THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUPPORTING TAIL LATENCY SLOS IN CEPH

SHAOBO GUAN
SPRING 2020

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

Timothy Zhu
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Ting He
Associate Professor of Computer Science and Engineering
Honors Adviser

* Electronic approvals are on file.

ABSTRACT

When optimizing the performance of a storage system, people often focus on reducing the average latency. Nevertheless, when the behavior of a workload exhibits burstiness, a traditional server that optimizes the average latency potentially generates a high tail latency (i.e., the time the server takes to serve some of the slowest requests). To ensure a consistent performance across almost all users, companies often desire a low and bounded tail latency. The objective of the thesis is to design a new system for meeting tail latency performance goals, known as Service Level Objectives (SLOs), in the context of distributed storage.

Prior work has demonstrated that analytical modeling approaches can help in configuring systems to meet tail latency SLOs in networks, hard drives, and SSDs, but these approaches have not been applied yet in distributed storage. These approaches are hard to implement in distributed storage because clients communicate with many servers, which leads to unpredictable interference patterns. This thesis takes a first step in analytically modeling distributed storage for meeting tail latency SLOs.

To evaluate our distributed storage model, we use Ceph, an open-source distributed system, as the proving ground. Our preliminary results show that our model can integrate with prior work to meet tail latency SLOs in distributed storage while efficiently using the storage resources.

TABLE OF CONTENTS

LIST OF FIGURES iii

LIST OF TABLES iv

ACKNOWLEDGEMENTS v

Chapter 1 Introduction 1

Chapter 2 Core Techniques of Ceph 4

- 2.1 Services of Ceph 4
- 2.2 Architectures of RADOS 5
- 2.3 CRUSH Algorithm 6
- 2.4 BlueStore 10
- 2.5 Messaging 11

Chapter 3 Basic Model for Rate-limiting and Priority 14

- 3.1 Overview 14
- 3.2 Challenges 14
- 3.3 How Bucket Model and Priority Work Together 15

Chapter 4 Design and Implementation 17

- 4.1 Scheduler 17
- 4.2 Profiler 18
- 4.3 Estimator 18
- 4.4 Trace Replay 20
- 4.5 Modified Shards 20

Chapter 5 Experimental Results 23

- 5.1 Setup 23
- 5.2 Benefits of Priority and Rate-Limiting 23
- 5.3 Benefits of Distributed Storage Modeling 26

Chapter 6 Related Work 31

Chapter 7 Future Work 32

- 7.1 Network Rate-limiting 32
- 7.2 CRUSH Rule Modification 32

Appendix A Algorithm for Scheduling Requests 33

BIBLIOGRAPHY 34

LIST OF FIGURES

Figure 1. Services of Ceph [10] 4

Figure 2. A Detailed Look of RADOS [10] 6

Figure 3. CRUSH map [13] 8

Figure 4. Comparison between FileStore and BlueStore [3] 10

Figure 5. Message structure 12

Figure 6. Message handling flowchart. 13

Figure 7. Shard structure in Ceph 22

Figure 8. Modified shard 22

Figure 9. Overloading the server with neither rate-limiting nor priority 24

Figure 10. Overloading the server without rate-limiting 24

Figure 11. Burst client 1 without rate-limiting 25

Figure 12. Burst client 1 with rate-limiting 25

Figure 13. Distributed estimator with unbalanced traces 27

Figure 14. Optimistic estimator with unbalanced traces 27

Figure 15. Pessimistic estimator with unbalanced traces 27

Figure 16. Distributed estimator with balanced traces 28

Figure 17. Optimistic estimator with balanced traces 28

Figure 18. Pessimistic estimator with balanced traces 28

LIST OF TABLES

Table 1. Comparison among 3 models..29

ACKNOWLEDGEMENTS

I want to express my gratitude to people who selflessly gave me so much help and suggestions that contribute to my success at Penn State. Firstly, I would like to show my appreciation to my honor advisor, Dr. Ting He. She gave me a lot of useful advice that helps me to make academic decisions with more confidence. Also, I want to thank my colleague, Yiyan Wei, who works on a topic that has a similar background to mine. He offered me a lot of technical support for setting up the testing environment. I would also thank my families for their care.

Foremost, I want to express my genuine gratefulness to my thesis supervisor, Dr. Timothy Zhu. What he has devoted to me is far beyond the obligation as an honor thesis supervisor. He is the most brilliant person I have ever worked with. He taught me not only the solution to solve a question, but also a lot of methodology for solving a type of questions. Also, he is so kind and patient. I am surprised that a busy person like him is still willing to hold a weekly meeting with me; even sometimes, he needs to explain fundamental concepts or to help me figure out messy set-up issues. Honestly, the skills and knowledge I learned from him are much more than those I learned from classes.

Lastly, I would like to thank the Schreyer Honor Program offered by Penn State. It gives me more chances to talk with the CSE faculties outsides of the classroom.

Chapter 1

Introduction

Meeting tail latency Service Level Objectives (SLOs) is an important topic in the field of cloud computing and distributed network storage [15, 16, 17]. SLOs refer to the desired performance goals when using services, such as cloud services and datacenter applications. Tail latency SLOs are a type of SLO for the tail latency metric, which represent the slowest requests (e.g., 99% slowest request). For cloud computing platforms like Google Cloud and Amazon Web Services, their customers are used to the instant response and low latency, and may not tolerate high tail latencies even if they occur rarely. To ensure consistent performance for all requests, practitioners and researchers have devoted significant effort in designing systems for managing tail latency performance [4, 7, 11, 14, 15, 16, 17]. In this thesis, we consider one such system: a distributed storage system.

Distributed storage systems are an important building block in many systems such as Amazon's Elastic Block Store (EBS), enterprise Network File System (NFS) storage, etc. As storage is often a bottleneck in systems, it is critical to ensure low and bounded tail latency in such systems. In particular, when some workloads are in a burst period, the queueing system embedded in the shared storage system will also hurt the performance of other workloads that behaves normally if there is no any protective mechanism. And the burstiness is hard to model and control since we do not know when it starts, how long and how intense it is before it truly occurs for different workloads in the real world.

Meeting tail latency SLOs in distributed storage systems is particularly challenging since the workloads using the system often interfere with each other, causing congestion and high tail latency. Furthermore, the performance of storage is often non-linear and hard to predict, and the distributed nature of these systems make it hard to control. Research has shown that reactive approaches that try to adapt system parameters to workload changes are unable to cope with bursty workloads [15]. Other research has investigated using modeling and analytic approaches for controlling tail latency [7, 15, 16, 17], but none of these consider the distributed nature of storage systems with clients communicating with multiple storage servers simultaneously.

The thesis investigates how to meet tail latency SLOs in the Ceph distributed storage system. Ceph is a popular open-source distributed storage system that is widely used in large scale datacenters. It is a typical object-based distributed system and even with higher scalability and efficiency than other object-based systems such as zFS [9], Lustre [2] and the Panasas file system [12]. Moreover, many state-of-the-art techniques, such as CRUSH algorithm [13] and BlueStore [1] have been designed and implemented on Ceph.

To meet tail latency SLOs in Ceph, this work builds upon the prior PriorityMeister/SNC-Meister/WorkloadCompactor line of work that takes an analytic modeling approach to controlling tail latency. This line of work only focuses on networked storage, while our work considers distributed storage. The key difference is that in a distributed system, each client is no longer served by a single server node. Instead, how the traffic of a workload is distributed to different servers and how it interferes with the traffic of other workloads are new problems that have not been considered in the prior work. Hence, the prior analytic modeling approach which regards the server as a single storage piece cannot characterize a distributed storage system

accurately. To enable this approach to work with Ceph, we needed to implement two key components.

Chapter 2

Core Techniques of Ceph

2.1 Services of Ceph

RADOS (Reliable Autonomic Distributed Object Store) is a fundamental part of Ceph. In other words, all the services offered by Ceph are some extra layers building upon RADOS. RADOS consists of many service nodes that have different functionalities, and the whole server is also known as a ***cluster.*** As figure 1 shows, the layer built on RADOS is called LIBRADOS, which offers an interface for users or applications to access the cluster directly. Another worth-mentioning thing of Ceph is that with additional metadata server nodes, it is able to virtualize as a file system.

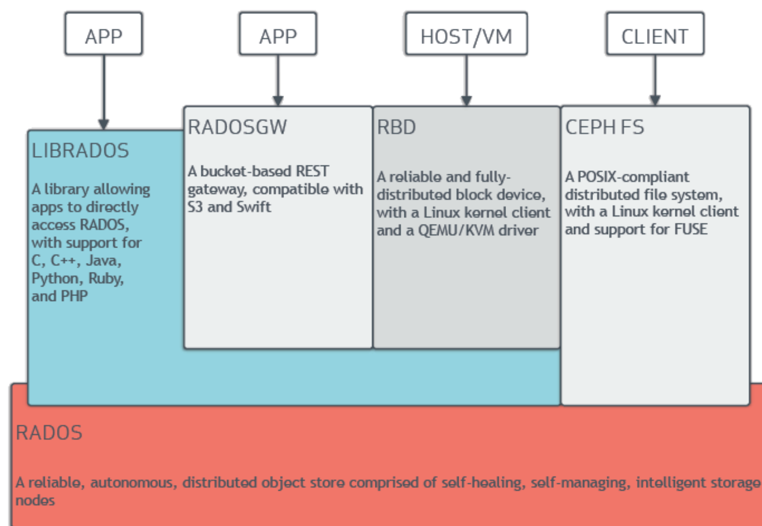

Figure 1. Services of Ceph [10]

2.2 Architectures of RADOS

As Figure 2 shows, a Ceph cluster is comprised of many computers in which different daemons are running for various purposes. Theoretically, a single node could run any number and any daemons. However, each node should only work as one type of daemon for the best performance. By deciphering the meaning of icons in Figure 2, this section will give a brief introduction of all kinds of daemons in a typical cluster.

The blue squares with a red strip represent OSDs (object storage daemons) in a cluster. The primary task of OSD is to manage the local storage resources. Although OSD cannot determine placement policies, followed by rules in a configuration file, OSDs can talk with each other and handle replication or recovery tasks. Our research mainly focuses on applying new techniques to the internal mechanisms of OSDs. More details about how OSDs manipulate data and process requests will be demonstrated later.

The squares containing a character M represent MONs (monitors). Inferred by the name, monitors are responsible for tasks about supervision. However, different from a traditional “monitor” concept in many distributed systems that need a central monitor to record and manage metadata, monitors in CEPH are more like policy deciders or information postmen that help OSDs to communicate with each other. The number of monitors in a cluster should always be odd, which ensures monitors could use the “minority obeys majority” policy when deciding something collectively. Furthermore, after Ceph 12.0, a daemon that is not shown in Figure 2, called MGR (manager), is mandatory to run along with monitors for a healthy cluster environment.

The squares that contain a branch-like icon represent MDSs (metadata servers), which is only needed if the clients want to use the file system services of Ceph. One impressive design of the MDS is that a cluster is encouraged to have multiple metadata servers, and they will work as nodes in a tree structure. In other words, each MDS can manage the metadata of a fraction of all nodes and achieve excellent performance by distributing tasks.

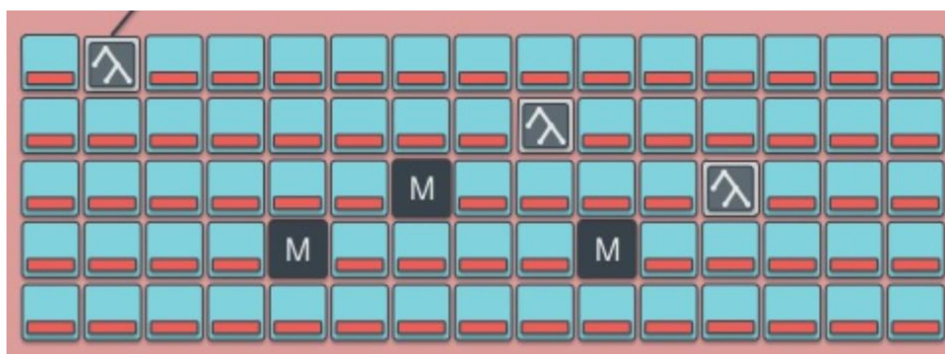

Figure 2. A Detailed Look of RADOS [10]

2.3 CRUSH Algorithm

As distributed systems separate storage resources to multiple server nodes, it is necessary to design a strategy to enable clients to know where are the data they want to read or write. Intuitive design is putting a monitor that stores metadata between clients and servers so that both sides could agree on the location of any specific data. However, Ceph uses a new placement policy called CRUSH (Controlled Replication Under Scalable Hashing), which bypasses the need for a central metadata server. In particular, both clients and servers will hold the same map

of data and calculate the location of an object independently. The result of the calculation will be the same, so clients and servers can agree on the location though they never decide it together.

2.3.1 Features of CRUSH

Pseudo randomness. CRUSH is a pseudo-random algorithm that uses a CRUSH map to do fast calculations without any lookup overhead. The reason why it is pseudo- random is that it is repeatable and deterministic. In other words, if the client uses CRUSH to calculate the object several times, without any crashes of disks or system failure, it should get the same answer, which is exactly the location of the object.

Statistically uniform distribution. Another great feature of CRUSH is its way to distribute objects. It is very significant for a distributed system to separate data evenly, and it is not hard for those systems that have a central manger to implement this feature because they just need to keep track of the status of each node and keep balance. However, for a hash-based algorithm, this task becomes harder due to the unknown world of a cluster. The pseudo-random hashing mechanism of CRUSH assures an analytical balance of distributing objects to achieve high performance.

Stable mapping. This feature means that when something in the input changes, the output of the algorithm is supposed to change as small as possible for easy maintenance. In particular, when some OSD crashes or shuts down accidentally, CRUSH will recalculate a new map with minimum change of OSD topology. With this feature, CEPH could move things as little as possible when some failure occurs in a cluster.

Hierarchical cluster map. The paper about CRUSH demonstrates elements building up the cluster map:

"The cluster map is composed of *devices* and *buckets*, both of which have numerical identifiers and weight values associated with them. Buckets can contain any number of devices or other buckets, allowing them to form interior nodes in a storage hierarchy in which devices are always at the leaves. Storage devices are assigned weights by the administrator to control the relative amount of data they are responsible for storing. [13]"

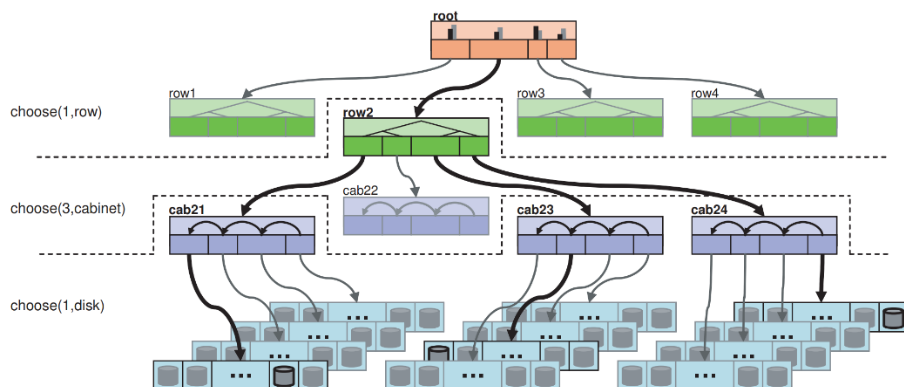

Figure 3. CRUSH map [13]

As figure 3 shows, in a CRUSH map, the actual storage devices (i.e. OSDs) are represented as leaf nodes of the map, and the interior buckets symbolize the topology of the physical hosts. Each node is assigned a weight for CRUSH algorithms. Also, by configuring the CRUSH rules, users could customize their CRUSH map for optimization.

There are four types of buckets that have different properties and serves for different purposes.

2.3.3 Types of buckets.

Uniform buckets. The mapping speed of this type is O(1), but it has an inferior performance on adding or removing nodes in the map. So, this type of bucket is good for the clusters which rarely changes the number of servers.

List buckets. This type of structure saves buckets on the same level in a linked list. Hence, the mapping speed of this type is linear, and it has an outstanding performance on adding nodes in the map because it only needs to insert the device into the head. This type of buckets is suitable for the clusters adding nodes frequently but removing them rarely.

Tree buckets. This type of structure saves buckets on the same level in a binary list, which is desirable for large-scale clusters. The mapping speed of this type is O(log), and it has a good performance both on adding and removing nodes.

Straw buckets. By randomness and weightiness related algorithms, this type of buckets could reach optimal performance for mapping, adding, and removing nodes in the map. However, in terms of configuration, it is the most complicated one amongst the four types.

2.3.4 Placement policy. The procedure for CEPH to place all objects to different OSDs contains three steps: TAKE, SELECT, and EMIT. The purposes for these three actions are apparent: the CRUSH algorithm takes an object waiting in some queue, selects the location it should go, and emits when all objects in the queue know their destination. The SELECT action is the tricky one. It goes through a CRUSH map, and when choosing which branch it should go, it will use the weight assigned to the current node and make some hash-based pseudo-random selection. In all, the SELECT action is the key algorithm that helps CRUSH to distribute objects statistically evenly.

2.4 BlueStore

Commonly, the operating system talks with disks via file systems, and file systems partition physical disks as small trunks and virtualize them as different files for the software. Distributed systems are still the software that still needs the help of operating systems and file systems to manage data. And Ceph followed this scheme for several years and finally created a new technology called BlueStore, which bypasses the medium file system layer and directly talk to the disks.

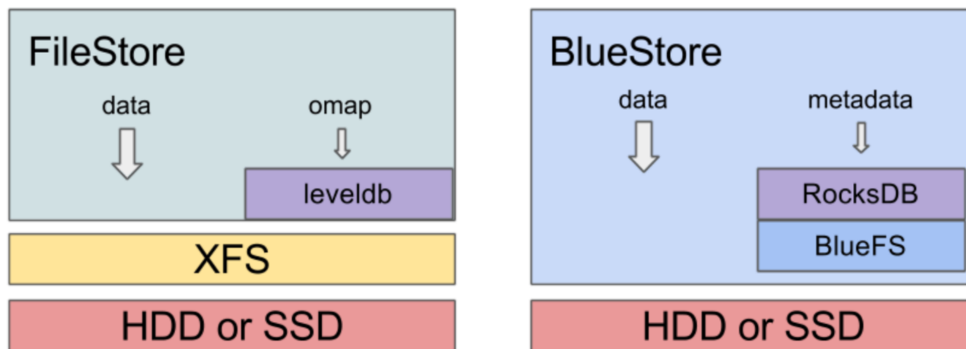

Figure 4. Comparison between FileStore and BlueStore [3]

As figure 4 shows, BlueStore is a combination of databases and file systems, which helps OSDs to manage low-level writes and reads. It contains an embedded database called RocksDB, which is a fork of LevelDB.

By storing key-value pairs, RocksDB could have efficient use of Disks, especially SSDs. It generally works as either an alternative back end for DBMS such as MariaDB and MySQL or directly works as an embedded tool for storage systems such as CEPH and SSDB. Further, especially for CEPH, RocksDB helps it to store the metadata of raw disks so that BlueStore could directly write objects into disks without any help of traditional file systems.

2.5 Messaging

The requests of clients (e.g., read and write) are encapsulated as Ceph messages. Moreover, messages are sent in an encoded format, and OSDs will not decode all information at once. This fact increases the difficulty for estimating how much storage resources a request may consume. A possible solution for this situation will show in Chapter 4.

2.5.1 Messaging Structure. Read and write related messages are encapsulated in MOSDOp, which is a subclass of MOSDFastDispatchOp. This type of message contains a vector that stores multiple sub-requests, called OSDop, which are layers of messages. Indeed, OSDop itself does not contain essential information about requests. Instead, the type, length, and offset of the request is buried in a final layer called ceph_osd_op, which is a member class of OSDop.

2.5.2 Message Handling Procedure. A message sent from the client through network firstly is checked by a message handler, which checks a specific flag to determine what kind of message dispatcher should be used in the next stage. All types of writing and reading related messages will be handled by the fast message dispatcher. The main job of the dispatcher is to assign the content of the message to a waiting queue, a so-called shard. There are multiple shards in an OSD, and each shard will be assigned to a specific worker thread. Next, a worker thread grabs out a message from its shard and starts to pass the message to the next stage. Specifically, the worker thread decodes the message, takes sub-operations in the operation vector out, and passes each of them to low-level storage.

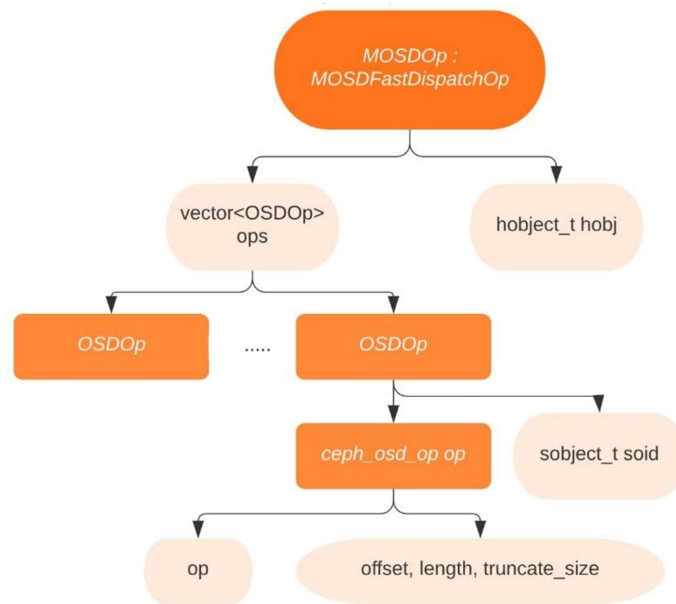

Figure 5. Message structure

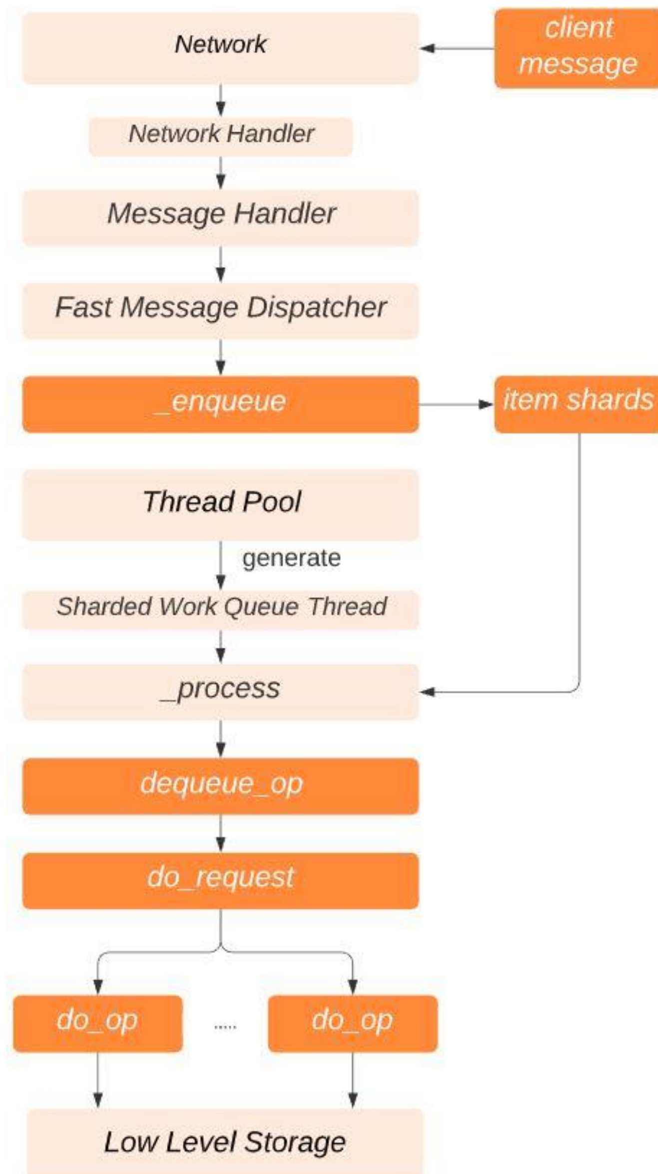

Figure 6. Message handling flowchart.

Chapter 3

Basic Model for Rate-limiting and Priority

3.1 Overview

Two core parameters determine the scheduling policy of the new QoS system: Priority and rate-limiting. Specifically, each workload has these two parameters, and the system determines the order of handling requests by them. Priority is easy to understand: it represents how significant a workload is. However, because of the rate-limiting mechanism, it is not true that the requests from higher prioritized workloads will always be handled before lower prioritized ones. Also, it is reasonable to do so because otherwise, workloads with low priority may easily fall into starvation. The mechanism of rate-limiting is easy: it records how much computation resources have already been consumed by a specific workload and constrains the workload if it utilized too many resources over time. This combination of parameters ensures requests from workloads that have stricter SLO could be handled with more care, while the requests from workloads that have a more lenient SLO will not be ignored during the busy period.

3.2 Challenges

3.1.1 Single vs. distributed. Since Ceph is a decentralized system, there is no place where all requests are collected and then redistributed. Hence, each OSD needs to have its own set of buckets. In other words, those buckets in different OSD but serving for the same workload do not know each other. To check if this mechanism is compatible with the built-in design of CEPH.

3.1.2 Insufficient information. As mentioned in chapter 3, Ceph sends encoded requests. Those messages are decoded at low level (i.e., right before BlueStore), where the model is hard to build. However, the system needs to know the type and size of the request in order to calculate the number of corresponding tokens at a higher level. To solve this issue, we design the bucket to add and subtract tokens in different layers. Further, whatever the size of a request is, it will be allowed to pass if the bucket is not empty. In other words, the bucket can store a negative number of tokens and can control the flow of requests, even if their sizes are unknown.

3.1.3 Achieve prioritization. The scheduling is processing at a software level. When the requests go into BlueStore, there is no guarantee for the system to handle the request with the highest priority. If there are too many requests are being processed, the model might lose the expected effect. Therefore, we want to constraint the number of requests so that prioritization can be achieved. In particular, there are two available ways: 1. We limit the number of worker threads that can concurrently processing requests; 2. We suspend worker threads when detecting that the number of requests reaches the limit. And we activate one worker thread once a request is processed successfully. Given that Ceph already has an upper bound for the number of worker threads, which is not very large, option 1 would be a more sensible choice.

3.3 How Bucket Model and Priority Work Together

The rate-limiting bucket is a quantitative model used to constrain the number of requests from workloads during a busy period. When a server starts to work, a bucket for every potential workload is created, and it is filled with tokens. When the server received a request from a specific workload, it will first check how many tokens the request is worthy of. Then, it will

subtract the tokens from the corresponding bucket and finally let the request be handled by low-level functionalities. If the system finds the bucket is empty or the tokens left is not enough, it will idle the request and handle other requests. Also, the token is refilled at a given rate periodically so that the idled request has a chance to be handled later.

Chapter 4

Design and Implementation

The system consists of the analysis part and the service part. The analysis part is responsible for calculating the parameters, such as bandwidth tables and bucket-related values, that are applied to the service part. And the service part runs the server with the given parameters to do scheduling and enforcing tasks. With the specialty of Ceph that we need to get the data for analysis by directly talking to a cluster, the analysis part cannot stand alone and do its math. Instead, it has to talk to the Ceph cluster via LIBRADOS to get the necessary data.

4.1 Scheduler

The scheduler is a thin layer in every OSD that controls the scheduling for messages from all pools. It achieves the bucket model by embedding different parts in different layers of OSDs.

4.1.1 initialize or update the pool structure. Each OSD contains a vector of structures that represent and store parameters of the bucket model for each pool. The maximum token and the rate of adding tokens, which are two of the parameters, are determined by the network calculus calculator in the analysis part and passed via RPC (remote procedure call).

4.1.2 find the best pool. When the system tries to grab a message item from shards, the scheduler determines whose message should be picked based on the bucket model. Specifically, the scheduler firstly compares a pair of pools and then let the winner to compare with the next pool and so on. In this way, the scheduler searches for the best pool with a linear time complexity, which is acceptable as each comparison is not expensive. The algorithm for comparing two pool is given in appendix A.

4.2 Profiler

A profiler empirically measures the performance of storage. It generates a bandwidth table that will be used by estimators to calculate the token value of a request. Since Ceph has built-in commands to benchmark the storage, instead of testing performance from scratch, we can run different commands to collect necessary data to build the table.

The testing cluster is built upon SSD storage. Hence, we need multiple parameters to profile due to the complexity of an SSD device. Firstly, since an SSD device has a great performance difference between writing and reading requests, the profiler will create a table for each type of request. Also, the size of a request is another factor affecting the performance. In general, the overhead of a small request dominates the processing time, so the bandwidth of small requests is less than the bandwidth of large ones. In all, the profiler will generate a table that returns the bandwidth with the given request type and size.

4.3 Estimator

An estimator is a tool that takes features of a trace (i.e., a request) and converts it to token values. Specifically, for an SSD-based Ceph cluster, the estimator takes the request size and type of a trace as input parameters. And it uses the bandwidth table generated by the profiler to calculate the corresponding token. Specifically, the estimator firstly finds the two entries of the bandwidth table that the request lies between and then calculates the specific bandwidth for the request via linear interpolation. Finally, the estimator returns the token value, which is the request size divided by the bandwidth.

Both the analysis part and the service part use the estimator. However, the implementations in the two parts are different since the code embedded in Ceph does not follow the structures in the prior work [15].

4.3.1 The estimator in the analysis part generally follows the structures in PriorityMeister. The old estimator tells the network calculus calculator, which determines bucket parameters, the corresponding token value of a given trace. With a distributed system, however, a single token value is not accurate enough to describe the resource-consuming behavior because different OSDs spend different resources for the trace. Therefore, the token values in the new system are vectorized, and one value only reflects the resource consumed by a single OSD. In fact, in a cluster where objects have no replication, there is only one OSD handling a specific trace. Further, for convenience, the system takes the maximum r-b points as a lower bound for performance among a group of OSDs so that all OSD in the same group will finally have the same bucket model. However, they may have different ones initially.

4.3.2 The estimator in the service part is different from the prior work [15]. After a message is decoded, the system is able to get request type and size, so it will calculate the corresponding token value by calling the estimator and will subtract it from buckets. Further, the bandwidth table is stored in a fixed place (e.g., /etc/ceph) so that Ceph can load it in the initializing procedure. The last worth-mentioning fact is that despite the vectorized token values in the analysis part, the return value of estimator in the service part only returns a single value, as the message is distributed to a specific OSD before the estimating procedure.

4.4 Trace Replay

After the parameters are determined and Ceph is activated, the system will test the performance by sending requests toward the cluster. Specifically, following a trace script describing request patterns that reflect some real-life scenarios, clients in the system send messages to read or write an object. Each trace consists of the offset, size, and type (i.e., read or write) of a file. In the prior work [15], since NFS virtualizes the remote storage as a file system and the clients realize the trace behavior via operating on a large file, the parameters in a trace can be directly used to implement a request. However, when it comes to a Ceph cluster in which data are stored by objects, the system uses LIBRADOS to talk to a cluster, and all operations are object-wise. Hence, the offset and size need to be twisted if we want to follow the same pattern with the prior work [15].

We solve the issue by creating a large number of same-sized objects in the cluster. Each object represents a block of a storage device. The size of each object is the sum of the maximum request size and a relatively large pre-determined block size. Whenever a client read a trace and transfer it to a real request, it will calculate the block number and generate a new in-block offset. Then, it uses those parameters to send the request. If the block size is much larger than the maximum request size, the requests doing wrong-behaved operations are negligible.

4.5 Modified Shards

In Ceph, messages sent from clients are queued in some places, called shards, in each OSD. Each shard contains a scheduler, which is a priority queue with a particular dequeue policy. By default, messages from different pools are stored together in a shard.

In the new system, we twist the structure of shards and filter messages from different pools. Specifically, there are now multiple schedulers in one shard, and each scheduler only queues messages from the same pool. In other words, each shard can still have messages from different pools, but messages are stored in different queues inside a shard. When a message is enqueued, the message dispatcher uses the original way to determine which shard the message should go, then put it to the correct scheduler by the message's pool id. When a worker thread tries dequeue from a shard, it uses the bucket rate limiters to determine the pool from which the message should be processed and then uses a map to find the scheduler that stores messages from that pool.

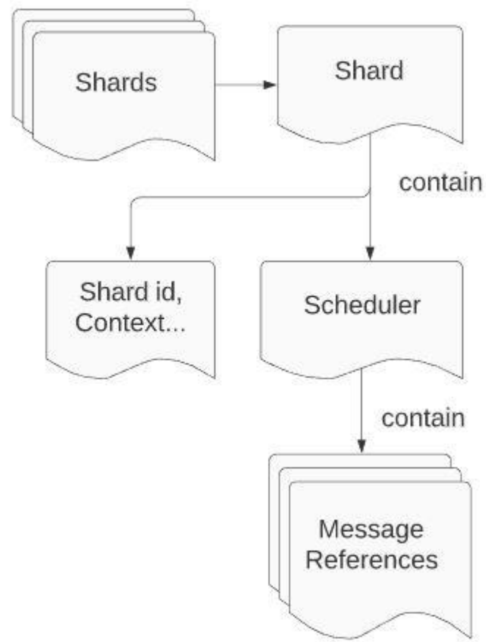

Figure 7. Shard structure in Ceph

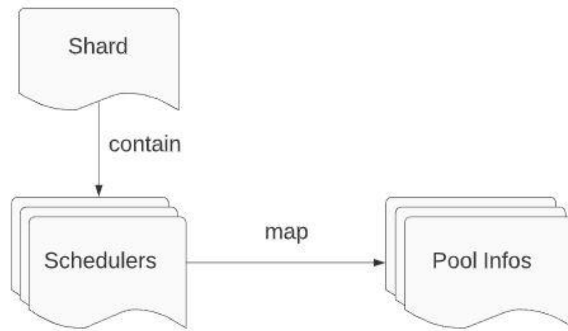

Figure 8. Modified shard

Chapter 5 Experimental Results

5.1 Setup

All experiments are operated on VMs supported by AWS. The Ceph cluster consists of three t3.medium VMs with 30 GB mounted EBS. The workloads are running on a t5.xlarge VM to send requests. Moreover, three pools representing workloads are called *client1, client2 and client3* with SLO 0.3, 0.4, 0.5, respectively. In other words, client 1 has the highest priority, and client 3 has the lowest priority.

5.2 Benefits of Priority and Rate-Limiting

The major benefit of the new system is that it differentiates the importance of pools. Another benefit is that while satisfying user-defined SLOs, the system does not blindly handle requests from the pool with a high priority. Instead, the rate-limiting controllers prevent low priority pool from starvation by capping high priority pools when they behave abnormally. The following two experiments demonstrate these benefits.

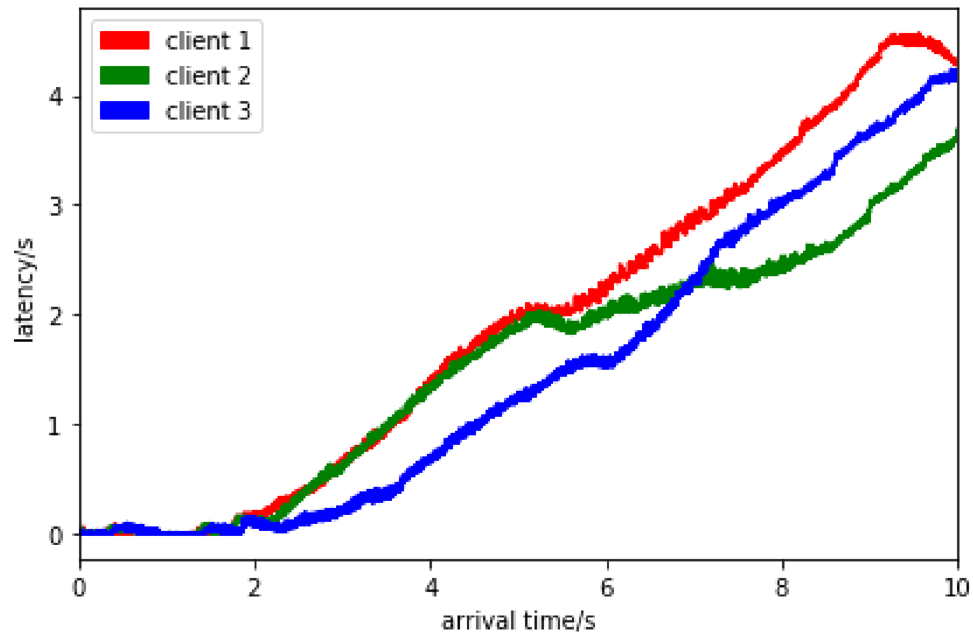

Figure 9. Overloading the server with neither rate-limiting nor priority

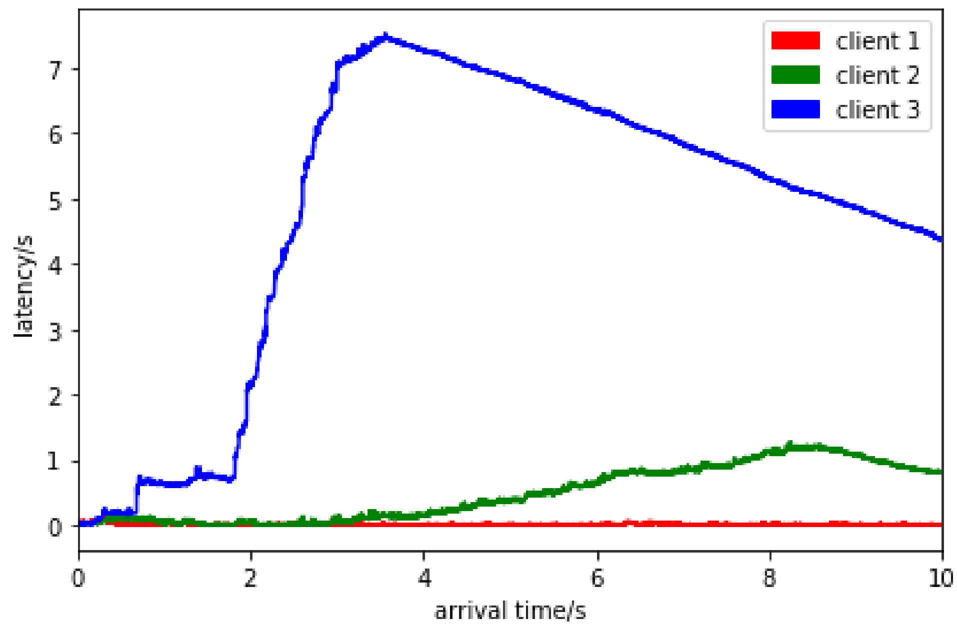

Figure 10. Overloading the server without rate-limiting

5.2.1 Experiment for prioritization. Three clients are running at a rate of 1000 IOPS , which overloads the server. Figure 9 shows the experiment in which both rate-limiting constraints and priority are disabled and figure 10 shows the experiment in which only rate-

limiting is disabled. We could see that the system, indeed, processes requests with the highest priority first and *vice versa* when prioritization is enabled.

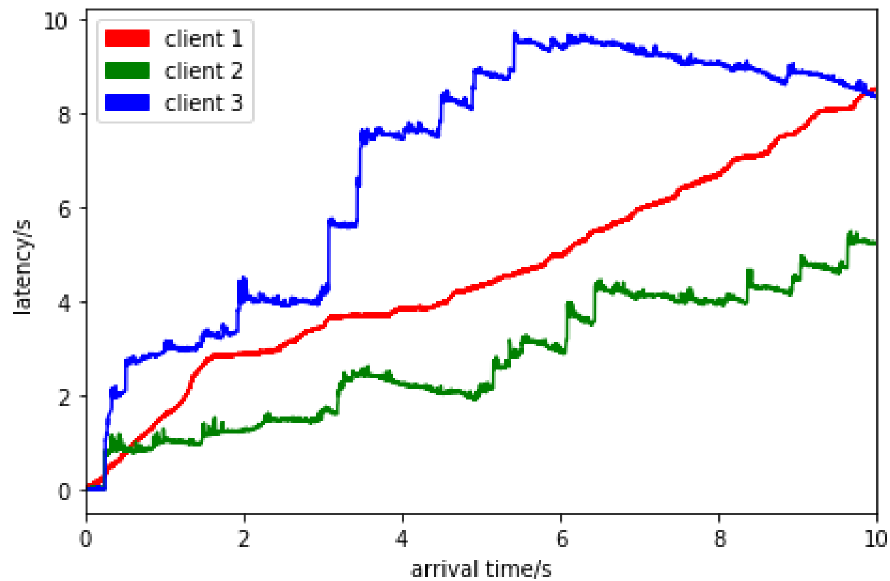

Figure 11. Burst client 1 without rate-limiting

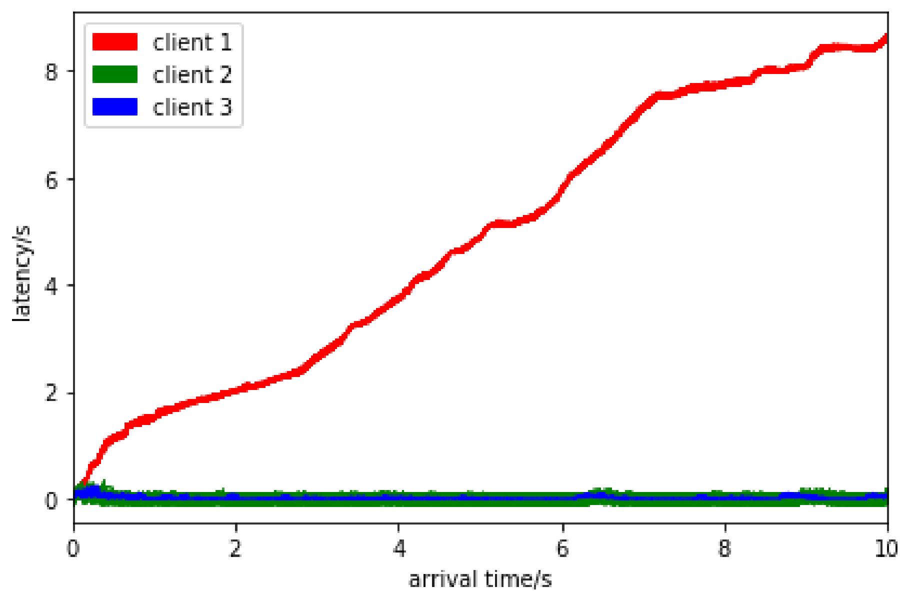

Figure 12. Burst client 1 with rate-limiting

5.2.2 Experiments for rate-limiting. In this experiment, client one is running at 3000 IOPS, which overloads the system, and the other clients are running at 300 IOPS. Figure 10

shows a scenario where rate-limiting is disabled, and figure 11 shows the opposite. We could see that rate-limiting ensures the performance of low-priority clients when a high-priority client is in a burst period.

5.3 Benefits of Distributed Storage Modeling

Besides system and structural differences, the major improvement of the new system compared with the prior work [15] is that the new models vectorize the shared storage instead of considering it as big single storage. The following experiments show the performance difference and trade-off between the distributed model and two baseline models under extreme cases. All three clients have a running rate of 1400 IOPS, and either all of them have a perfectly balanced trace (i.e., the number of requests going to 3 OSDs are equal) or completely unbalanced (i.e., all request goes to the same OSD).

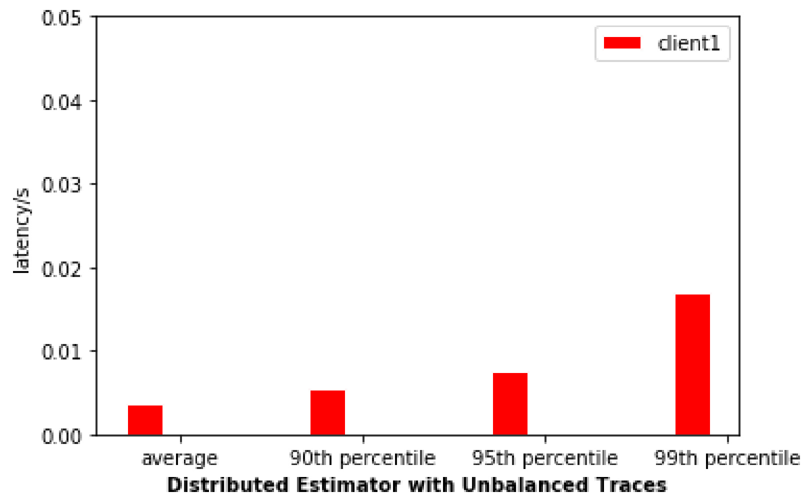

Figure 13. Distributed estimator with unbalanced traces

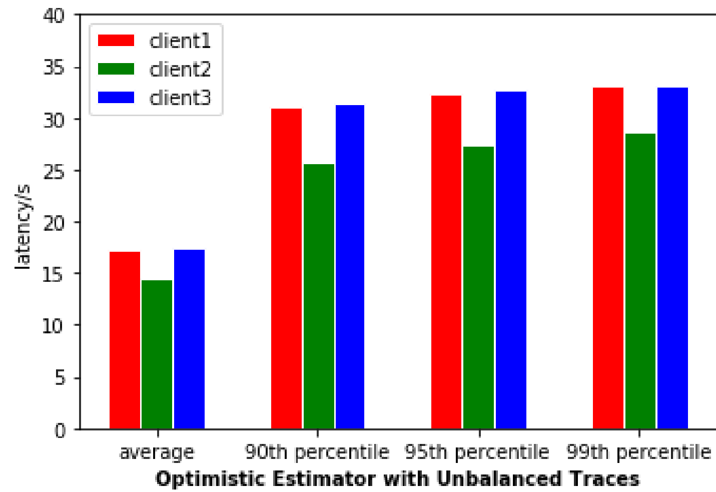

Figure 14. Optimistic estimator with unbalanced traces

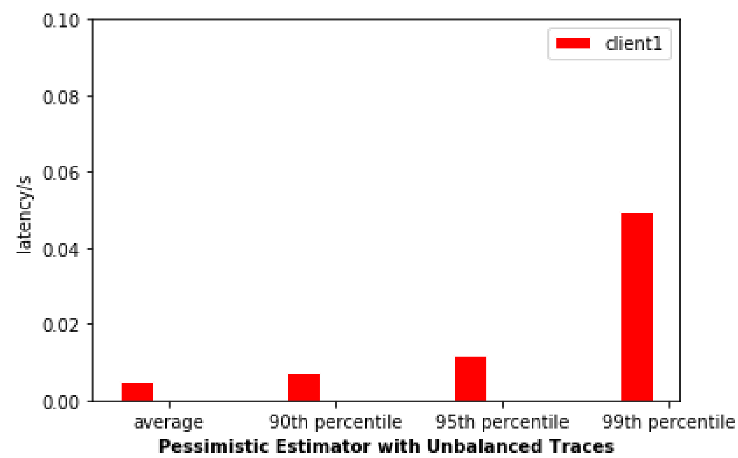

Figure 15. Pessimistic estimator with unbalanced traces

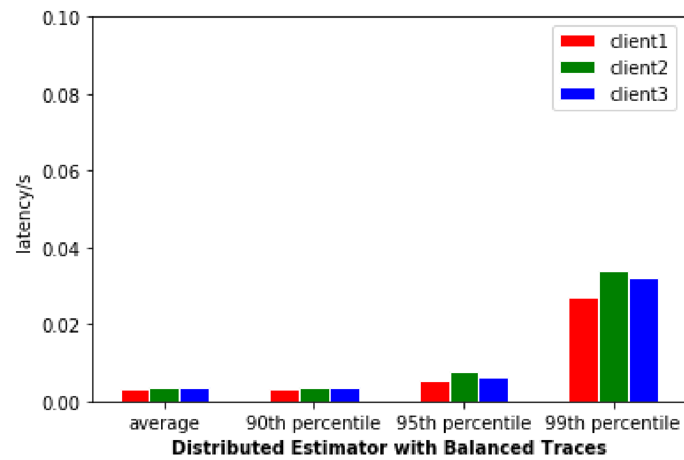

Figure 16. Distributed estimator with balanced traces

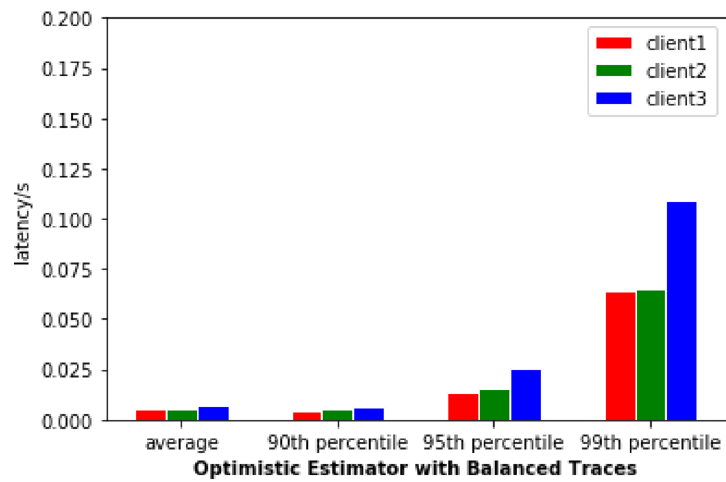

Figure 17. Optimistic estimator with balanced traces

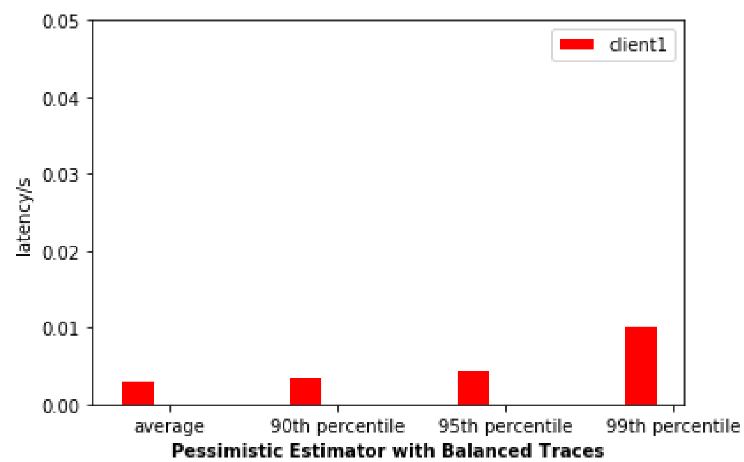

Figure 18. Pessimistic estimator with balanced traces

5.3.1 Balanced traces. Figures 13, 14, and 15 show the performance of the system with three different models when the traces are perfectly balanced. We could see that both the optimistic estimator and the distributed estimator admit all clients, which is expected behavior. However, the pessimistic estimator does not admit all clients and causes the storage system underloaded.

5.3.2 Unbalanced traces. Figures 16, 17, and 18 show the performance of the system with three different models when the traces are completely unbalanced (i.e., all traces were accessing the same OSD). We could see that both the pessimistic estimator and the distributed estimator admit only admit one client, which is expected behavior. However, the optimistic estimator admits all clients and causes the storage system overloaded.

Table 1. Comparison among 3 models

	Overloading?	*Underloading?*
Optimistic model	Possible	Never
Pessimistic model	Never	Possible
Distributed model	Never	Never

5.3.3 Conclusion. The optimistic model assumes the server can take advantage of parallelism for any trace behavior. We could see that no matter if traces are balanced or not, the model uses the highest bandwidth to calculate the rate-limiting parameter and admits all workloads. This strategy works perfectly fine when the trace is balanced. However, when the trace is biased to a single OSD, the model overestimates the performance of the server, and admits too many requests which overload the system.

The pessimistic model assumes the server is always running at its minimum bandwidth. The advantage of the model is that it never overestimates the performance of the server, and the

model correctly characterizes the performance when traces are extremely biased to a single OSD. However, when traces are balanced, the model still admits only one client even though the real bandwidth supports three clients, so the storage and time resource is wasted.

With the help of an object location map, when calculating rate-limiting parameters, the distributed model knows and records how traces are distributed by vectorizing the bucket model. In this way, the model does not assume the trace distribution of the server. Hence, the model prevents the server from overloading and underloading by admitting an appropriate number of clients.

In conclusion, as shown in table 1 and experiments, we could see that the accuracy of two baseline estimators heavily depends on the behaviors and characteristics of traces. In contrast, the distributed estimator can model the burstiness correctly in both edge cases.

Chapter 6
Related Work

Supporting latency SLOs is an active research area with abundant literature. The related works could be partitioned into two categories.

First, there is a body of work that tries to dynamically adapt system parameters to meet latency SLOs [5, 6, 8, 11, 14]. However, not all of the work supports tail latency SLOs. For those that do, they generally work well in well-behaved scenarios, but they have undesirable performance with bursty workloads.

Second, to address the shortcomings of the first body of work, some research has taken analytical approaches to support latency SLOs [4, 7, 15, 16, 17]. All of them are theoretically supported by a branch of mathematical analysis, known as Network Calculus, for analyzing the queueing behavior in the system. Silo [7] and QJUMP [4] are state-of-the-art systems focusing on the network queueing mechanism. PriorityMeister [15] is one of the systems that achieve end-to-end tail latency SLO in a networked storage system. SNC-Meister [16] is a more advanced system than PriorityMeister that could admit more workloads with the support of Stochastic Network Calculus. WorkloadCompactor [17] is also a more advanced system than PriorityMesiter but it focuses more on workload placement. All these systems show robustness to handle bursty workloads by analyzing the workload behavior. However, all these systems are only applicable to networks, hard drives, and SSDs, but not distributed storage. As we show this thesis, it is non-trivial to extend this work to distributed storage because distributed storage clients interact with multiple storage servers, which causes various interference patterns.

Chapter 7
Future Work

7.1 Network Rate-limiting

In the prior work [15], the enforcer consists of the network part and the storage part. The old network enforcer only applies to a one-to-one case, in which the server is centralized. Hence, the network is modeled by a fixed number of network paths, including a client-in path, a client-out path, a server-in path, and a server-out path. However, in a shared network context, there are multiple server hosts so that a single server-out/server-in network path is not accurate. Moreover, based on the CRUSH map, pools may store in different OSDs and overlap with each other, which makes the network modeling more complicated. The next step after this thesis is to see the availability of migrating the network model. And if it is unable to do so, what would be a new tool to analyze the network traffic in a shared network.

7.2 CRUSH Rule Modification

In Ceph, a CRUSH map configuration file empowers users to control the destinations of pools. By using a map wisely, the system can handle burstiness by ways other than rate-limiting. For example, if a pool with high priority works intensively, the system could move it to a new OSD that has more bandwidth left so that the SLOs of pools with low priority could be guaranteed. In all, we could envision a model that configures CRUSH maps wisely to distribute requests in an effective way to prevent high interference among workloads.

Appendix A

Algorithm for Scheduling Requests

Name: ComparePools
Input: pool information p1, p2
Output: 1 if p1 should be handled first; -1 if p2 should be handled first; 0 if they are of the same importance
***Algorithm*:**

If p1 is *empty*:
    If p2 is *empty*:
        Output 0
    Else:
        Output -1;
Else if p2 is *empty*:
    Output 1

If p1 obeys *rate-limiting*:
    If p2 obeys *rate-limiting*:
        If *Priority* of p2 > *Priority* of p1
            Output 1
        Else if *Priority* of p1 > *Priority* of p2:
            Output -1
    Else:
        Output 1;
Else if p2 obeys *rate-limiting*:
    Output -1;
//Use Round-robin if the scheduler cannot decide which pool should be handled first
Output Round-robin(p1, p2)

BIBLIOGRAPHY

[1] Abutalib Aghayev, Sage Weil and Michael Kuchnik *et al.* File system Unfit as Distributed Storage Backends: Lessions from 10 Years of Ceph Evolution. In *SOSP'19* Pages 353-369. DOI: https://dl.acm.org/citation.cfm?doid=3341301.3359656
[2] P. J. Braam. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug. 2004. /lustre
[3] CEPH. Ceph: Why to Use BlueStore. Retrieved December 9, 2019 from https://www.oddeye.co/blog/ceph-why-to-use-bluestore/
[4] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In USENIX NSDI, 2015.
[5] Ajay Gulati, Arif Merchant, and Peter J. Varman. pclock: an arrival curve based approach for QoS guarantees in shared storage systems. In Proceedings of *the* 2007 *ACM SIGMETRICS international conference on Measurement and modeling of computer systems,* SIGMETRICS '07, pages 13–24, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-639-4. doi: 10.1145/1254882.1254885. URL http://doi.acm.org/10.1145/1254882.1254885.
[6] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. Parda: proportional allocation of resources for distributed storage access. In Proccedings of *the 7th conference on File and storage technologies*, FAST '09, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm? id=1525908.1525915.
[7] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *ACM SIGCOMM*, pages 435–448. ACM, 2015.
[8] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457– 480, November 2005. ISSN 1553-3077. doi: 10.1145/1111609.1111612. URL http://doi.acm.org/10.1145/1111609.1111612.
[9] O. Rodeh and A. Teperman. zFS—a scalable distributed file system using object disks. In Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies, pages 207–218, Apr. 2003.
[10] Ross Turk. CEPH Intro and Architectural Overview (May 2013). Retrieved December 8, 2019 from https://www.slideshare.net/buildacloud/ceph-intro-and-architectural-overview-by-ross-turk
[11] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: enabling high-level slos on shared storage systems. In Proceedings of *the Third ACM Symposium on Cloud Computing, SoCC '12,* pages 14:1–14:14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1761-0. doi: 10.1145/2391229.2391243. URL http://doi.acm.org/10.1145/2391229.2391243.

[12] B. Welch and G. Gibson. Managing scalability in object storage systems for HPC Linux clusters. In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies, pages 433–445, Apr. 2004.
[13] Sage A. Weil, Scott A. Brandt, Ethan L. Miller and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. URL https://ceph.com/wp-content/uploads/2016/08/weil-crush-sc06.pdf
[14] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, August 2006. ISSN 1553-3077. doi: 10.1145/1168910.1168913. URL http://doi.acm.org/10.1145/1168910.1168913.
[15] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *ACM SOCC*, pages 29:1–29:14, New York, NY, USA, 2014. ACM. ISBN 978-14503-3252-1. doi: 10.1145/2670979.2671008. URL http://doi.acm.org/10.1145/2670979.2671008.
[16] Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. SNC-Meister: Admitting MoreTenantswithTailLatencySLOs. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 374–387, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987585. URL http://doi. acm.org/10.1145/2987550.2987585.
[17] Timothy Zhu, Michael A. Kozuch, and Mor Harchol-Balter. Workloadcompactor: Reducing datacenter cost while providing tail latency SLO guarantees. In *ACM SoCC*, pages 598–610, New York, NY, USA, 2017. ACM. URL https://dl.acm.org/doi/10.1145/3127479.3132245.

ACADEMIC VITA

Shaobo Guan

EDUCATION

Pennsylvania State University (PSU), PA, USA **Aug. 2016 – May. 2020**
B.S. in Computer Science with honors, minor in Math

RESEARCH EXPERIENCES

Optimization Scheme of Tail Latency for Shared Network Storage **Mar. 2019 - Apr. 2020**
Researcher, EECS Department, PSU | Honor Program Thesis Supervisor: Timothy Zhu

- Conducted research on optimizing CEPH tail latency.
- Performed literature reviews and gained understanding of CEPH, a distributed storage system.
- Developed new models to the source code to test on Amazon AWS, CEPH.
- Conducted performance testing, and iterated on incremental improvements to reach lower tail latency within SLOs.

Development of Dialogue Management Architectures for Human-Robot Communication **Sep. 2019 - Dec. 2019**
Member, NLP Lab, PSU | Supervisor: Prof. Rebecca Passonneau

- Participated in a novel human-robot communication system design based on meaning representation language (MRL) that is a variant of first order logic (FOL).
- Collected training data to build machine learning models to learn to generate English from MRL.
- Paired each MRL expression from robot dialogues with an English phrase of roughly equivalent meaning based on collected data.
- Tested a graphical user interface for the data collection, creation of natural language data, and evaluation of the trained models.

WORK EXPERIENCES

Udcredit Network Co., Ltd. **Jul. 2018**
Software Development Engineer Intern, Hangzhou, China

- Developed blockchain application components using Java Web, and built Linux scripts.
- Designed and wrote a program converting digital contracts sent by customers on Internet to Java format versions, then returned to customers.

COURSE PROJECTS

Network Proxy Design and Implementation

- Designed a network routing system that can dynamically perform load-balancing and proxy requests to idle servers.
- Implemented a cache layer that can automatically caches popular content to enable off-line functionalities.

Distributed Algorithm Developed for Map-Reduce

- Developed a map-reduce library with PThread to manage multiple workflows.
- Implemented custom semaphore logics to solve concurrency issues during task scheduling.

HONORS

- The Evan Pugh Scholar Award (PSU) **Apr. 2019**
- The President Sparks Award (PSU) **Apr. 2018**
- The President's Freshmen Award **Apr. 2017**

ADDITIONAL INFORMATION

- Technical Skills: Python, C, C++, MATLAB, JAVA, SQL, VeriLog.
- Standard Tests: TOEFL 111 (R30 L29 S26 W26); GRE 330 (V162 + Q168) + AW3.0