a different rack.

Switch failures. In traditional storage systems a ToR switch failure results in a recovery storm, where all the abandoned servers connected to that switch are actually alive. In contrast, MemCube handles switch failures simply by leveraging the multiple paths between any two servers. Since any k switch failures in BCube(n,k) result in only graceful degradation [35] but no data loss or unavailability, it is not critical and the failed k switches can be replaced in a relatively long period of time.

5 Discussion

Over-provisioning ratio. If a primary server fails, its recovery servers must have enough RAM to accommodate the recovered data. So the RAM of all the servers need to be over-provisioned beforehand. We obtain the following Theorem 2 for the *over-provisioning ratio* (θ), the formal proof of which is given in Appendix B. In BCube$(16,2)$, e.g., if $\theta = 1.15$ then at least 6 failures can be tolerated; and if $\theta = 1.4375$ then at least 14 failures can be tolerated. In contrast, RAMCloud does not have a deterministic θ due to its randomized data placement. Note that similar to the CubicRing failure discussed in Section 4.3, if no enough RAM available on some specific servers MemCube can be simply "degraded" to RAMCloud without any data loss.

Theorem 2 *Consider a MemCube on BCube(n,k) and suppose that before any failures each server installs α GB of RAM and stores β GB of data. We define the* ***over-provisioning ratio*** *as $\theta = \alpha/\beta$. If we want to keep the CubicRing structure after the r^{th} failure in the worst case, we should have $\theta \geq 1 + \frac{r}{nk+n-k-r}$.*

Fragmentation. After a primary server fails MemCube recovers its data to multiple new servers, on which the recovered fragmented data may lost locality. Although locality has no effects on our current data model, this issue might become important if MemCube supports richer models in the future. Given a set of KVs (S), we define the *fragmentation ratio* (μ) as the initial number of primary servers responsible for S divided by the current number of servers for S. Higher μ means lower fragmentation and thus is desired for better locality. As discussed in Section 3.2, the number of recovery servers involved in a recovery is configurable. Larger numbers increase the aggregate bandwidth but result in higher fragmentation, and vice versa. We study the tradeoff between aggregate bandwidth and μ in Section 6.4. A simple method for defragmentation is to replace the failed server with a new one and restore the data.

Heterogeneity and stragglers. The backup servers may have different parameters of disk/CPU/RAM/network resources. MemCube handles heterogeneity by assigning backup data according to the bottleneck resource. E.g., if the network bandwidth of backup servers is the bottleneck for recovery, MemCube will assign backup data to them proportional to their bandwidth [45] so that they can finish the recovery with similar time.

MemCube uses a simple method to handle stragglers. Since the numbers of servers on the recovery/backup rings and the bandwidth of each server are known, MemCube can compute the expected recovered size for each server given a time window. A recovery server (R) periodically computes for each of its backup servers (B) the ratio (π_R^B) of the data size recovered from B to the expected recovered size from B within the last period. If π_R^B is lower than a pre-defined threshold, then B will be considered as a straggler and R will use B's secondary server instead. The coordinator identifies stragglers from recovery servers in a similar way, and it will reassign the straggler's responsible data to other healthy recovery servers. In both cases non-local recovery will occur, but we expect little impact on the overall performance.

Consistency guarantees. A complete discussion of consistency guarantees is beyond the scope of this work, and here we briefly discuss the main factors affecting consistency. False positives are inevitable in failure detection. Therefore, MemCube adopts *atomic recovery*, where once the coordinator declares a server P dead, it will ask all P's backup servers to (i) reject any further backup requests from P and (ii) indicate P to stop its service. Buffered backup writes from P before the declaration should be finished since they have been returned. Primary servers also periodically contact with their backup servers so that they can stop servicing pure read requests after being declared dead. Therefore, false detection in MemCube is not *fatal* (but *expensive*).

MemCube uses ZooKeeper [39] enabled coordinators to store its global mapping information between the key space and the primary servers. There are one active coordinator and several standby coordinators which are competing for a single lease, ensuring at most one coordinator to be responsible at a time. After the active coordinator fails, some standby coordinator will acquire the lease and become active. If a server fails concurrently with a coordinator failure, e.g., the recovery servers R cannot get response from the coordinator, R will ask the ZooKeeper service to locate the new active coordinator and then report the failure to it. Afterwards the normal recovery procedure is performed.

Since there are $k+1$ paths between any two servers in BCube(n,k), MemCube is unlikely to have a network partition. If this happens, an operator can stop the entire system and wait until the network recovers. Similarly, non-transitive failures [52] are unlikely since all paths to a suspiciously failed server are tested.

Operational Issues. MemCube is designed on top of BCube, which has similar cost and wiring complexity with FatTree. For isntance, both BCube and FatTree use 128 wires for building our 64-server testbed (discussed in Section 6). Clearly there might be a bandwidth waste in MemCube if the network is not busy, but the advantages of BCube include not only high bandwidth and throughput, but also fast failure detection and recovery, graceful degradation during switch failures, etc. Also we note that BCube uses COTS switches/NICs, and thus the extra cost is low and acceptable.

As described in Line 2 of Pseudocode 1 in Section 4.2, for minimizing the recovery time MemCube stops all the relevant services during the recovery. For BCube$(16,2)$ with 4096 servers, for example, given the normal failure rate (about $1 \sim 2$ failures/server/year [7]) and the recovery time (a few seconds as shown in Section 6), the "background" network utilization of recovery traffic is less than 10^{-4}.

A dangerous situation is that the entire system loses power at once. A simple way to address this problem is to install on each server a small backup battery. The battery ONLY needs to extend the power long enough to ensure that the backup server's *buffered* backup data (that is yet to be written to disks) be flushed. When power returns the cold start is performed like many concurrent recoveries of all servers.

6 Evaluation

We have implemented a prototype of MemCube by adding a *MemCube module* to memcached-1.4.15 on Linux, which contains: (i) a *connection manager* that maintains the status of neighbors and interacts with other servers; (ii) a *storage manager* that handles *set/get/delete* requests in a server's RAM and asynchronously writes backup data to disks by appending the data to its on-disk log that is divided into 8MB tracts; and (iii) a *recovery manager* that reconstructs primary/backup data (and the corresponding mapping) on the new primary/backup servers and inspects the recovery process. We also implement a simple *global coordinator* that maintains the configuration, the addresses of servers, and the mapping between the key space and the servers along with the size of data stored in each server's RAM.

6.1 Testbed

We have built a testbed with 64 PowerLeader servers and five Pronto 3780 48-port 10GbE switches. Each server has 12 Intel Xeon E5-2640 2.5GHz cores and 64 GB RAM, and installs six Hitachi 7200 RPM, 1 TB disks and one 10GbE 2-port NIC.

We use four switches to construct a 64-node BCube(8,1) network to run MemCube, where each switch acts as four 8-port virtual switches and connects to 32 servers. We also use the five switches to build a 64-node tree and a 64-node FatTree to emulate and test RAMCloud [45] on Ethernet. For tree, we simply have each of four switches connect to 16 servers and the fifth switch act as the aggregate switch, getting a relatively high over-subscription ratio of 1 : 16. For FatTree, we use three switches in the first level and two in the second. In the first level we use two switches to connect to 24 servers each and act as three 8-port virtual switches, and use the third switch to connect to 16 servers and act as two virtual switches; and each switch has the same number of ports connected to the second level switches as that to the servers. In the second level each switch acts as four 8-port virtual switches. We also build a 1 : 4 oversubscribed FatTree where the first level has

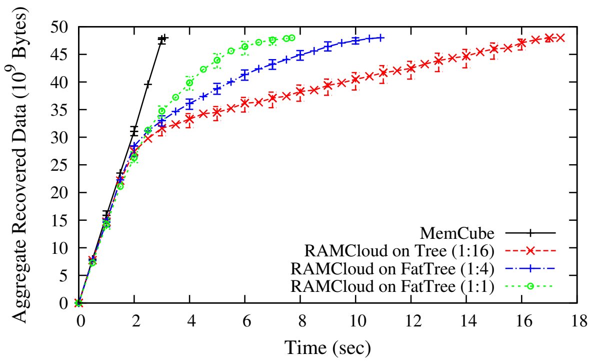

Figure 4: Single server failure recovery.

4 switches each being connected to 16 servers and the second level has the fifth switch being connected to four ports of each of the first-level switches. FatTree uses ECMP routing with hash-based path selection to achieve load balancing. Our testbed therefore supports both MemCube on BCube and RAMCloud on tree/FatTree.

All experiments use a disk replication factor of 3, i.e., 1 *primary* copy in RAM and 3 *backup* copies on servers' disks. Clearly, given the disk bandwidth of 100∼200 MB/sec and the number (6) of disks per server, the network bandwidth of 10 Gb/sec, and the ratio of recovery servers to backup servers (14/49), the bottleneck is at the inbound network bandwidth of recovery servers.

A client initially fills the 64 primary servers each with 48 GB of primary data. During the process we measure the write throughput of one MemCube primary server. We slightly modify the Redis benchmark [13, 26] to adapt to MemCube, which uses a configurable number of busy loops asynchronously writing KVs. The maximum write throughput of a single MemCube primary server is about 197.6K writes per second when it runs 8 single-threaded service processes each corresponding to 200 loops. In contrast, the maximum throughput of an *unmodified* Memcached server is about 225.5K writes per second when it runs 4 single-threaded service processes each corresponding to 250 loops.

After a failure happens, the recovery is conducted following Pseudocode 1. Our evaluation answers the following questions: How fast can MemCube recover a single server failure, even with stragglers (§6.2)? How well does MemCube perform under various patterns of failures (§6.3)? And what is the impact of using different number of recovery/backup servers (§6.4)?

6.2 Single Server Failure Recovery

We first evaluate the recovery of a single server failure in MemCube. The client sends a magic RPC to a primary server that kills its service process. The recovery procedure is started after waiting 300 milliseconds of heartbeat timeout. The coordinator waits until all the primary/backup data is recovered and reports the size of the aggregate recovered (primary) data over time. We also evaluate RAMCloud [45] both on tree and on FatTree with ECMP [34], where each primary server uses 14 recovery servers and 49 backup servers (which are the same as in MemCube).

(*n*,*k*) # servers	**(4,3)** 256	**(8,2)** 512	**(8,3)** 4096	**(16,1)** 256	**(16,2)** 4096
MemCube	1.20	1.01	0.51	1.44	0.48
RAMCloud	26.59	42.98	83.40	22.68	40.59

Table 2: Simulated recovery time (in seconds).

The result is depicted in Fig. 4. Each point is an average of 5 runs except the last points because the fast runs may have completed. MemCube recovers 48 GB of data in 3.1 seconds. The aggregate recovery throughput is about 123.9 Gb/sec, very close to the optimal aggregate bandwidth bounded by the NIC bandwidth and the number of recovery servers. Every recovery server achieves the recovery throughput of about 8.85 Gb/sec.

The recovery process of RAMCloud is also depicted in Fig. 4. On tree, RAMCloud has similar performance with MemCube in the beginning but gets a dramatic degradation after 2 seconds. This is because the recovery servers randomly choose their new backup servers without a global view of the network, and the tree has an over-subscription ratio of 1 : 16 which generates severe congestion at the root. At beginning local flows within a switch saturate the recovery servers' NICs, the aggregate bandwidth of which is the same as that in MemCube. But after the local flows complete the aggregate recovery bandwidth will drop. Non-blocking FatTree is designed to alleviate this problem, but since ECMP randomly selects paths for the flows, the full bisection bandwidth is not guaranteed but only stochastically *likely* across multiple flows. Thus long-lived recovery flows are problematic with ECMP and RAMCloud (both on tree and on FatTree) experiences long recovery time. Note that the results in Fig. 4 are worse than that in [45], where RAMCloud recovers 35 GB of data in 1.6 seconds in a 60-node cluster. This is because in [45] (i) RAMCloud uses 5 32Gbps-InfiniBand switches to build the testbed (while we emulate 16 8-port 10GbE switches); and (ii) it uses all the nodes as recovery servers for the failed server (while we use only 14 recovery servers).

We also evaluate the recovery for larger scales of MemCube (on BCube(n,k)) and RAMCloud (on non-blocking FatTree) through simulations. Since in most cases the bottleneck is at the bandwidth of recovery servers, we simplify the simulations by using NS2 [8] to simulate the process of transferring primary/backup data for the failed server which has 48 GB of primary data. The result is summarized in Table 2, where the first

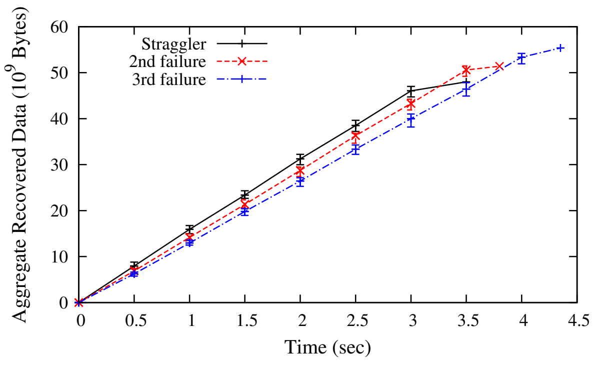

Figure 5: Straggler and multiple failures.

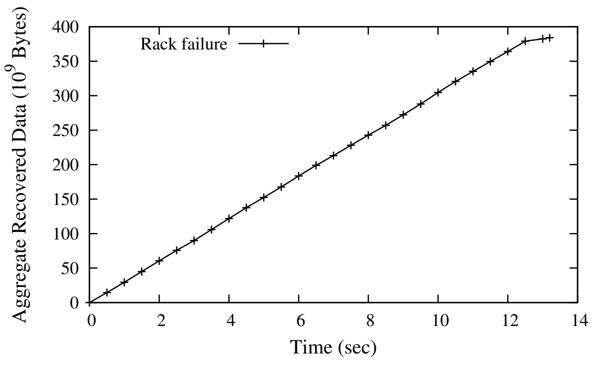

Figure 6: Rack failure recovery.

row lists different combinations of (n,k) and the numbers of servers, and the next two rows respectively show the corresponding recovery time (in seconds) for MemCube and RAMCloud. Note that we only simulate the process of primary/backup data transfer and ignore the failure detection time (a few hundred milliseconds), the coordination time (100+ milliseconds), the time of reading the first tract from disks (about 100 milliseconds), and the potential bottleneck at CPU which is because a recovery server uses k NIC ports for recovery. Therefore, although the simulated recovery time for both BCube(8,3) and BCube(16,2) is only about half a second, in practice it would be difficult to recover faster than 1 second.

We emulate a straggler during the recovery by limiting a backup server's outbound bandwidth to $1/3$ the bandwidth in normal recovery ($\frac{123.9}{49\times 3} \approx 0.84$ Gb/sec). In this experiment, every new primary server R computes π_R^B (defined in Section 5) for each of its backup server B every half a second, the threshold is set to 0.7, and the straggler occurs 1 second after the recovery begins. The recovery procedure is depicted in Fig. 5 (denoted as *Straggler*), each point of which is an average of 5 runs. The result shows that MemCube performs well after the straggler occurs by using other backup flows to saturate the recovery server's spare bandwidth. After 1.5 seconds MemCube will detect the straggler and use its secondary server instead, which finishes its recovery at about 3.5 seconds. The additional time compared to MemCube's normal recovery is because the straggler recovers less data than others between 1 and 1.5 seconds.

6.3 Multiple Server Failures Recovery

We evaluate the recovery of multiple failures with the one-by-one pattern, i.e., one failure happens after the previous failure has already been completely recovered. All subsequent failures happen on the same recovery ring of the first failure, which generates the worst-case scenario. The result is depicted in Fig. 5, which shows the size of recovered data over time for the second and third failures. Each point is an average of 5 runs. This figure shows that there is a graceful degradation of recovery performance as failures happen one by one, mainly due to the decrease of the number of recovery servers and the increase of the primary data size.

We then emulate a rack failure by sending a magic RPC to the 8 servers connected to an 8-port virtual switch to kill their MemCube processes. Currently our prototype only supports rack failure recovery of primary/backup data for primary servers failures (Lines 4 and 5 in Pseudocode 1), but extending for supporting the other two types of recovery (Lines 7 and 8) is straightforward. The recovery procedure is depicted in Fig. 6, where each point is an average of 5 runs and the differences to the mean are less than 5% (omitted here for clarity). MemCube recovers a rack failure of 8 primary servers in about 13.2 seconds, Compared with the single failure recovery, the recovered data size increases by 8×, the total number of recovery servers increases by 4×, and the recovery time increases by about 4×, meaning the per-server recovery throughput is only about $1/2$ that in single failure recovery. This is because both primary data and backup data are recovered from *all* servers to *all* servers, in contrast in single failure recovery (as discussed in Section 4.2) only $\frac{1}{2b-1} = 1/13$ of the backup data recovery contends with the primary data recovery, where $b = 7$ is the number of servers on a backup ring. Clearly, even when multiple primary-recovery-backup structures overlap there is still no severe competition during the recoveries of multiple concurrent failures.

MemCube handles a switch failure with graceful performance degradation by leveraging the multiple paths of BCube. To evaluate this, we first measure the write throughput of a primary server, disable a switch connected to that server, wait for 1 second, and then measure the write throughput again. Running 8 single-threaded server processes, before the switch failure the write throughput $\approx$ 197.6K writes/sec. After the switch failure the write throughput $\approx$ 162.2K writes/sec with a degradation of less than 18%, which is mainly because the redundant paths traverse more intermediate nodes.

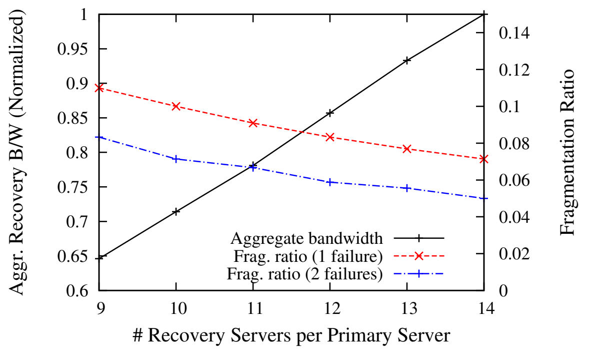

Figure 7: Tradeoff of different # recovery servers.

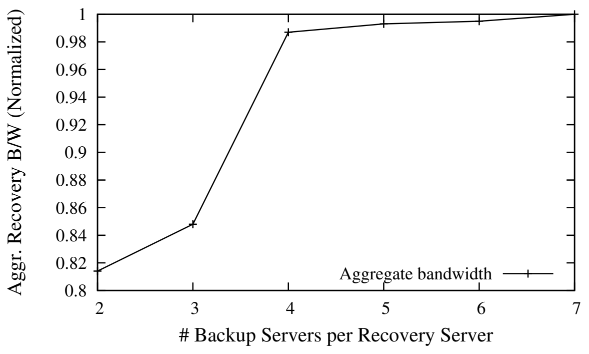

Figure 8: Impact of different # backup servers.

6.4 Impact of # Recovery/Backup Servers

We study the impact of different number of recovery servers used by a primary server on the average aggregate recovery bandwidth and the *fragmentation ratio* (introduced in Section 5). The bandwidth is computed as the recovered size (of primary data) divided by the recovery time and normalized to the baseline with all the 14 recovery servers being used. Each recovery server adopts all its 7 backup servers. As shown in Fig. 7, with the increase of the number of recovery servers, MemCube gets an almost linear speedup for the bandwidth, while the ratio decreases (meaning more severe fragmentation) after the recovery of both 1 and 2 failures. In practice MemCube can choose the tradeoff between fragmentation and recovery bandwidth by using different number of recovery servers accordingly.

We study the impact of different number of backup servers (b) used by a recovery server on the aggregate recovery bandwidth. The primary server uses all its 14 recovery servers. Since when b is small the bandwidth of backup servers (BW_B) may become the bottleneck instead of the bandwidth of recovery servers (BW_R), the primary data is assigned to the recovery servers R according to R's $\min(BW_R, \sum_{B \in Ring(R)} BW_B)$. As shown in Fig. 8, when the number of backup servers is small ($b = 2,3$) the bandwidth of backup servers is the bottleneck. This is because the data sent by backup servers is twice as much as that received by recovery servers due to the concurrent recovery of backup data (and note that 1 backup server serves 2 recovery servers). When b increases to 4, the aggregate bandwidth is almost the same as the baseline ($b = 7$), because since then the performance is again bounded by the bandwidth of recovery servers. Although the number of backup servers has little impact on the recovery time when $b \geq 4$, we suggest to use all the backup servers to (i) achieve high CubicRing durability, and (ii) prevent the aggregate disk bandwidth to become a bottleneck when the number of disks per server is smaller than our configuration ($= 6$).

7 Related Work

Efficient KV. The idea of permanently storing data in RAM is not new. E.g., in-memory databases [30] and transaction processing systems [40] keep entire databases in the RAM of one or several servers and support full RDBMS semantics. RAMCloud [45] is proposed as a large-scale in-memory key-value store where the data is kept entirely in RAM and the backup copies are scattered across many servers' disks. RAMCloud utilizes high-bandwidth InfiniBand networks to achieve fast failure recovery. MemCube inherits some key designs of RAMCloud including the primary-recovery-backup architecture and the coordinator for key space management. MemCube improves RAMCloud by leveraging CubicRing to address several critical network-related issues, including false failure detection due to transient problems, recovery traffic congestion, and ToR switch failures. Besides, MemCube is implemented on Ethernet which has cost and scalability benefits.

Redis [13] is a key-value store that keeps all data in RAM. Redis has a richer data model than MemCube, e.g., atomic increment and transactions. However, it can prevent data loss ONLY when it is used in the *flushing mode*, where every write has to be logged to disks before it returns. MemC3 [28] improves Memcached [9] by incorporating the CLOCK replacement algorithm [1] and Concurrent Cuckoo hashing [47]. It serves up to 3× as many queries per second. MemCube can easily migrate from Memcached to MemC3 for higher throughput.

Flash memory is receiving increasing attention for flash-based storage systems (e.g., SILT [42], FAWN [12], FlashStore [23], SkimpyStash [24], and HashCache [11]). One disadvantage of in-memory storage systems is the high cost and energy usage per bit. However, when considering cost per operation, RAM is about 1000× more efficient than disk and 10× than flash memory [46]. Andersen et al. [12] and Ousterhout et al. [46] separately generalize Jim Gray's rule [33] and conclude that (i) for high access rates and small data sizes RAM is the cheapest; and (ii) the applicability of in-memory storage will be continuously increasing.

Detection/recovery. Failure detection has been widely studied in the context of monitoring remote elements by using end-to-end timeouts [18, 14, 20, 50, 37]. E.g., the ϕ-accrual detector [37] provides a measurement of detection confidence and lets applications decide corresponding actions. Recently, Falcon [41] and Pigeon [36] propose to install sensors to obtain low-level information of hardware, OS, processes, and routers/switches, to aid diagnosis. These works are complementary to MemCube and can help to provide more accurate detection. E.g., MemCube could use Pigeon to check backup servers' SMART [51] data to pre-warn disk problems. MemCube's local detection eliminates the necessity of installing code in switches (which makes Pigeon less applicable in real deployment) for congestion detection. Host failure recovery techniques (e.g., microreboot [21, 17]) focus on masking and containing failures, which can be directly applied to MemCube: a MemCube recovery will not take place until failures cannot be masked.

FDS [44] is a locality-oblivious blob store. It recovers 92 GB of data in parallel in 6.2 seconds on a 256-server 10GbE FatTree, which is less efficient compared with MemCube. Thus although FDS claims "locality is unnecessary", we show locality does matter in fast failure recovery. Similar to FDS, D-Streams [54] use parallel recovery for reliable distributed stream processing.

8 Conclusion

This paper's top-level contribution is architectural: We suggest to exploit network proximity in distributed systems to restrict failure detection and recovery within the smallest possible range, in order to minimize the uncertainty and contention induced by the network. We apply this principle to fast failure recovery of an in-memory key-value store (MemCube) by constructing the CubicRing structure: All the servers form a primary ring, and for each primary server its 1-hop neighbors form a recovery ring and 2-hop neighbors form backup rings. As failures happen, MemCube (i) leverages the CubicRing structure to quickly recover lost data, and (ii) maintains the structure.

We plan to improve MemCube in several aspects including rich data model, indices, efficient log cleaning/optimizing, super columns [19], strong consistency (i.e., linearizability [38, 32]) guarantees, and low-latency serializable transactions [56]. We also plan to implement automatic reassignment of the key space mapping when the load distribution dynamically changes. On the other hand, MemCube depends on cubic topologies, and how to apply the proposed principle to tree-based networks (e.g., FatTree) is still an open issue.

Acknowledgement

This work was supported by the National Basic Research Program of China (973) under Grant No. 2011CB302601, and the National Natural Science Foundation of China (NSFC) under Grant No. 61379055 and 61222205. We thank Ji Wang, Huiba Li, Zhihui Sun, Rodrigo Fonseca, the Pronto tech support team, and the anonymous reviewers for their help to improve this paper. This work was performed when the first author was visiting MSRA, and we thank Guohan Lu and Jiaxin Cao for the insightful comments. Some parts of the paper were written when the first author was visiting the NetOS group of Computer Lab at University of Cambridge, and we thank Professor Jon Crowcroft for his accommodation and Anil Madhavapeddy, Ripduman Sohan and Hao Liu for the discussion. Some preliminary results of this work were presented at HotCloud'12.

Appendix

A. Proof of Theorem 1

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

By Theorem 1, at the beginning there are $(n-1)(k+1)$ servers on the recovery ring. Let $m=(n-1)(k+1)$. After the first server fails, MemCube 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, MemCube 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 by parity of reasoning, after the r^{th} failure (reasonably assuming $m-r>>1$), MemCube should satisfy $\beta(1+\frac{r}{m-r+1})\leq\alpha$. Therefore, if we want to keep the CubicRing structure after the r^{th} failure in the worst case (where a subsequent failure always happens on a server that is a recovery server in the previous failure recovery), the over-provisioning ratio θ should satisfy $\theta=\alpha/\beta\geq 1+\frac{r}{m+1-r}=1+\frac{r}{nk+n-k-r}$.

References

[1] http://books.google.com/books?id=5wDQNwAACAAJ.

[2] http://kylinx.com/papers/cmem1.4.pdf.

[3] https://ramcloud.stanford.edu/wiki/pages/viewpage.action?pageId=8355860sosp-2011-reviews-and-comments-on-ramcloud.

[4] http://wiki.open.qq.com/wiki/CMEM.

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

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

[7] http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/.

[8] http://www.isi.edu/nsnam/ns/.

[9] http://www.memcached.org/.

[10] 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.

[11] 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.

[12] 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.

[13] ANTIREZ. An update on the memcached/redis benchmark. http://antirez.com/post/update-on-memcached-redis -benchmark.html.

[14] BERTIER, M., MARIN, O., AND SENS, P. Implementation and performance evaluation of an adaptable failure detector. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings* (2002), pp. 354–363.

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

[16] BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F. B., AND TOUEG, S. The primary-backup approach, 1993.

[17] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot - A technique for cheap recovery. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004* (2004), pp. 31–44.

[18] CHANDRA, T. D., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (1996), 225–267.

[19] 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.

[20] CHEN, W., TOUEG, S., AND AGUILERA, M. K. On the quality of service of failure detectors. *IEEE Trans. Computers 51*, 5 (2002), 561–580.

[21] CULLY, B., LEFEBVRE, G., MEYER, D. T., FEELEY, M., HUTCHINSON, N. C., AND WARFIELD, A. Remus: High availability via asynchronous virtual machine replication. (best paper). In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings* (2008), p. 161.

[22] 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.

[23] DEBNATH, B. K., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. *PVLDB 3*, 2 (2010), 1414–1425.

[24] 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.

[25] 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.

[26] DORMANDO. Redis vs memcached (slightly better bench). http://dormando.livejournal.com/525147.html.

[27] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 401–414.

[28] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013* (2013), pp. 371–384.

[29] FERNER, C. S., AND LEE, K. Y. Hyperbanyan networks: A new class of networks for distributed memory multiprocessors. *IEEE Transactions on Computers 41*, 3 (1992), 254–261.

[30] GARCIA-MOLINA, H., AND SALEM, K. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng. 4*, 6 (1992), 509–516.

[31] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM* (2011), pp. 350–361.

[32] GLENDENNING, L., BESCHASTNIKH, I., KRISHNAMURTHY, A., AND ANDERSON, T. E. Scalable consistency in scatter. In *SOSP* (2011), pp. 15–28.

[33] GRAY, J., AND PUTZOLU, G. R. The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for cpu time. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987* (1987), U. Dayal and I. L. Traiger, Eds., ACM Press, pp. 395–398.

[34] GREENBERG, A. G., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Vl2: a scalable and flexible data center network. *Commun. ACM 54*, 3 (2011), 95–104.

[35] 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.

[36] GUPTA, T., LENERS, J. B., AGUILERA, M. K., AND WALFISH, M. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013* (2013), pp. 427–441.

[37] HAYASHIBARA, N., DÉFAGO, X., YARED, R., AND KATAYAMA, T. The Φ accrual failure detector. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil* (2004), pp. 66–78.

[38] HERLIHY, M., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[39] 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.

[40] 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.

[41] LENERS, J. B., WU, H., HUNG, W., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011* (2011), pp. 279–294.

[42] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: a memory-efficient, high-performance key-value store. In *SOSP* (2011), pp. 1–13.

[43] MAHAJAN, P., SETTY, S. T. V., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud storage with minimal trust. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings* (2010), pp. 307–322.

[44] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., , AND SUZUE, Y. Flat datacenter storage. In *OSDI* (2012).

[45] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J. K., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *SOSP* (2011), pp. 29–41.

[46] 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.

[47] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *J. Algorithms 51*, 2 (2004), 122–144.

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

[49] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's time for low latency. In *HotOS* (2011).

[50] SO, K. C. W., AND SIRER, E. G. Latency and bandwidth-minimizing failure detectors. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007* (2007), pp. 89–99.

[51] STEVENS, C. E. At attachment 8 - ata/atapi command set. *Technical Report 1699, Technical Committee T13* (2008).

[52] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013* (2013), pp. 357–370.

[53] 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.

[54] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP* (2013).

[55] 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.

[56] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP* (2013), pp. 276–291.