# Research for Architecture and Management in RAMCloud

# Liwei Gao

# The George Washington University CSCI 6461

Abstract

RAMCloud is a new solution to the storage for large-scale datacenter applications. It is designed as a distributed key-value store that keeps data entirely in DRAM. By using secondary storage backup, RAMCloud is durable and enables to recover when servers crash. With high-speed Internet and advantage of fast I/O in DRAM, the system could provide low-latency responses to user requests even if the applications are very large. In this paper, we would introduce the basic structure of RAMCloud model. This paper also provides an overview of the modern solutions to classic issues in RAMCloud, including log structure, balancer, coordinator, and fast recovery after crash.

**Introduction**

In the past 4 decades, the evolution of disk was uneven. On one hand, the capacity of magnetic disks has improved very rapidly. As shown in table 1 (Ousterhout et al., 2010), the disk capacity was only 30MB in mid 1980s, but the figure rose to 500GB in 2009. It escalated for almost 1000 times. On the other hand, the performance of disks has not developed as well as its capacity: maximum transfer rate just increased 50 times while latency only doubled itself. By using Jim Gray’s Rule (Gray & Putzolu, 1987), we may see that access latency of disk actually 360x worse than before. As Ousterhout et al. (2010) said, as the desired access rate to each record increases, the cost per usable bit of disk will increase; eventually is no better than DRAM. It seems DRAM is an alternative for disk in the future. Actually, lots companies have already used DRAM as their storage nowadays. For example, Google and Yahoo have stored all their indexes in DRAM. Google even kept its snapshots in RAM.

Table 1

*Disk Comparison Today and 25 Years Ago*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Mid-1980s | 2009 | Improvement |
| Disk capacity | 30 MB | 500 GB | 16667x |
| Maximum transfer rate | 2 MB/s | 100 MB/s | 50x |
| Latency (seek + rotate) | 20ms | 10ms | 2x |
| Capacity/bandwidth (large blocks) | 15 s | 5000s | 333x worse |
| Capacity/bandwidth (1KB blocks) | 600 s | 58 s | 8333x worse |
| Jim Gray’s Rule (1KB blocks) | 5 min | 30 hours | 360x worse |

In the meantime, the architecture of applications has changed a lot in recent year, mainly because of the development of Internet. Unlike the traditional file system and relational database, the large-scale web applications tend to store and compute in different physical locations. Concretely, as the figure 1 (Ousterhout et al., 2010) shows, besides the application servers for logic businesses and front-end loaders, the data centers also need specific storing servers for data. These application servers are commonly stateless, which are helpful when expended to thousands servers and response to hundreds of million users. Nevertheless, as a web application grows it must undergo a series of revisions. Traditional techniques solve the issue by introducing ad hoc, which might not work so well when the applications reach to a new level of scale or a new feature is introduced. For example, it costs average 130 internal requests and multiple accesses to hard disks when some large-scale websites, such as Facebook and Amazon, receive a new HTTP requests as a part of generating HTML pages. If the scale of servers grew for 4-5 levels, the complexity of the web application be increased and latency of disks be unaffordable.

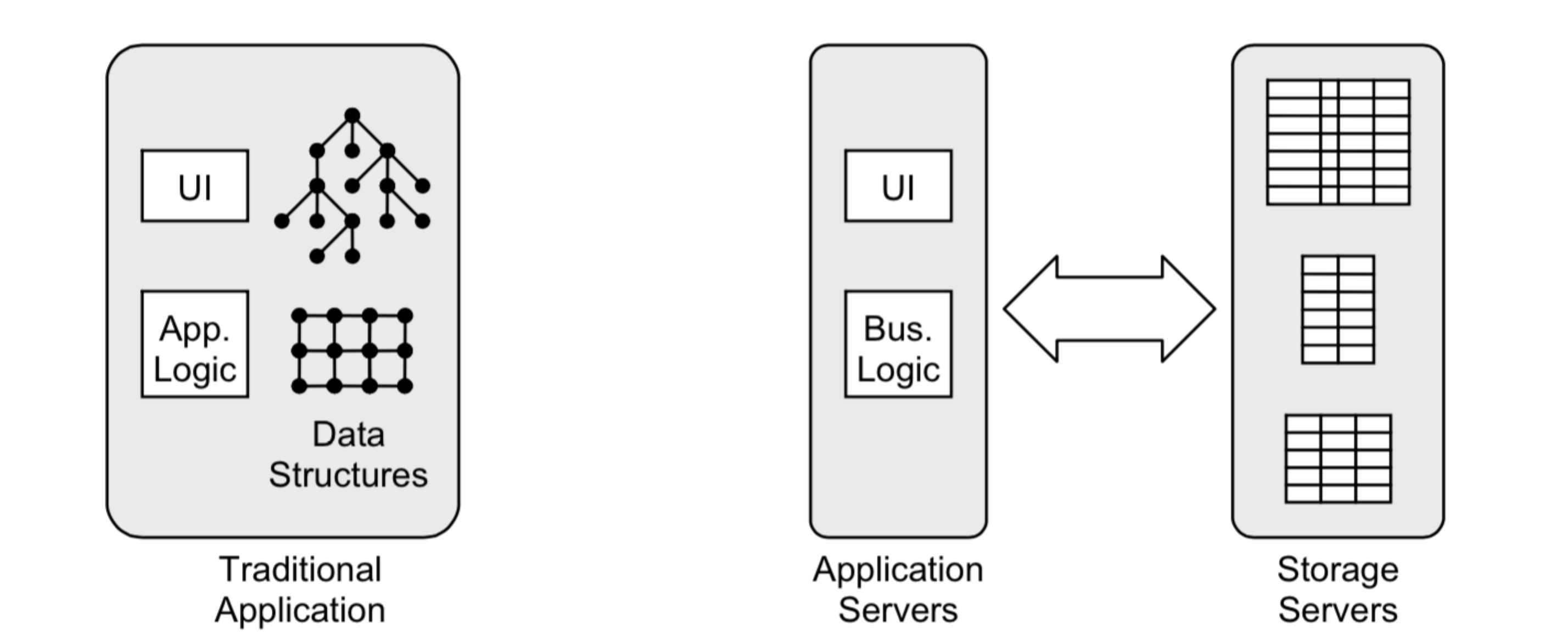


Figure 1. Comparison of traditional and large-scale web applications.

Motivated by the above two issues, large quantities of researches have been done in these years. The classical solutions include: memcached (Fitzpatrick, 2004), database partitioning, flash disk, SSD, MapReduce (Dean & Ghemawat, 2008), Hadoop (Shvachko, Kuang, Radia, & Chansler, 2010), NoSQL (Cattell, 2011), Distributed File System and so on. With the support of Facebook, Mellanox, NEC, NetApp and SAP, the team leaded by Ousterhout gave us a new solution named RAMCloud. It stores all of its information in the main memories of commodity servers, using hundreds or thousands of such servers to create a large-scale storage system. It moves the data center from hard disks to DRAMs, and uses the disks for backup or category. By using replication and backup, RAMCloud maintains the durability and availability of traditional disk-based file systems. In addition, benefited from DRAM, the latency of data I/O drops magically. RAMCloud seems to meet the compatibility of the large scale (100-1000 TB) and low latency (5-10ms for access to data in datacenter, 100-1000x better).

Despite the advantages of RAMCloud, Outerhout’s team also face some challenges. Firstly, the RAMCloud has thousands of physical nodes and all data are in these nodes. As we know, the DRAM is volatile memory. We need to recover the data of the node efficiently after its crash. Secondly, every node has many invalid data, which stored both in DRAM or disk. If we keep the redundancy data, it will degrade perform of RAMCloud system. So a new technique should be introduced to distinguish the invalid data in disk and keep invalid data from reviving. Finally, when there are lots of invalid data in a segment, we normally merge them to save space in DRAM. It is analogous to the garbage collection of SSD. However, the garbage collection in RAMCloud system will consume large quantities of resource.

In this paper, we argue the structure of RAMCloud (section 2) as well as the solutions to the classical issues (section 3-6), which includes the log structure, two candidate balancers, RAFT algorithm used in coordinator and fast recovery in cluster.

**Architecture of RAMCloud**

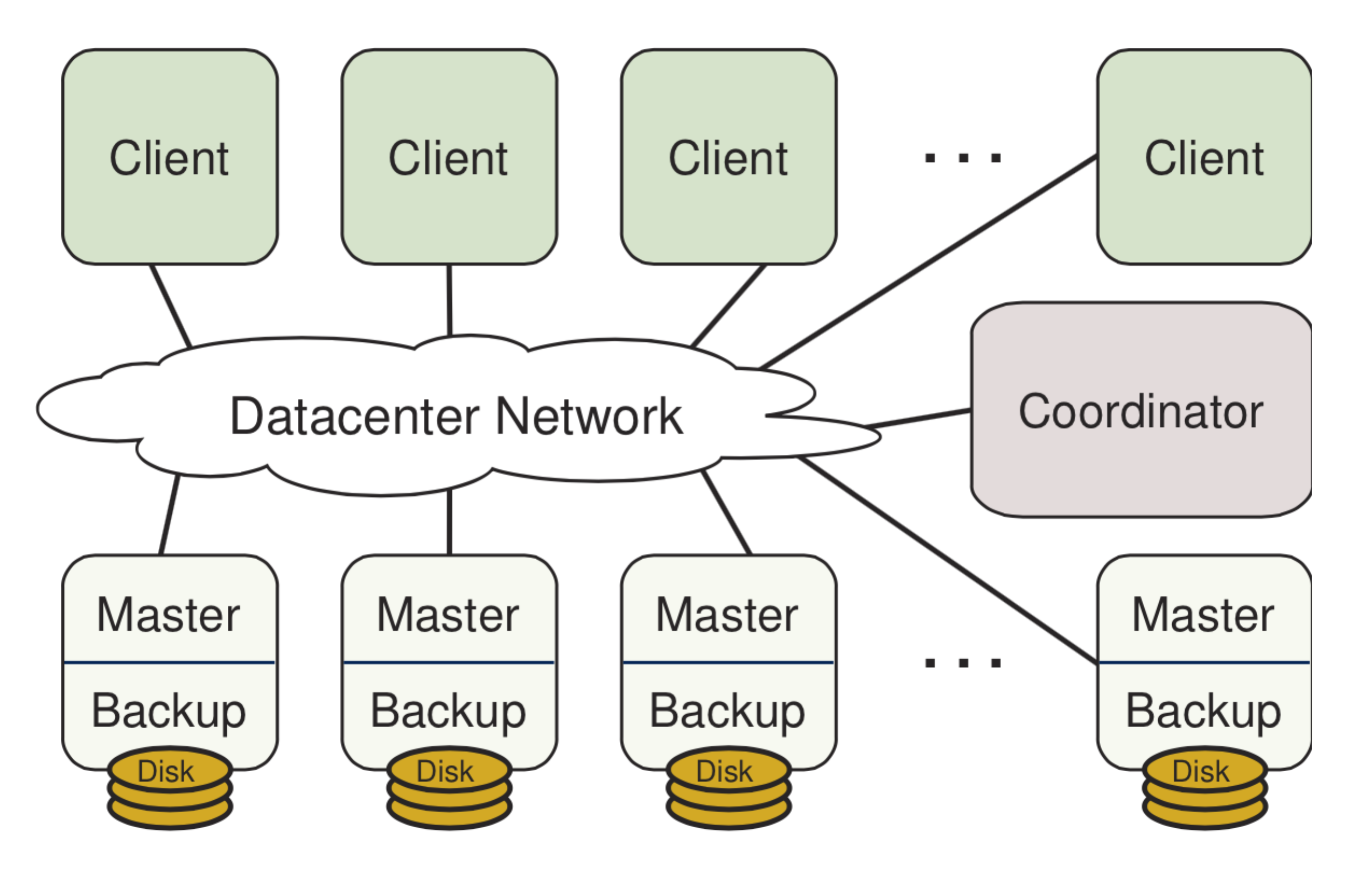
RAMCloud is a storage system that could keep data in DRAM of thousands of servers with durability, just like the disk server. The ideal RAMCloud system takes use of high speed Ethernet to offer remote I/O for low latency. It takes the system cluster 5-10μs to read hundreds of bytes from one server to another. In real world, the time is a little longer. It costs 0.5-10ms to do the same work, depending on whether the data are in the DRAM or not. When it comes to input, writing for small objects usually needs 16μs. The system based on disk could only do 1000-10000 tiny read requests per second. But if we use RAMCloud, the number rises to 1 million per second.

Figure 2. RAMCloud cluster architecture.

As shown in the figure 2 (Rumble, 2014), a RAMCloud system consists of 3 parts: a set of storage servers, coordinator and clients. Each storage server has two components: master and backup. Master module manages the objects stored in memory of the server. Backup module, on the other hand, uses the local manganic disk or SSD to store the backup copy of data from other servers. Coordinator manages the profile of masters and backups, such as the memberships and data distributions among the cluster of servers. Since coordinator does not involve in the data reading and writing, it will not be the bottleneck of cluster.

RAMCloud provides a simple key-value data model, which consists of interpreted data blobs called objects (Rumble, Kejriwal, & Ousterhout, 2014). Objects are tagged by variable-length keys and grouped in table that could span one or more servers. Object should also keep its integrity when they were written or read. RAMCloud has done some optimization for small objects (a few hundred bytes or less), but may not support objects up to 1MB. Fortunately, this is enough to handle large-scale web concurrent requests.

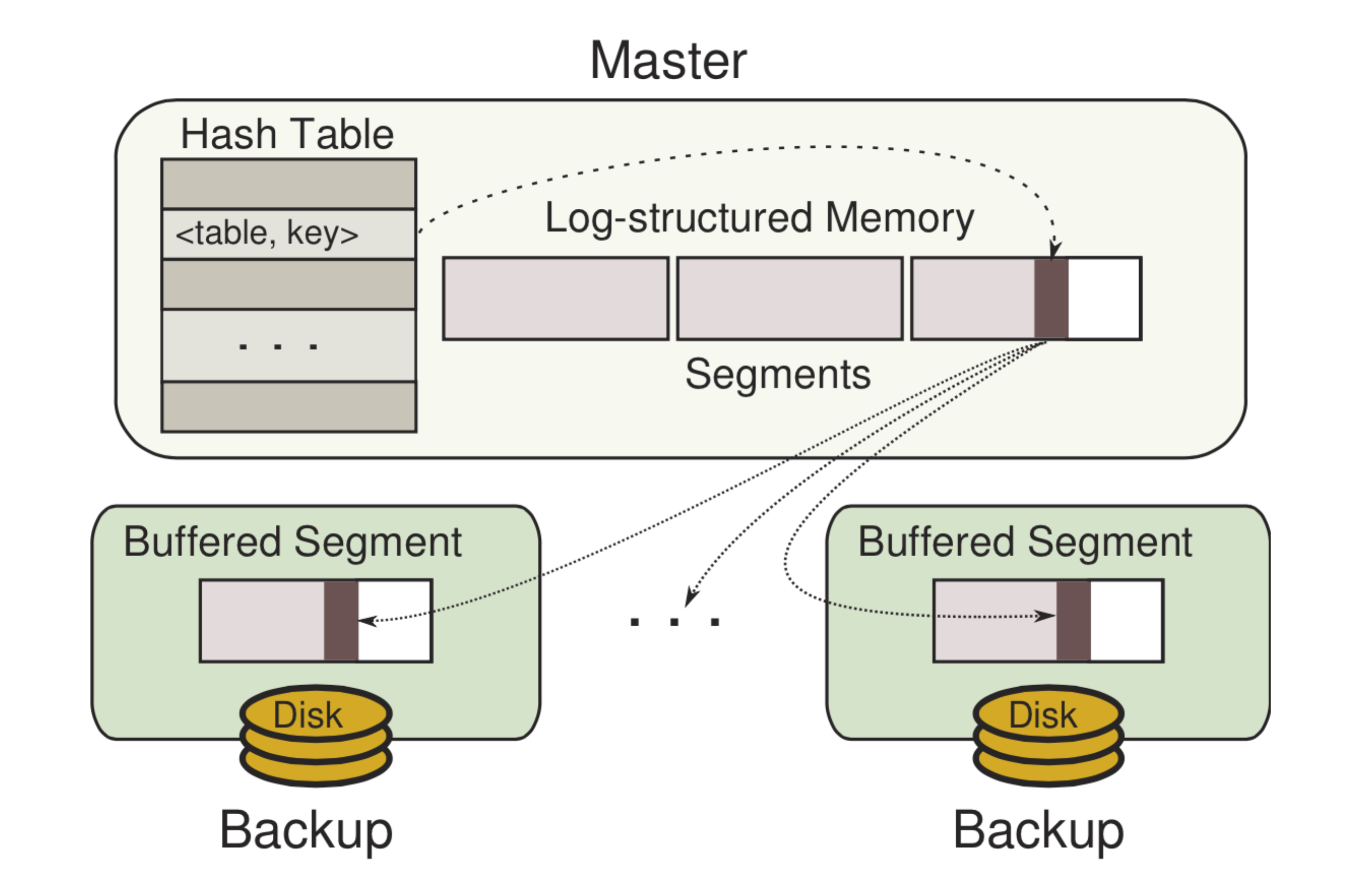


Figure 3. Master server architecture.

The collection of objects is stored in the DRAM of master’s memory. In figure 3 (Rumble, Kejriwal, & Ousterhout, 2014), all of them have a pointer in a hash table. By using hash table and key, each object could be located in DRAM very quickly. Master must keep the copy in the secondary storage, so that system would not lose data when certain server crashed. The backup data is organized as a format of log file to get maximum efficiency. Each master has its log, which is divided into 8MB pieces called segments. Each segment has its own copy in backup of other servers in cluster. Typically, we have 2-3 replicates for one segment. Although it may consumes more space in disks, as the Moore’s law the average cost of the design will be affordable and even beneficial in the future.

With an example, we could have a further understood about activities done in RAMCloud. When a master receives a write request from a client, which could be a sign up request for web application or something else, it adds the new data as a object to its DRAM. Meantime, the pointer of the object is created in hash table. Then backups copy the information of the object and append the new one at current head of segment. As soon as the backups store the data in their nonvolatile buffers, they respond to the master without issuing an I/O to secondary storage. When the master received replies from all backups, it gives a response to the client (for instance a success or error message). Since the backups will not store the segment in secondary disk until it is complete, this approach not only save the waiting time for I/O to secondary storage, but also uses the secondary storage bandwidth wisely by depositing the fragments of segments.

However, there are many challenges faced by RAMCloud and its researchers: 1) To improve the utilization rate of memory without degrading the performance of system, they used log-structure allocator to manage the objects in servers; 2) For single point of failure problem, RAMCloud use a coordinator to handle the cluster and introduce the raft algorithm; 3) By taking advantage of massive resources of cluster, RAMCloud could recover quickly after crash. In the following 3 sections, we will talk about these three techniques in detail one by one.

**Log-structured memory**

**Why Log-structure**

RAMCloud could have used traditional allocators to manage the objects in master’s memory. But these allocators will create many fragments and waste lots of resource in memory. The table 2 gives us plausible workload changes that might occur in a share storage system (Rumble, Kejriwal, & Ousterhout, 2014). Each workload has three parts, indicating the situations before, when and after delete operation of memory. Based on these workloads, figure 4 (Rumble, Kejriwal, & Ousterhout, 2014) shows the total memory needed by different allocators to support 10G of live data, which include Linux allocator glibc; non-copying allocators for speed and multiprocessor hoard (Berger, McKinley, Blumofe, & Wilson, 2000), jemalloc (Evans, 2006), tcmalloc; slab-based allocator memcached; JDK1.7 and non-copying garbage collector BehmGC. The red line in figure 4 indicates the minimum resource needed in optimal situation.

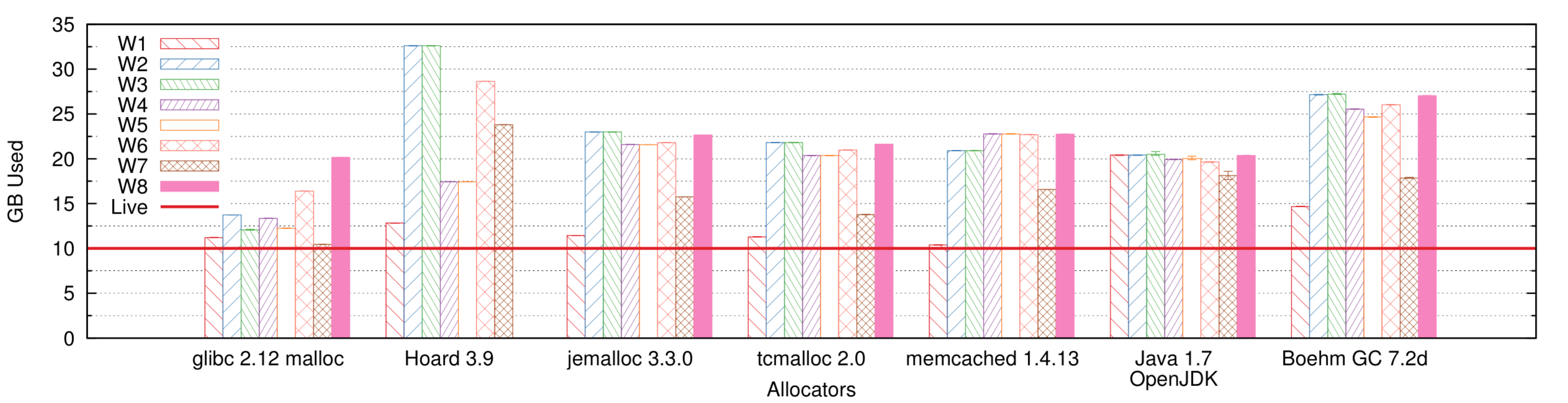


Figure 4. Memory needed by allocators in different workloads.

As shown in figure 4, the traditional allocators (non-copying allocators and copying allocators) do not work well for changing workloads. Non-copying allocators may have lots of fragments after delete operation, which wasted almost half memory. Copying allocators on the other hand may solve the fragmentation problem by moving live data to free heap space. But the cost of the operation is very expensive and grows in proportion to its scale of data.

Table 2

*Summary of Workload*

|  |  |  |  |
| --- | --- | --- | --- |
| Workload | Before | Delete | After |
| W1 | Fixed 100 Bytes | N/A | N/A |
| W2 | Fixed 100 Bytes | 0% | Fixed 130 Bytes |
| W3 | Fixed 100 Bytes | 90% | Fixed 130 Bytes |
| W4 | Uniform 100-150 Bytes | 0% | Uniform 200-250 Bytes |
| W5 | Uniform 100-150 Bytes | 90% | Uniform 200-250 Bytes |
| W6 | Uniform 100-200 Bytes | 50% | Uniform 1,000-2,000 Bytes |
| W7 | Uniform 1,000-2,000 Bytes | 90% | Uniform 1,500-2,500 Bytes |
| W8 | Uniform 50-150 Bytes | 90% | Uniform 5,000-15,000 Bytes |

**Log-structure model**

To solve the problems of traditional allocator, RAMCloud uses log-structure instead. The solution of RAMCloud is simple and obvious as the figure 5 (Rumble, Kejriwal, & Ousterhout, 2014) shows. First, we group the key-value pairs with flexible length (objects) into fixed-length segments (10mB) and use these segments as basic units. For example, 10GB may have approximately 1000 segments. New data will add to the log head of segments and each segment replicated on disks of 3 backup servers. For rapidly searching on the data in cluster, we set a global hash table to identify every key-value pair in the master servers.

We should never use the on-disk log file during normal work. The only situation it could be used is that the system crashed and needed to recover. In detail, the log files could be classified into 3 types (Rumble, Kejriwal, & Ousterhout, 2014):

1) Metadata: It consists the table ids, keys, version numbers and values of objects. During crash recovery, master reconstructs the hash table depending on the latest entities.

2) Log digest: Every new log segment has a log digest to describe the attributes of this segment. We should also use the latest log digest to load metadata when replay recovery.

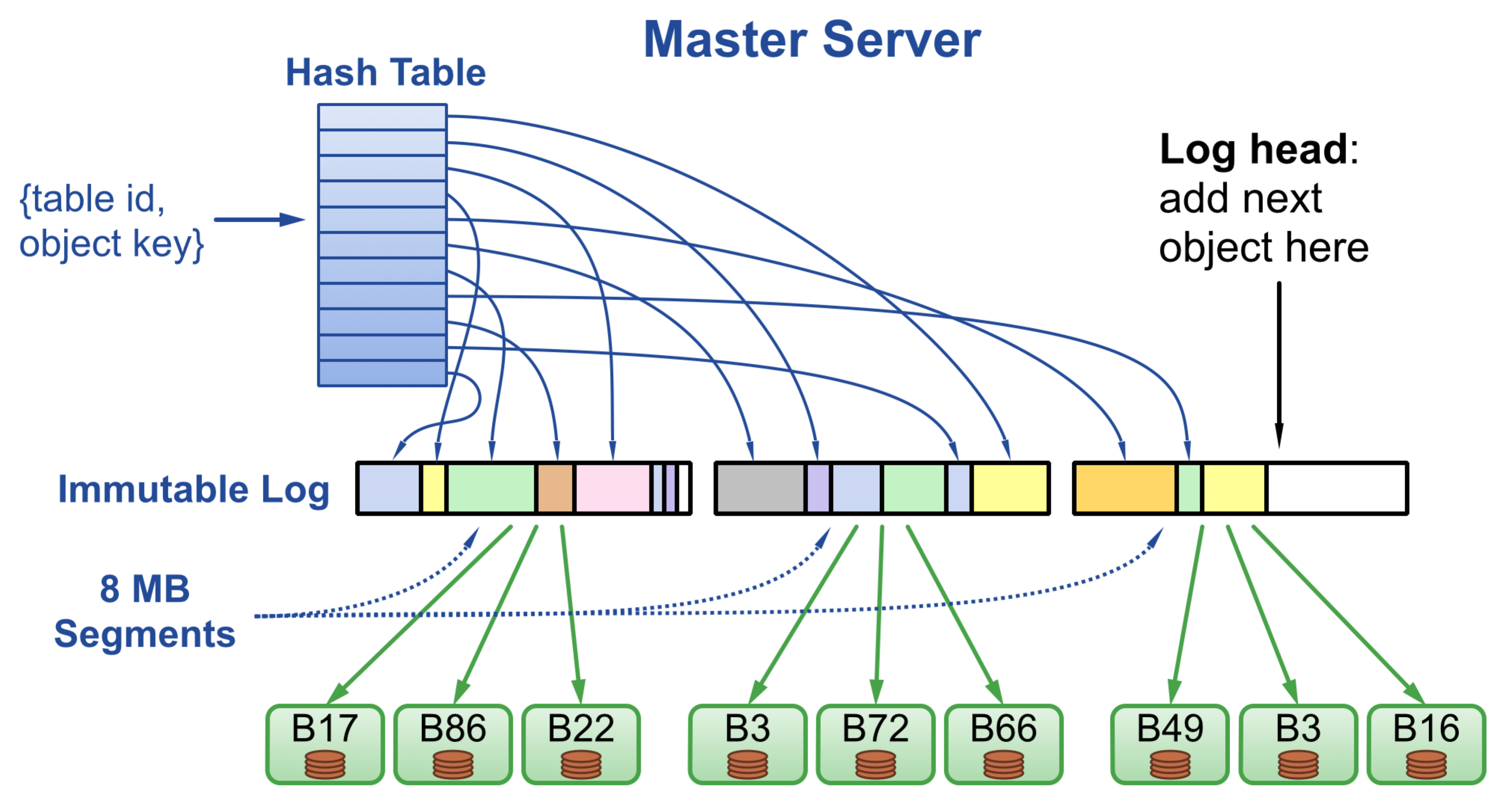
3) Tombstone: It is a special log that identifies the deleted objects (or updated objects) in segments. Unlike the previous two types, tombstone cannot be modified once it was created. 

Figure 5. Log-structure model.

When we do the deletions and updates on RAMCloud, there are lots of invalid data in the segments. This leads to two problems: how to distinguish them with valid data and how to clean garbage.

**Invalid Data Distinguish**

For the first problem, pointers in the hash table could distinguish invalid data in memory easily. In backups, RAMCloud uses tombstone to identify the invalid objects. Tombstones could keep the invalid data from reviving when RAMCloud replays recovery. When we delete or update objects, a new tombstone object, which points to the invalid data, will be added to the log head in the segment. The new objects could not be the tombstones of old objects, mainly because that RAMCloud does not have the mechanism to keep the objects order when clean garbage. For example, if we have an object A and update it with object B. Then we delete the B. Now we have 2 invalid objects in the segment, which share the same key. The order of garbage clean for these two objects is unpredictable. If B was cleaned earlier than A, A would revive when system recover. So we need a tombstone for update operation. Tombstone should be removed eventually only when all corresponding objects have been cleaned. In the next 2 sections, we will talk about the solution for the second problem.

**Two-level Cleaning**

Log cleaner is the bottleneck of performance of log-structured memory. The idea is that we should copy all valid objects to new segments and recycle the space for old segments. It is similar to the garbage collection of SSD. However, the fragment management is very expensive. The cost of cleaning grows when utilization rate increases. As figure 6 (Rumble, Kejriwal, & Ousterhout, 2014) shows that we have to move 8Bytes data to get 2Bytes space when the usage rate is 80%. If the utilization rate is 90%, we need to copy 9Bytes data for 1Byte freed.

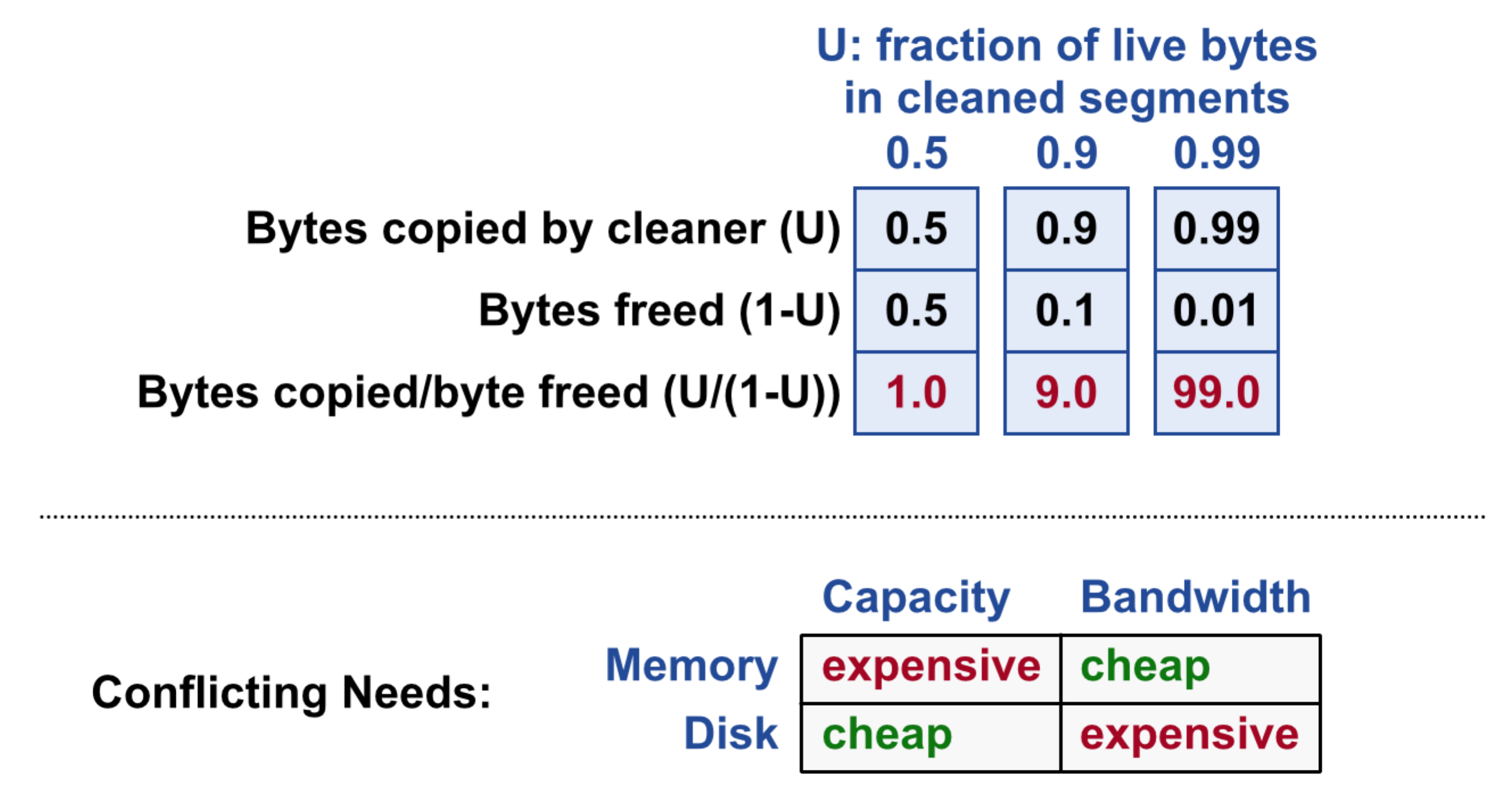
The figure 6 also indicates that there are conflicting needs between memory and disk. For memory, the bandwidth is very cheap because of DRAM’s low latency. But DRAM is expensive and we should focus on its utilization rate. In contrast, bottleneck of backups is the I/O speed of disk. The cost of bandwidth of backups is very expensive. In order to clean the master and backups independently, RAMCloud introduced a two-level cleaning strategy shown in figure 7 (Rumble, Kejriwal, & Ousterhout, 2014). By using this strategy, masters could be cleaned without reflecting the data in disk. 

Figure 6. Choice of disk and memory.

The first level cleaning is segment compaction, which only deals with in-memory segments. When execute cleaning, it compacts the segments one by one and copies the data to a new location. The id and data are unchanged in disk, so there is no I/O operation. Although segment compaction keeps the same log both in memory and disk, it may use much less space because the objects and tombstone in master and backups were removed completely right after the clean.

In absence of segment compaction, all segments would be equal to 8MB. However, segments may have different sizes with compaction. The traditional heap allocator could produce lots of fragments. Instead, RAMCloud uses the technique called seglet to solve the problem. RAMCloud divides each segment into fixed-size 64KB seglets. Since every segment may have a set of seglets, the number of the seglets then indicates the size of segment. The seglet could not eliminate the effect of fragment. Concretely, every segment may waste one half seglet on average at the end of compaction. Fortunately, with a 64KB seglet size the fragmentation represents about 1% of memory space. The other potential problem is that seglets is discontinuity because log can span seglets’ boundary. According to Rumble (2014), RAMCloud solves the problem by introducing an additional layer of software that hides the discontinuous nature of segments. The abstraction is akin to mbufs (McKusick, Bostic, Karels & Quarterman, 1996) and skbuffs (Benvenuti, 2006). With 64KB seglets, about 0.2% of 100-byte objects would be stored discontinuously.

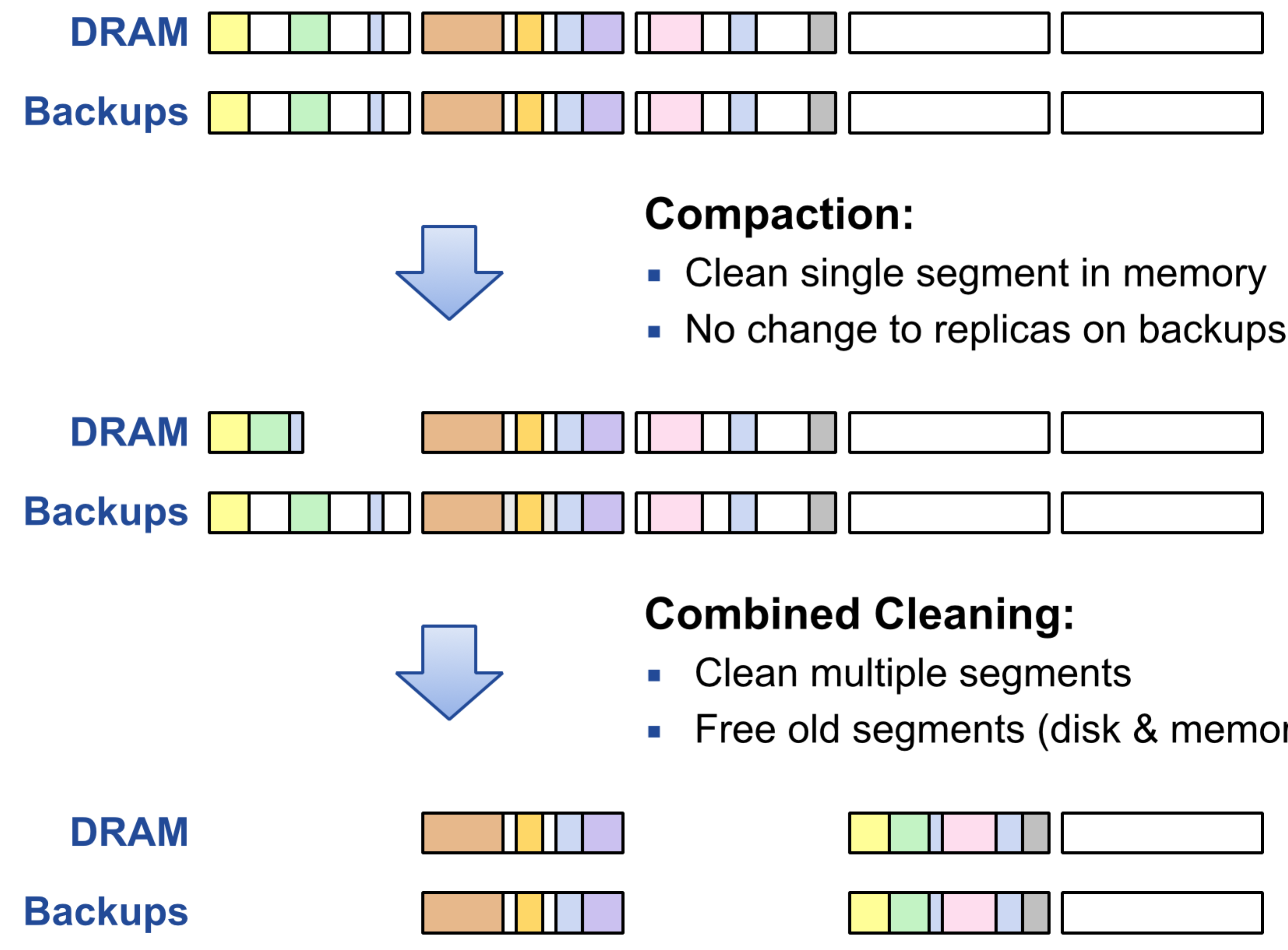


Figure 7. Compaction and combined cleaning.

The second level cleaning is combined cleaning, which frees old segments both in disk and memory. The combined cleaning combines the multiple old segments into new one and synchronizes the new data to backups. This cleaning consumes more space and time than the first one. So, in real world, we usually postpone the combined cleaning. The benefit of cleaning segment later is that the more objects deleted the lower the segment’s utilization will be. The bandwidth and space costs will drop then.

**Parallel Cleaning**

With the development of multi-core techniques, server with 16 or 32 cores CPUs is very normal in recent years. By taking advantage of multi-core CPUs, RAMCloud employs multiple cleaner threads simultaneously. As a result, it is also called parallel cleaning.

There are three reasons why parallel cleaning is easy to be used in the RAMCloud. First, log structure and meta-data are simple, which makes the cleaner thread much easier. In addition, the log is immutable once it was created. So the cleaner never need to worry about the integrity of objects when they are replicating. Finally, hash table stores the directed reference to the objects, which also makes the update straightforward. Cleaners copy the live data to new segment, update the object reference in hash table automatically and free the cleaned segment.

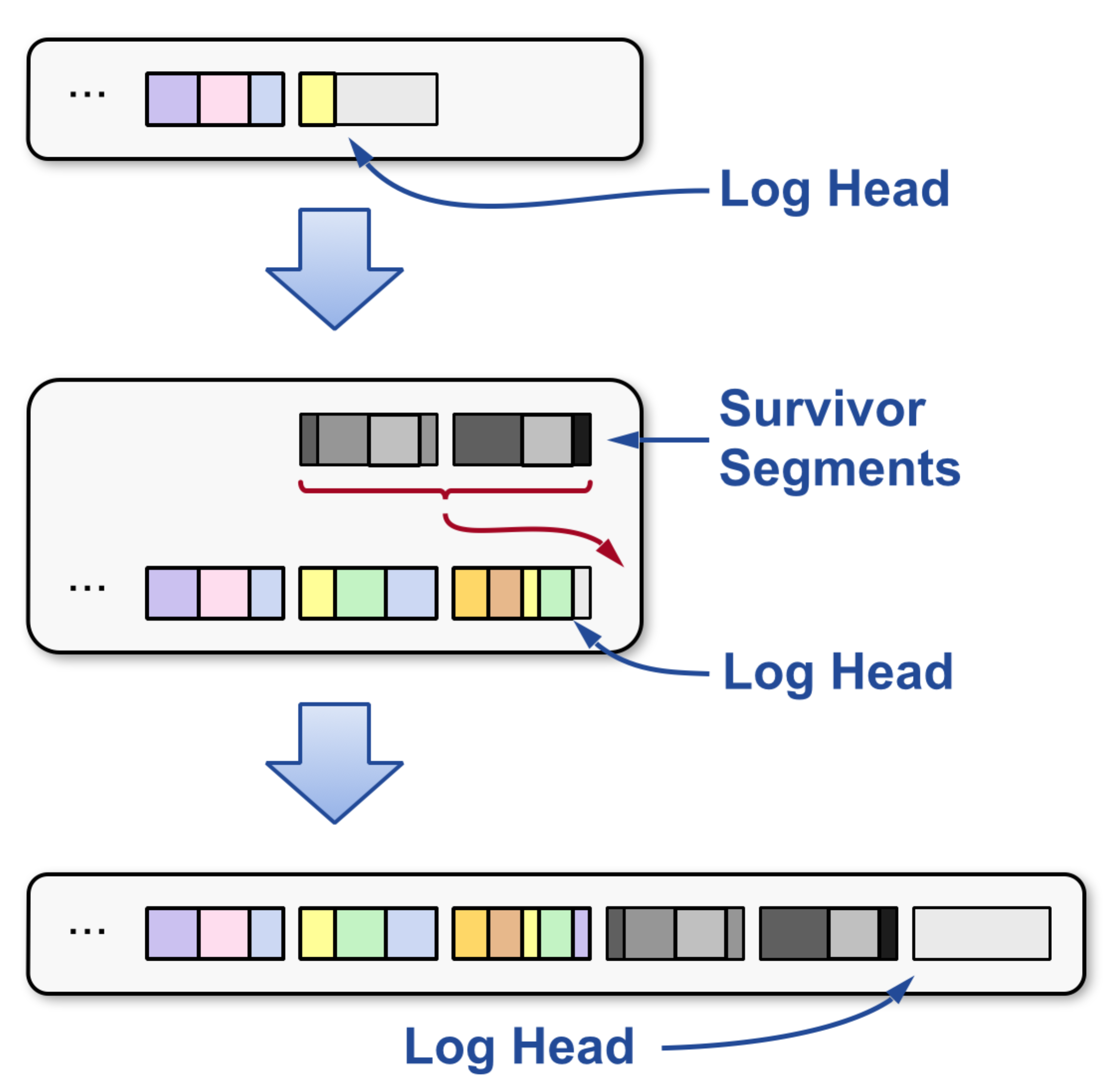


Figure 8. Parallel cleaning.

Parallel cleaning has three points of contention between service threads and cleaner threads. Ousterhout and his team also show their solutions.

First, a contention occurs when clean threads and service threads both add the data to log head. RAMCloud’s solution is to use a strategy to move all survivor data to a new segment in clean thread. In figure 8 (Rumble, Kejriwal, & Ousterhout, 2014), each cleaner thread allocates a segment as its own resource. Synchronization is required only when allocating segments. Once the allocation is done, cleaners can write the survivor data to survivor segments independently. Upon a survivor segment is completed, it will be copied to the log and released by the system. Since the survivor data are in different segments, they do the replicas simultaneously to multiple backups, which increases the throughput of master.

Second, there is a contention when cleaner and service threads compete for the hash table. Hash table identifies which object is available and the address reference to the objects in memory. Cleaner thread uses it to check whether the object is alive while service thread uses it to find objects while read and write. RAMCloud uses fine-gained locks to manage the access request: cleaner and service must acquire and release an extra lock while it access to hash table.

The final contention is that cleaner may not release segments that are used by the service thread. Once a clean thread has cleaned a segment, the space of segment in memory should be released. However, hash table is unlocked and could be access by the service thread even before the segment is released. If service used the segment at this point, it will be very hard for system to decide when to free the segment. RAMCloud provide a simple solution: free the segment only after all RPCs of service threads accomplished. It is safe to release after all service threads’ works are done because the future service threads will not use the segment again.

**Cleaning Balancer**

In the last section, we talked about two distinct cleaners in two-level cleaning. The parallel cleaning could be seen as a parallel computing version of two-level cleaning. They are essentially the same when they do single cleaning process. RAMCloud must decide when and how much of each to run memory compaction and combined cleaning. It should find a balance in order to get the best performance of write throughput out. In this section, we introduce two scheduling policies (balancers): fixed duty cycle balancer and adaptive tombstone balancer. We will talk the common logic inside two cleaners first and describe them in detail one by one.

**Common Policies Shared by Two Balancers**

There are three common logic components shared by two balancers. The first common component determines when on-disk log has grown too large. If backups are full of data, it has no choice but to do combined cleaning in order to keep backups from running out of space and maintain the recovery fast. The second determines if there are too many tombstones in the memory. If system could not free seglets from the memory by using compaction clean, a combined cleaner must be used then. Balancers used this strategy to avoid abusing combined cleaner since it is more expensive than compaction cleaning.

The third component is more complex than the previous ones. As we known, cleaning early is never cheaper than cleaning later on. The longer the system delays cleaning, the more dead objects it accumulates to be removed at one time. However, the utilization rate of memory should not be too high, because cleaning also uses memory to operate copy and move. The third component is to decide when the free space is too low and cleaning must occur. The algorithm runs as follows (Rumble, 2014): Let L be the fraction of all memory (including unallocated seglets) occupied by live objects and F be the fraction of memory in unallocated seglets. One of the cleaners will run whenever F ≤ min (0.1, (1 − L)/2). In other words, cleaning occurs if free memory has dropped to less than 10%. On one hand, cleaner will delay the cleaning to improve the efficiency; on the other hand, when predicts the risk of running out of memory, cleaner will be activated to release memory for the system.

**Fixed Duty Cycle Balancer**

The first balancer of RAMCloud is fixed duty cycle balancer. As the name shows it sets a fixed duty cycle for the combined cleaner. Its mechanism is simple: given a duty cycle D, the cleaner thread spends at most D% of the run time to do the combined cleaning; and spends the rest time to run the memory compactor. The ratio of cleaning types is fixed, but the order of them is random. For example, if we set duty cycle as 60, from 150 cleanings, we do 60% (90) combined cleaning and 40% (60) segment compaction.

The difficulty of the algorithm is to find an optimal value for combined duty cycle. As the research of Stephen (Rumble, 2014), the optimal value depends on many variables, including both the client workload and how the system is configured. For system configuration, RAMCloud takes memory utilization, object size, object lifetimes, and backup bandwidth into consideration. With a proper moderate combined duty cycle settings, it was proved that RAMCloud could do very well in most situations.

**Adaptive Tombstone Balancer**

The second balancer RAMCloud uses is called adaptive tombstone balancer. This balancer runs compaction unless too many tombstones have accumulated in memory, in which case it runs the combined cleaner. It considers the segment compaction in high priority because it is more efficient than combined cleaning. But RAMCloud must use combined cleaner in two situations. First situation is that there are too many tombstones. The compaction cannot delete the tombstones in the memory without storing them in backups. Since the utilization rate climbs with the number of tombstones in the master’s memory, the efficiency of compaction will drop. Eventually the combined cleaner is cheaper than segment compaction to remove the tombstone.

RAMCloud also has an algorithm to decide when too many tombstones have accumulated. Let T be the fraction of memory occupied by live tombstones, and L be the fraction consumed by live objects. Then (1 - L) represents the sum of fractions for all tombstones, dead objects, and unallocated space. There is said to be too many tombstones only when T / (1 − L) ≥ max Tombstone Ratio% (Rumble, 2014). The left side of the formula calculates how many proportion of the free space is owned by tombstone. The tombstone Ratio here indicates the optimal value to make the system works well. After the experiments on multiple workloads, researchers (Rumble, Kejriwal, & Ousterhout, 2014) found that 40% is a fair value. As the formula shown above, the combined cleaner seems to be used more frequently when there are many small-scale files.

**Comparison of Balancers**

To compare the above two balancers, 12 worst experiments with extreme situations were done on the 40% fixed duty cycle and 40% tombstone ratio balancers. Most of the tests used small objects, low access locality, high speed backup I/O and high memory utilization. As shown in the table 2 (Rumble, 2014), the tombstone balancer in these cases has 6.3% penalty, which is only half of the fixed duty balancer’s.

Comparison was also done in optimal situations. As shown in the table 3 (Rumble, 2014), both balancers do well in optimal situations. The adaptive tombstone balancer has slightly advantage than the fixed duty balance with higher average (95.1%) and maximum (109.3%) in the table.

Table 3

*Comparison of fixed balancer and tomb balancer.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Balancer** | **Avg** | **Min** | **Max** | **Stddev** |
| **Fixed 40%** | 93.40% | 63.60% | 100.00% | 8.50% |
| **Tomb 40%** | 95.10% | 63.40% | 109.30% | 8.30% |

Table 4

*12 Extreme Situations Tested for Balancers.*

|  |  |  |  |
| --- | --- | --- | --- |
| **Experiments with Throughput ≤ 80% of Optimal** | **Fixed 40** | **Tomb 40** | **Combined** |
| **Overall** | 18 (12.5%) | 9 (6.3%) | 27 (9.4%) |
| **With 90% Memory Utilisation** | 18 (12.5%) | 7 (4.9%) | 25 (8.7%) |
| **With 70% Memory Utilisation** | 0 (0%) | 2 (1.4%) | 2 (0.7%) |
| **With 100 MB/s Backup Bandwidth** | 10 (6.9%) | 5 (3.5%) | 15 (5.2%) |
| **With 250 MB/s Backup Bandwidth** | 3 (2.1%) | 3 (2.1%) | 6 (2.1%) |
| **With 500 MB/s Backup Bandwidth** | 5 (3.5%) | 1 (0.7%) | 6 (2.1%) |
| **With 1 GB/s Backup Bandwidth** | 7 (4.9%) | 2 (1.4%) | 9 (3.2%) |
| **With 100-byte Objects** | 6 (4.2%) | 6 (4.2%) | 12 (4.2%) |
| **With 1000-byte Objects** | 5 (3.5%) | 3 (2.1%) | 8 (2.8%) |
| **With 10000-byte Objects** | 7 (4.9%) | 0 (0%) | 7 (2.4%) |
| **With Zipfian Accesses** | 3 (2.1%) | 2 (1.4%) | 5 (1.7%) |
| **With Hot-and-Cold Accesses** | 7 (4.9%) | 2 (1.4%) | 9 (3.2%) |
| **With Uniform Accesses** | 8 (5.6%) | 5 (3.5%) | 12 (4.2%) |

As a result, the adaptive tombstone balancer with a 40% tombstone ratio may provide a better performance within all experiments when compared to a 40% fixed duty cycle balancer. Although we cannot guarantee that 40% is an optimal value for fixed balancer, it seems that tombstone balancer does have an advantage in extreme situations. So RAMCloud uses it as a default balancer currently. However, with the progress of hardware and research for the optimal value, the adaptive balancer may play much better than the tombstone. Since the adaptive balancer is very sensitive to the duty cycle, dynamic duty cycle based on workloads, which is normally used in networks (Lin, Qiao, & Wang, 2004), may help to solve the problem.

**Coordinator and Raft Algorithm**

In the section 2, we talked about the coordinators, which manage the servers’ distribution in the RAMCloud system. In coordinator, we used algorithms to make the cluster of servers work as a coherent group and survive the failures of their member. These consensus algorithms play a key role in large-scale software systems. In this section, we will talk about a consensus algorithm Raft in RAMCloud and why we choose it.

**Why Raft?**

There are many consensus algorithms having similar functions with Raft, such as Paxos (Lamport, 2001), Oki and Liskov’s Viewstamped Replication (Oki & Liskov, 1988) and so on. But Raft has its own novel features make it superior to some of these algorithms (Ongaro & Ousterhout, 2014): First, Raft uses a stronger form of leadership than other consensus algorithms. Secondly, it resolves the conflicts very quickly by using randomized timer to elect leaders. Finally, Raft has a mechanism for changing the set of servers in the cluster, which allows the cluster to continue operating normally during configuration changes.

Although Paxos ensures the above features, it has two significant drawbacks. The first problem is that Paxos is hard to understand. Not only the students but also the professional researchers have obstacle to fully understand the algorithm. The second problem is that Paxos’ feasibility is not strong. The original designer Lamport’s descriptions of Paxos aimed mainly at single-decree Paxos and failed to give a proper extension to multi-Paxos. The followers’ versions (Mazieres, 2007; Kirsch & Amir, 2008; Renesse, 2011) to optimize Paxos are different from each other and even conflict the Lamport’s sketches. As the result, RAMCloud uses Raft as its default consensus algorithm.

**Overview of Raft**

Raft is a protocol for implementing distributed consensus. Typically, a Raft cluster may have 5 servers, which makes3 it to tolerate 2 failures. Each node can be in one of three states (Howard, 2014): the follower state, the candidate state or the leader state. The follower state is passive: it votes to the leader state and synchronizes the data with the leader. The second state, leader, is connected to clients (directly or indirectly) and sends data to nodes in other states. The candidate state is used to elect a new leader. Figure 9 (Ongaro & Ousterhout, 2014) shows the transitions of the three states and we will discuss it below.

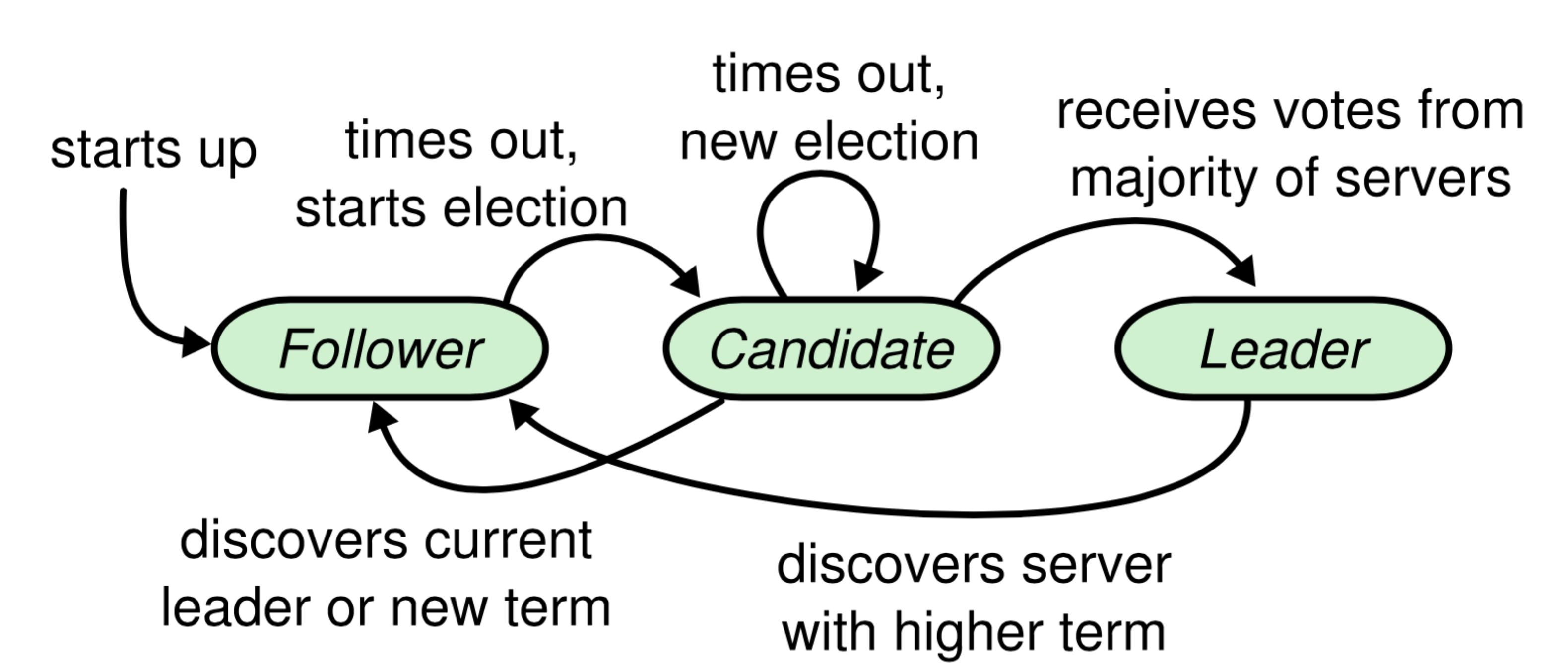


Figure 9. Raft algorithm model.

The basic sequence of Raft algorithm has three processes. The first process is called leader election. All our nodes start in the follower state. If followers don’t hear from a leader then they can become a candidate. The candidate then requests votes from other nodes. Nodes in follower state will reply with their vote. The candidate becomes the leader if it gets vote from a majority of nodes. All changes to the system now go through the leader. Each change is added as an entry in its log. This log entry is currently uncommitted so the leader’s value keeps unchanged. To commit the entry, the node first replicates it to the follower nodes. Then the leader waits until a majority of nodes have written the entry. The entry is now committed on the leader node and the node’s value is updated. Then the leader notifies all followers that the entry is committed. This process is called log replication.

**Leader Election**

There is two timeout settings used in leader election. First is the election timeout, which represents the amount of time a follower waits until becoming a candidate. After the election timeout the follower becomes a candidate and starts a new election votes for itself and sends out Request Vote message to other nodes in the cluster. RAMCloud uses random election timeout picked between 150ms and 300ms (Ongaro & Ousterhout, 2014). This preserves servers from becoming candidates at the same time. If the receiving node hasn’t voted yet then it votes for the candidate and resets its election timeout. As we known, the candidate has a majority of votes becomes leader. In this term, requiring a majority of votes guarantees that only one leader can be elected for each time.

The second timeout is called heartbeat timeout. Once a candidate converts to a leader, it sends out Append Entries messages to the followers in intervals specified by this timeout. Followers then respond to each Append Entries message to claim the authority of leader. This term will continue until a follower becomes a candidate. If one leader crashed, a new leader election will begin.

Normally there are no nodes become candidates at the same time because of randomized election timeout. However, if two nodes do become candidates at the same time then a split vote can occur. This could be caused by the same election timeout or latency of networks. Raft has a special restriction for this situation. If two nodes become candidates and receive the equal votes, they need to compete an election timeout for a new round. Since the timeout is randomized, the probability of the conflicts will decrease exponentially.

**Log Replication**

Once a node is chosen to be the leader, we need to replicate all changes to our system to all nodes. Raft does this by applying the same Append Entries that was used in heartbeats. Upon the arrival of new change, it is appended to the leader’s log and sent to the followers on the next heartbeat. An entry is committed once a majority of followers acknowledge it. Client will get a response from leader to flag the success of communication.

During the log replication, Raft also faces the network partition problem. If we have a partition in the networks, the leader with fewer followers will stay uncommitted because it cannot replicate to a majority. Raft heals the network partition by using another special restriction. The leader with fewer followers will see higher election term and step down (Howard, 2014). In other words, this node and its followers will roll back their uncommitted entries and match the new leader’s log. Then the log in the successful leader is now consistent across the whole cluster. This is very helpful when we merge two RAMClouds or add new servers to existing RAMCloud.

**Interaction with Clients**

In the overview part of this section, we stated that the clients only communicated with leaders either directly or indirectly. Raft does provide a mechanism for new clients to find their leader servers. When a new client tries to connect the cluster, it first picks a server randomly. If the server is a leader, the connection accomplishes. Otherwise, the server will reject the request of clients with a timeout and send the latest leader information to the clients.

**RAMCloud Fast Recovery**

When a RAMCloud server crashes, the objects in master must be reconstructed by replaying the log. The crashed master’s data will cause a stall in RAMCloud and be unavailable until the hash table has been reconstructed. If the recovery is fast enough, where 1-2 second is a fair number (Ongaro, Rmuble, Stutsman, Ousterhout & Rosenblum, 2011), it will constitute continuous availability for most applications. In this section, we will show the way RAMCloud detects server failures and the three basic steps to recover: setup, replay and cleanup.

**Failure Detection**

RAMCloud has two ways to detect server failures. First, if there are clients activated in the cluster, any timeout failure for remote procedure call will be sent to coordinator. Second, even if in absence of clients, RAMCloud also scans and checks failures by itself. Each RAMCloud server periodically broadcasts a ping RPC to neighbor servers and attempt to access to the data. The choice of neighbor servers is normally random, which offloads the work on networks. Once a failure is detected in RAMCloud, it must be reported to the coordinator. Then coordinator arranges a communication with the server. If it is still not working, recovery flow will be activated.

**Setup**

As the first step of recovery flow, the goal of setup is to find all replicas of log segments belonging to the crashed master and prepare the recovery masters for replay.

At the beginning of setup, RAMCloud needs to find all log segment replicas in the backup. Since there is no centralized map of replicas in coordinator, all backup need to query the crashed replicas in their own servers. Each backup server should send a list of replicas owned by crashed master to coordinator. The coordinator reconstructs the lists and makes a new map for these replicas.

Once the new map is accomplished, coordinator must determine whether the map equals to the entire log of the crashed master. RAMCloud detects the completeness of the log by using log digest information stored in each segment. The log digest is very small, which is less than 1% storage overhead even when uncompressed (Ongaro, Rmuble, Stutsman, Ousterhout & Rosenblum, 2011). Since every segment has a log digest and backups have multiple copies for the same crashed replicas, coordinator set a mechanism to use the newest log digest when detect the incompleteness.

Finally the coordinator should select recovery masters and assign each of them a partition to recover. The choice of partitions for a master had already been made before it crashed. The information of the choice is called will, which is periodically uploaded by master to the coordinator.

**Replay**

The second step of recovery is replay. It fetches log segments from different backups and reconstructs the hash table in the new recovery master, which consumes the vast majority of time in recover.

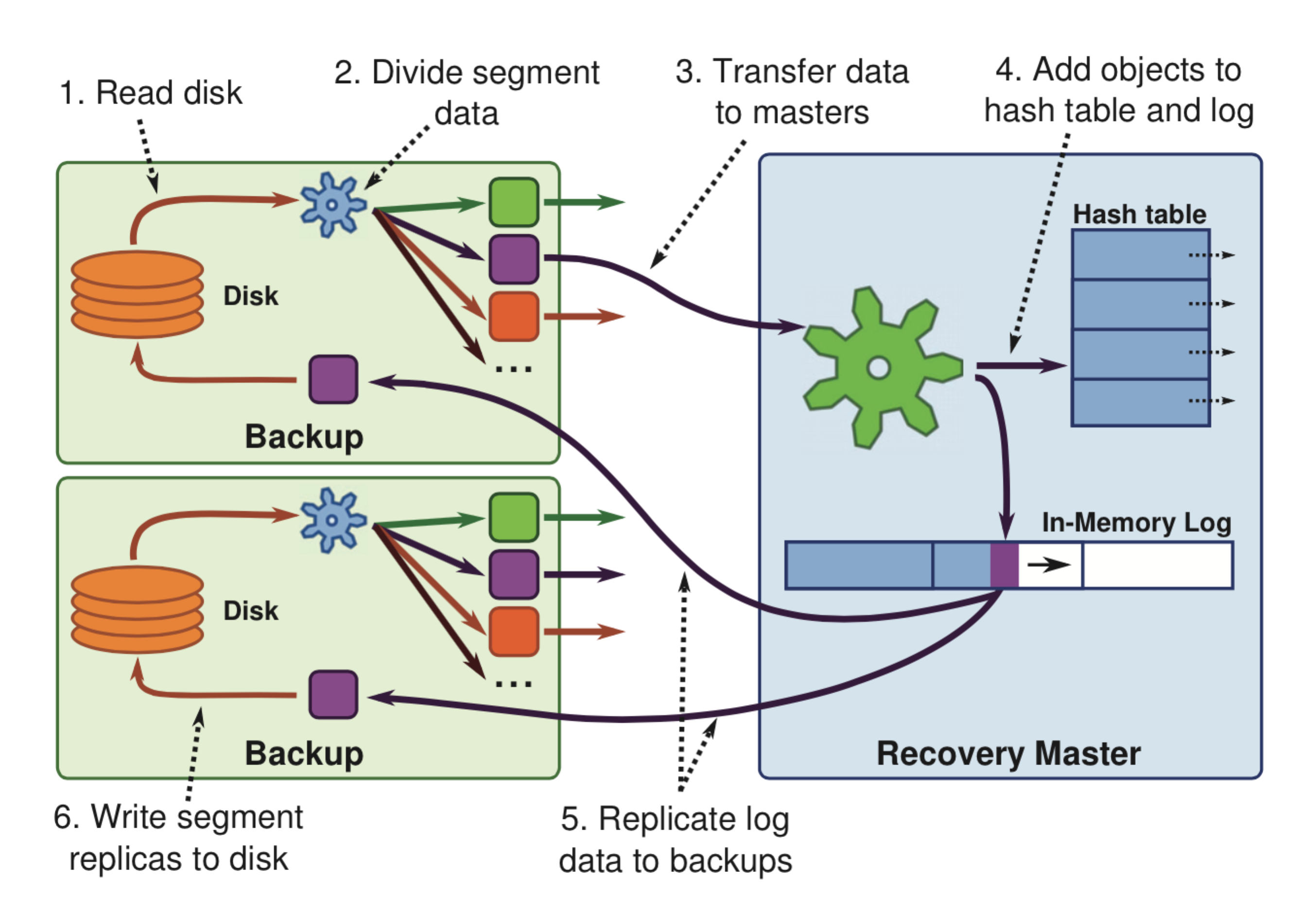


Figure 10. RAMCloud recovery model.

Concretely, there are six stages during replay. As shown in the figure 10(Ongaro, Rmuble, Stutsman, Ousterhout & Rosenblum, 2011), the segment is first read from disk into the memory of a backup. Then the backup divides the records in the segment into separate groups. These data are transferred over networks to the recovery master for its own partition. The master adds the data to its hash table and log. After the master recovers all its segments, it creates new backup replicas just like normal operation. Finally, backup writes segment replicas to disk.

To improve the performance, RAMCloud employs two mechanisms. The first one is data parallelism, which asks backups to read different segments from disk and recovery masters to reconstruct different partitions all in parallel. The second is pipelining: all six stages above should proceed in parallel. For example, when a segment is being transferred over networks, another segment is being read from disk on backups. Similar pipelining occurs on all of resources of cluster, including CPUs, masters, disks, networks and so on. The greatest problem of this technique is the pipeline stalls. The segment replay order in pipelining is very important. If a recovery master requests the segment that has not been ready in backup, then the master will stall. RAMCloud solves the problem by asking each backup decides the segment order in advance and sent the information to coordinator during setup. Each master uses this information to predict which segment data is likely to be loaded.

**Cleanup**

Once a recovery master’s recovery accomplishes, it will send a notification to coordinator. The coordinator modifies the configuration of this master server to indicate the node is available now. If the failure was detected by client requests, coordinator informs the client that crashed data is now recovery. After all recovery masters complete their works, the coordinator notifies all backups to free the space for crashed segments. At this moment, the recovery is done.

**Conclusion**

In this paper, we have discussed about the architecture of RAMCloud. The RAMCloud structure includes log structure and coordinator. We showed the detail of log structure and the balancers used to choose the cleanings in it. We also talked about the consensus algorithm used in coordinator and described the basic steps of fast recovery when the master crashed.

It is obvious that RAMCloud will have great effects on the IT industry. RAMCloud simplify the design of large-scale web applications and make it flexible for the developers. In addition, RAMCloud is conducive to the progress of cloud computing for its low latency. Finally, the wide use of RAMCloud will promote the growth of battery management system, server cluster and DRAM.

However, there are still some disadvantages in RAMCloud. First of all, RAMCloud is too expensive, which 50-100 times disk system and 5-10 times flash system. As the blog of Demirbas (2010), there are striking trends that paper ignores to mention that while disk has $0.07 cost/GB, the RAM has $60 cost/GB. Jeff Darcy (2010) posts critiques of RAMCloud. His main point is that you cannot shoehorn everything in one system or in RDBMS and applications need many different kinds of storage. The second shortcoming comes from low-latency RPC. Although there are high-speed networks such as Infiniband, Myrinet, Arista 7500, the normal datacenters still depend on Ethernet, which has latency for 5-10μs. We should improve the hardware to get the idea design of Ousterhout (1μs).

As the result, the future project should focus on large-scale low-latency RPC, higher-level data model, cluster management, tail latency, notification and memory hierarchy.

**References**

1. Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., ... & Stutsman, R. (2010). The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, *43*(4), 92-105.
2. Gray, J., & Putzolu, F. (1987, December). The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *ACM SIGMOD Record* (Vol. 16, No. 3, pp. 395-398). ACM.
3. Fitzpatrick, B. (2004). Distributed caching with memcached. *Linux journal*, *2004*(124), 5.
4. Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113.
5. Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (pp. 1-10). IEEE.
6. Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, *39*(4), 12-27.
7. Rumble, S. M., Kejriwal, A., & Ousterhout, J. K. (2014, February). Log-structured memory for DRAM-based storage. In *FAST* (Vol. 1, p. 16).
8. Evans, J. (2006, April). A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDCan Conference, Ottawa, Canada*.
9. Berger, E. D., McKinley, K. S., Blumofe, R. D., & Wilson, P. R. (2000). Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, *35*(11), 117-128.
10. Rumble, S. M., Kejriwal, A., & Ousterhout, J. K. (2014, February). Log-Structured Memory for DRAM-Based Storage [PowerPoint slides]. Retrieved from https://ramcloud.atlassian.net/wiki/download/attachments/6848659/LSM%20FAST%20%28Feb%29.pptx?version=1&modificationDate=1394322427921&api=v2
11. Rumble, S. M. (2014). *Memory and object management in RAMCloud* (Doctoral dissertation, Stanford University).
12. McKusick, M. K., Bostic, K., Karels, M. J., & Quarterman, J. S. (1996). *The design and implementation of the 4.4 BSD operating system*. Pearson Education.
13. Benvenuti, C. (2006). *Understanding Linux network internals*. " O'Reilly Media, Inc.".
14. Lin, P., Qiao, C., & Wang, X. (2004, March). Medium access control with a dynamic duty cycle for sensor networks. In *Wireless Communications and Networking Conference, 2004. WCNC. 2004 IEEE* (Vol. 3, pp. 1534-1539). IEEE.
15. Ongaro, D., & Ousterhout, J. (2014, June). In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference* (pp. 305-320).
16. Lamport, L. (2001). Paxos made simple. *ACM Sigact News*, *32*(4), 18-25.
17. Oki, B. M., & Liskov, B. H. (1988, January). Viewstamped replication: A new primary copy method to support highly-available distributed systems. In*Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (pp. 8-17). ACM.
18. Mazieres, D. (2007). Paxos made practical. Retrieved from http://www.scs.stanford.edu/˜dm/home/papers/paxos.pdf
19. Kirsch, J., & Amir, Y. (2008). Paxos for system builders. *Dept. of CS, Johns Hopkins University, Tech. Rep*.
20. Van Renesse, R. (2011). Paxos made moderately complex. *Cited on*.
21. Howard, H. (2014). ARC: analysis of Raft consensus. *Technical Report UCAM-CL-TR-857*.
22. Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., & Rosenblum, M. (2011, October). Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (pp. 29-41). ACM.
23. Demirbas, M. (2010). Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. Retrieved from http://muratbuffalo.blogspot.com/2010/12/case-for-ramclouds-scalable-high.html
24. Darcy, J. (2010). In Memory Data Grids. Retrieved from http://pl.atyp.us/wordpress/?p=2695