Zoolander: Modeling and managing replication for predictability

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Daiyi Yang

Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Master's Examination Committee:

Christopher Charles Stewart, Advisor

Gagan Agrawal.

Abstract

Social networking and scientific computing workloads access networked storage much more frequently than traditional static-content workloads. These workloads improve their speed by issuing requests parallel, which offers a lot speedup if its slowest parallelly access is fast but will also suffers some unexpected slowdowns if the former assumption cannot be guaranteed. In this paper, we study replication for predictability, which speeds up the slow storage accesses by running the same workload on duplicate nodes and using the fastest response, different from traditional replication for throughput. Based on the mechanism of replication for predictability, we proposed Zoolander, an analytical model predicting the percentage of quickly completed access, i.e. SLA. Zoolander combines factors of the replication strategies, the distribution of heavy tail access and the queuing delay, to output the most efficient solution. Then we created an enhanced Zookeeper like coordination service supporting replication for predictability. Zoolander was precise which could achieve SLA of 0.002 absolute errors over diverse workloads. Using Zoolander, we achieved speedups of 375% and reduced the cloud servers needed by 50%.

This document is dedicated to my family.

Acknowledgments

I have many people to thank for my making it this far: my advisor, Prof. Christopher Stewart, for everything he's done; all of my department mates, for their knowledge, assistance, and encouragement; and the CSE Department staff for everything they do.

Vita

2005.....Jinshan High School

2009.....B.E. Software Engineering, Beihang University

2009 to present .....M.S. Computer Science & Engineering, The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 SLA Agreement (SLA) and Data Intensive Workload

Service level agreement (SLA) usually describes a boundary of the latency for a certain type of requests or storage accesses at different percentile level; it provides a quantification measurement of the performance for a workload. For example, a SLA says "99% of the read requests can be responded within 200ms"[1][2][3], this SLA can be seen in a lots of traditional static content workload which allows us to expect the majority of its requests to be finished fast. Data intensive workloads like those social networking applications or large scale simulation programs which always contain several TBs (Terabyte) or even PBs (petabyte) of data, and have to respond millions of requests in one second, are very different from the traditional static content workloads.

This type of workloads usually uses the replication or partition strategy with parallel accesses to help speed up its response time. Take the MapReduce [4] framework as an example, which is a widely used framework in data-intensive workload proposed by Google. MapReduce usually finishes one taskes in several phases, each of which divide jobs and "map" every piece into different nodes for processing then combine and "reduce" to finish the task. In such workload as described above, even one of its parallel accesses in a certain phase gets an unexpected long finished time; the whole task will be delayed. So the response time for data-intensive workload depends on its slowest parallel access

rather than the mean, which calls for a more enhancement storage system with higher SLA (e.g., 99.99% of all accesses), to maintain the low latency response for the whole system.

1.2 Solutions to Improving SLA

Replication and partition are widely used strategies to improve the SLA in traditional workloads, and workloads with parallel accesses demand for more enhanced SLA storage system. There are many studies in this field for the purpose of improving the whole SLA and former researcher mostly focused reigning in performance anomalies which reduce the percentile of expected long latency (e.g., make the whole distribution of response follow the normal distribution with low variance). They solved this problem by either finding the root cause of the anomaly then fix it [22] or avoiding the anomaly in real-time by using some online management systems [2, 26, 29].

Specifically, there is also a different solution to improve the SLA by masking the long latency responses, which is proposed by SCADS director [29], called replication for predictability. SCADS director deploys two copies of exactly same storage replicas, then forces every client to send its requests to both of the two copies and use the first reply as its result. The first reply is always not an anomaly so the SCADS manager can mask some of the non-deterministic performance anomalies in this way. This replication for predictability is quite different from the traditional replication for throughput, because it does not actually improve the throughput although it does reduce the variance. However, prior work [29] as SCADS director merely used replication for predictability for the ad-

hot goal; their replication number is limited only up to two and neither of them can gain the expected performance goals demanded for the date-intensive workloads. We discuss replication for predictability with more detail in 1) modeling its benefit with and 2) using it to improve SLA for both write and read requests.

1.3 The Zoolander

Zoolander is an analytical model based on replication for predictability; it predicts SLA in the form like: how many percentiles of the requests can be finish within certain latency. The input of Zoolander contains four parts:

(1) target latency bound

(2) service time distribution

(3) storage access pattern

(4) the network latency distribution

And the online management system of Zoolander enhances it with the scalability under different platforms and inputs, e.g., if the arrival rate doubles of the target bound increases. We then created the RP Zookeeper to validate the Zoolander, which is an enhancement storage system based on the Zookeeper Coordination Service by adding replication for predictability. RP Zookeeper can achieve improvement by 2 orders of magnitude compared to traditional replication for throughput system, in its SLA of 15ms (99.995% v.s. 99%). Moreover, it can reduce the execution time of a scientific computing workload by 373% in the guide of Zoolander. And the validation shows that

the SLA predicted by Zoolander can achieve less than 0.002 errors across different workload, Zookeeper configuration, and hardware platform.

1.4 Organization of this Thesis

This thesis is organized as follows: Chapter 2 describes the data-intensive we target and explains the reason of their demand for the high SLAs. Chapter 3 shows the difference of our work compared with prior performance models and storage systems. Chapter 4 analyzes the principles of replication for predictability and presents the Zoolander's prediction model. Chapter 5 shows the implementation of RP Zookeeper and its system validation based on Zoolander. Chapter 6 proves the usage of Zoolander in the online management on two data-intensive benchmarks. Chapter 7 gives the conclusion.

Chapter 2: Background

Cloud computing allows people to combine the computing power of low cost machines together to replace the traditional expensive custom super servers in some certain usage like network storage and parallel computing. This provides a possible way to achieve the same computing power but with lower cost, benefitting end users with faster and real-time operations in network application and scientific users with lower cost computing power.

2.1 Storage System based on Cloud based Architecture

Storage systems in the cloud computing environment are usually designed to be in-memory for the purpose of best performance. It is supported by the thousands of low cost machines comprised in the datacenter. Compared to the traditional custom super server, which always consists of many computing units on a shared memory, machines in the large datacenter share only the power delivery and the network resource, which is called share nothing design. The advantage of this design is obvious: hardware failures can be cover simply by a simple fail-over redundancy strategy since each of the nodes is independent; and one typical storage access (usually containing a network round-trip, some light computation and one memory access) can be finish fast as long as the node accessed are working correctly. But once the application has some consistency demand that forces nodes to communicate, the latency of access can increase sharply and it also

limits the scalability of the whole performance. One possible solution is to use key-value based storage structure with proper partition or replication strategy; the problem here is when and how to apply the replication or partition to achieve the best performance. Zoolander provide some hints on this question to help managers find a most ideal solution.

2.2 Date-intensive Workloads

Traditional workloads like e-commerce workloads based on static content also use the partition and replication to improve their SLA but the SLA they need is not as high as the data-intensive workloads: 99% of fast storage accesses can sufficiently meet their requirement for performance because there is only a few storage accesses in these workloads need to be finish in extremely short latency. Hence one can often expected 99% SLA on the webpage response time from the 99% SLA on the storage access in this type of workloads.

On the other hand, the SLA demand of data-intensive workload is much higher, because they need much more fast storage accesses and even a few long latency storage accesses can cause performance decrease of the whole system. We describe two of such typical workloads below.

2.2.1 Inner-Join Olio

Unstructured data storage systems become popular today because their advantage in scalability allows people easily scale one hash-table into several partitions to maintain

the same performance when data size increased. The inner-join olio presents a typical workload which is used in the social networking service based on the unstructured data storage system. We created it from changing the SQL based social networking service Olio into hash-table data storage. The inner-join olio is totally unstructured without any index. Specifically, here is a typical request in the inner-join olio: To find all the attendees of a given event with id 100 whose first name begins with character “A”;

The SQL query, if it uses the SQL based storage system, will be:

```
SELECT username
From Person INNER JOIN eventandperson
ON eventandperson.eventid = 100 and
person.firstname = ‘A%” ;
```

Since inner-join olio is based on the unstructured data storage with the key-value structure, this request will be process in two phases: First, all the username of attendees of event with id 100 must be retrieved; Then, the username in the result set returned by the phase 1 whose first name begin with character “A” can be achieved and returned. Since social networking service usually contains a large amount of data, each of the two phase described will contain a large number of storage accesses. We can reduce the whole request’s response time by parting the storage access in each of the phases into batches and executed parallel in different nodes. However, the phase two cannot be started until all the storage accesses in the phase 1 finish and get the dataset returned, and the whole request cannot be responded until all the accesses in the phased 2 are done. Different from the traditional workload, even one access get unexpected delayed here, the

whole request will be delayed even though other may be finished quickly, because the barrier like phase mechanism here. This means that in this workload, the slowest parallel access can actually determine the lower bound of the response time of a whole request. Hence, to guarantee a similarly good performance like static-content workloads, inner-join olio needs more accesses in it to be quickly executed, which means that it demand a higher SLA storage system.

2.2.2 Gridlab-D

Gridlab-D is a more complex event-driven smart-grid simulation application running on a share-nothing cloud, to model the distribution of energy in many devices, from the power plants to air conditioners. The working process of Gridlab-D is like a chain of event trigger mechanism: one event cannot be simulated and trigger the succeed one until the former ones being finished and trigger it.

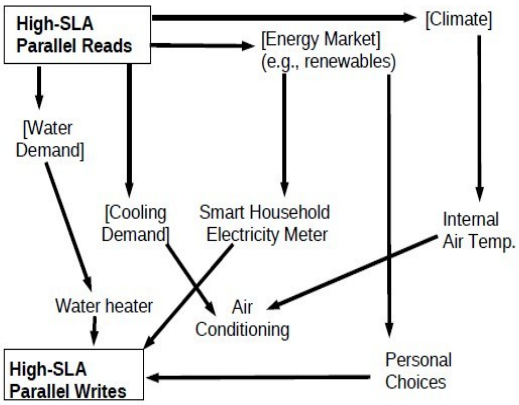

Figure 1: The execution process of a simulation in Gridlab-D, the arrows indicate the data dependence and the trigger order.

For example, a track of the energy usage for a household, a water heater and also with some personal choices in it may be simulated as the process shown in the Figure 1: all the events will be started only after the storage accesses which are some High-SLA parallel reads to be finished; then some of the simulation can be parallel distributed into different nodes to be executed and the other having some data dependence must wait to be trigger from its former event; finally, after all the simulations get finished, the result can be wrote back to the storage system through some High-SLA parallel writes. The storage accesses here actually consists two phase which are a bunch of parallel reads then writes but parallel writes cannot be executed until all the parallel reads finish. Hence, the slow storage accesses in this workload can affect a large number of simulation executions which depend on them, inflating the total response time.

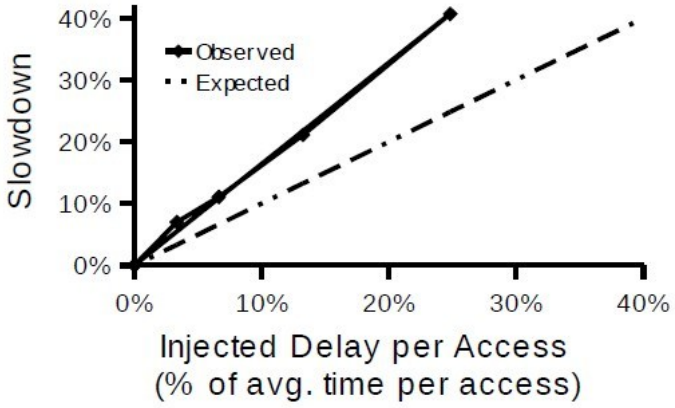

Figure 2: The slowdown result of the Gridlab-D workload with random delay injected into 2% of its storage accesses. The dotted line shows the expected slowdown and the solid line shows the actual slowdown observed.

To prove this, we implemented the Gridlab-D workload for a 5 minutes simulation then injected a random delay into 2% of its all storage accesses. Unlike the traditional static content workloads, whose slowdown is probably proportional to the delay injected, we actually get a result with 53-133% larger delay as shown in Figure 2. This means that a few slow accesses can cause an unexpected long slow down.

In sum, in a data-intensive workload, to guarantee all of its response time to be with a high SLA, its storage accesses must be completed quickly within a tight time constraints because the response time will get more affect from a small number of slow accesses. This outsized effect calls for a scalable solution which could easily be applied to existing cloud storage system that improves the SLA by several orders of magnitudes while maintaining the same low latency.

Chapter 3: Related Work

As been pointed in the last Chapter that data-intensive services require a storage system that guarantees much high SLA because of the devise and damaging workload it may produce. Towards this problem, much prior work has been proposed. In this chapter, we show the difference in our work which is strictly based on mechanism of replication for predictability and then prove the usage of replication for predictability with a wider range in real online system management.

Of all of prior works, the SCADS director is the most related system [29]. SCADS points out even small partitions under light workload can incur serious performance anomalies violating SLAs. It improves the SLAs by dynamically scaling the data into partitions as the load gets heavier and creates a duplicated copy for each of the partitions. We use the word duplicate to denote the server with exactly same data used in replication for predictability to distinguish from the word replica used in replication for throughput. By using the mechanism above, the SCADS director achieves 99.5% SLA for the EBate.com workload. However, we found the SCADS director had its constrains in two aspects:

(1) The target workload used in the SCADS director does not require an extremely high SLA (i.e. 99.99%), so only one duplicate is used in SCADS director. And they do not

provide detailed analysis on replication for predictability like whether the SLA can be further improved by adding more duplicates.

(2) The implementation of SCADS director is based on a relaxed-consistency guarantee which ensures them to get the read accesses being processed efficiently. But it also limits them to support only the read storage accesses. While there are some workloads requiring their write accesses to be reliable too as the workload Gridlab-D described in the Chapter 2.

Hence, SCADS cannot answer the question like "How can achieve a 99.99% SLA for the workload Gridlab-D by adding duplicates using replication for predictability?" Other system based on replication for predictability like Mantri [2] which uses replication for predictability to improve the reliability of Map-Reduce tasks also has the similar constrains as SCADS that only validates replication for prediction on add merely one duplicates.

Zoolander covers the constrain of prior works; it is able to precisely reason the improvement of SLA by adding many duplicates (up to 16) and its implementation (the RP Zookeeper) supports replication for predictability for a wider range of use (for both read and write storage accesses). Our model refers many prior models from queuing models [12, 25, 30] to machine-learned models [3, 8] to control theory [7, 19, 23]. These are all theories that are widely used in many successful productions.

Finally, our work focus on non-deterministic anomalies, which are more difficult to be found and reduced compared to deterministic anomalies, which have been well studied by other related work like EntomoModel [26] and IronModel[28].

Chapter 4: Zoolander

Zoolander is an analytical model based on replication to prediction, to predict SLA for a given replication strategy. Replication for predictability, which has been briefly discussed above, is a useful strategy to improve the SLA. In detail, replication for predictability uses duplicated servers to run the same access and chooses the first response as its result. As a result, by applying replication for predictability, N duplicates can mask up to N-1 anomalies and help reduce the response time of the system when the root causes of the SLA are anomalies other than workload access patterns.

Alternatively, it is also possible to use the N duplicates to run different access (by using a load-balance to forward requests into different servers), which can improve the throughput by N times compared to the throughput by applying replication for predictability. This strategy of replication that forces each request to be processed in just one cloud server for the purpose of adding number of requests that can be handled in a fix period of time is famously used in traditional static content workloads, called replication for throughput. However, when performance anomaly occurs, i.e. some storage accesses take much longer time to be processed than expected; replication for throughput may produce a worse performance then replication for predictability. Worse still, in networked storage system, which needs replication to improve its SLA, there can be many factors that cause performance anomalies, e.g. the OS scheduler, buffer sharing, or even a simple

misconfiguration. Hence, replication for throughput cannot necessarily improve the system's performance as expected.

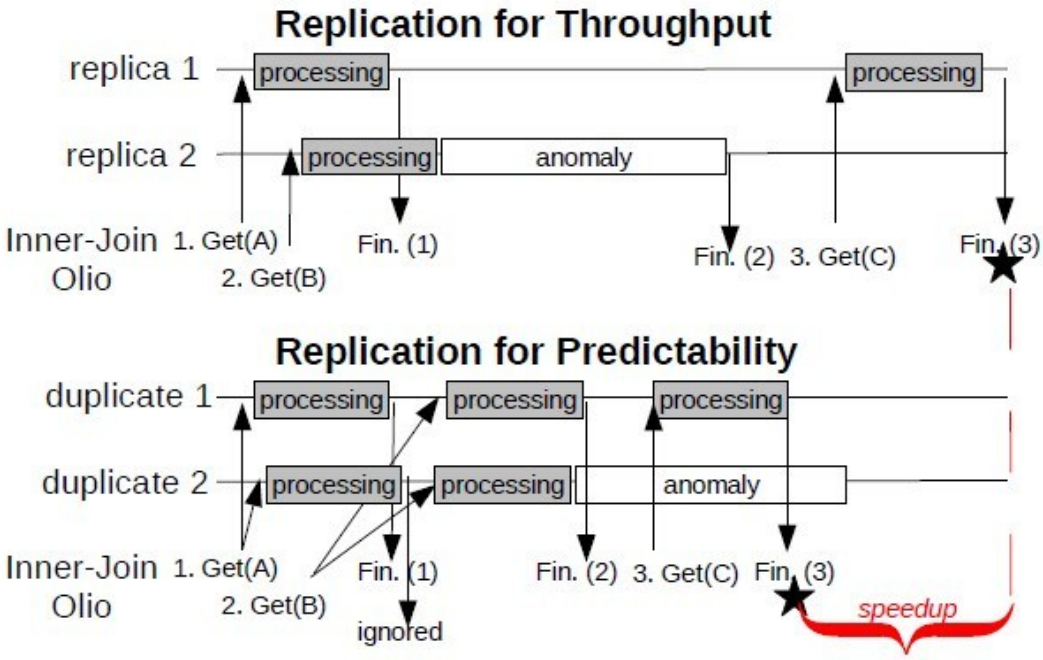

Figure 3: An example that only replication for predictability speeds up the inn-join olio workload when performance anomalies occur. Horizontal lines indicate the local time line of a node in cloud. a. The whole request is completed in the star mark.

Figure 3 depicts a comparison between replication for throughput and replication for predictability by running Inner-Join olio workload in a situation caused by a performance anomaly when replication for predictability improves the performance more than replication for throughput. Because of the trade-off between replication for throughput and replication predictability, and the large amount of existing research on the former one, we also need a method to quantify replication for predictability in order to guide

manager's choice when planning to use replication to improve SLA of the system. This is also an important reason for us to propose Zoolander.

4.1 Basic principle

Zoolander is designed to improve the system's SLA based on the strategy of replication predictability, so it also inherits the requirement of this strategy. Here are two basic principles of replication for predictability which must be sufficed, to ensure that Zoolander works correctly as expected:

(1) Duplicates must operate independently. This means duplicates mush have their own resource or virtual resource and storage accesses should not have any dependencies between any of the duplicates. So anytime one anomaly occurs in some duplicate, others duplicates won't be affected.

(2) The storage system must guarantee that every storage access is processed independently. Zoolander can only mask non-deterministic performance anomalies, which requires the storage system to guarantee that every of its storage accesses has the same possibility to be finished quickly. This is important because in some storage systems, their storage accesses will probably get delay after processing a fixed number of accesses or when processing a access on a fixed access pattern. This delay cannot be masked by Zoolander because all of the duplicates will necessarily get delay in the same time so none of them can return fast to mask the delay, and this is just a typical type of deterministic anomaly.

We will study these two principles latter. Here we will prove the consistency of our setup of Zoolander first to guarantee the correctness.

4.2 Consistency, Availability and Partition tolerance

According to the CAP Conjecture, network system can gain only two characters in data consistency, partition tolerance or availability by the cost of sacrificing the other. Traditional systems based on replication for throughput always sacrifice their availability to scale their data into several nodes without any overlap so each of these nodes can always keep the most up-to-date data which guarantees its consistency and it can reduce the slowdown from the increase of network in each nodes by scaling data into more servers, but once one node gets interrupted, the whole system cannot work which means that the availability is sacrificed. Some other systems [1, 4, 10, 29] may use partition tolerance to scale their throughput and some alternative replication to keep its availability but the consistency are always sacrificed.

In our Zoolander, since that every duplicates in Zoolander keeps exactly the same data, so that system is able to work properly as long as one of its duplicate is alive which guarantees a high availability. And we force all the clients to send their requests through a high-throughput broadcaster which keeps a FIFO (first in first out order) order queue and is connected to each of the duplicates. Then, we coerce a duplicate to be killed if it misses any access broadcasted from the broadcaster or fails to finish a storage access issued. By using this mechanism, none of the out dated nor conflict data can be returned in Zoolander, which guarantees the consistency. However, Zoolander cannot keep its

performance when network overhead increase, because it is designed for the predictability response both in SLA and data value instead of throughput, but this approach can sometime even exceed the performance of traditional replication for throughput in system with much non-deterministic performance anomalies, as shown in Figure 3. In sum, according to the CAP Conjecture, Zoolander has no guarantee on its throughput but ensures both high availability and solid consistency. We will further discuss the implementation of this design in detail in Chapter 5.

4.3 Studies on the Service Time

After discussing the tradeoff among availability, network tolerance and consistency in Zoolander, to prove that replication for predictability can help existing storage systems to improve their SLA, we did experiments on service time (i.e., processing time for a storage access) in some famously used systems to achieve a detailed study on the two principles described in Section 4.1. We did this study in our private cloud so that we could get more precious control and also repeat our experiment, to get a more accurate result.

4.3.1 System environment

Our private cloud is based on a 16-processor, 32-core Dell cluster operating at 2.66 GHz with 3MB L2 cache. Each of the virtual nodes is run in one core exclusively in order to get rid of L2 cache sharing which may cause some deterministic performance slowdown when the number of virtual nodes increased. Here we use User-Mode-

Linux(UML) [11] which is a port of Linux operating system running in user space of any X86 Linux system. The RedHat Linux with kernel level 2.6.18 is used as the operating system interface in each of the virtual machines. Then we constructed a serious of PERL scripts to help manage the virtual nodes in the cloud. The PERL scripts are based on the concept of Usher [18] which allows 4 operations including:

(1) Start predefined virtual machines on allocated server hardware

(2) Create and config the virtual network on the nodes running in the cloud;

(3) Expose public IP addresses and virtual names of all virtual machines;

(4) Stop a selected virtual machine;

Our design of the private cloud guarantees its infrastructure to be compatible with any other public clouds using X86 Linux instances, e.g., Amazon EC2.

At the beginning we chose Zookeeper as our storage substrate [15], which is a hash-table like key-value storage system retrieving its data by given keys. There are three reasons that motivate us to choses it:

(1) Zookeeper is used in production of Yahoo underling the Yahoo Message Broker and PNUTS. Hence studying on Zookeeper can provide us with more meaningful results that are able to help improve systems that are used in real world.

(2) Zookeeper uses the wait-free coordination, which eliminates the effect on the latency of storage accesses from the user workload computation. This guarantees the latency of storage accesses to reveal the actual processing of Zookeeper servers, reducing the variance of service time.

(3) Although Zookeeper uses the design of wait-free which is a relaxed consistency strategy compared to traditional lock based strategies, its enforcement of write-order consistency still helps it to maintain stronger consistency than many published key-value systems [9, 10].

(4) Zookeeper's key-value storage infrastructure provides fast access and loose structure that can help data-intensive services grow.

To simplify the notation, we use following terms to describe a Zookeeper setup in our thesis:

(1) A *Zookeeper group* denotes a group of cloud servers hosting the same version of data;

(2) A *Zookeeper node* denotes one server in a certain Zookeeper Group.

Zookeeper allows inconsistent read and read accesses are sent to only one Zookeeper node, which improves Zookeeper's throughput on read by N times where N is the number of Zookeeper nodes. On the other hand, write accesses must contact a majority of Zookeeper nodes within a group so the group may become unavailable if the communication among most of nodes broke.

4.3.2 Distribution of the Processing Time

To test the processing time of Zookeeper, we created an initialization script that took the IP addresses of all Zookeepers nodes within one group and automatically set up their configuration files. Then we did the experiment based on this focusing on all-read or all-write storage accesses each of which operated on data with size 2KB. And we set the number of keys to be limited to 1000 so that all the data can be fitted within the L2 cache.

For simplify, we sent storage accesses seriously with no concurrency according to a preset rate. Note that duplication in paper is performed at the group level.

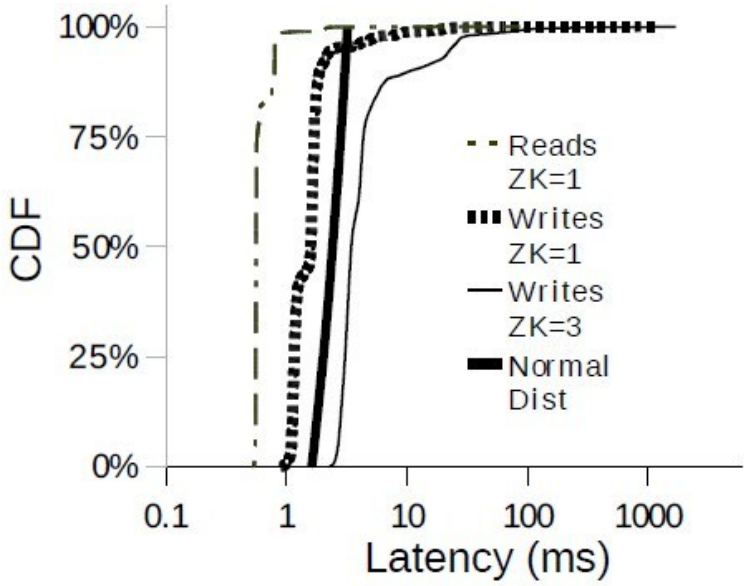

Figure 4: CDF of access latencies in Zookeeper for read and write access. The number of ZK depicts the number of nodes in a Zookeeper group.

And since read accesses are only processed through one node in Zookeeper, we did test on multiple nodes only for write. The result is shown in the Figure 4 above, as 4 cumulative distribution functions (CDF) for the latencies of write and read accesses in Zookeeper Group with 1 node and also the write accesses in Zookeeper Group with 3 nodes. As can be seen in the CDF, the processing time of fast responses follow in normal distribution with very low variance in all the experiments. But variances are much higher for the whole CDFs, with coefficients ranging from 1.5-8. To make it clear, we added an additional CDF which followed strictly in normal distribution with standard deviation

into Figure 4. The mean of the normal distribution CDF is 25% larger than the write latency in ZK = 1 (group with 1 node). Comparing the tail of all CDFs in Figure 4, we can see that the tails of all experiments overtake the normal distribution, although their means are smaller. And we found that their tails increased further as requests accessed more system resources: read accesses were much faster than write access and latencies of write to groups with one node were shorter than latencies of write to groups with three nodes. This is because write accesses to a Zookeeper group with one single node leads to local disk access that will not happen under processing of read accesses and the local resource management, i.e. handling I/O interrupt, may cause some anomalies. Moreover, write accesses to groups with more than one nodes cost some more latency to send messages among nodes in it for the guarantee of consistency. Finally, we can see the evidence that Zookeeper could only gain the SLA of 98.8%, 95.7%, 91.5% for read and write accesses to one node group and write accesses to three node group even when the bounder of latency were set to two times of the mean. Otherwise, we had to raise the latency bounder to 16X, 26X and 99X for the three types of accesses to achieve 99.99% SLAs, which is unacceptable. Besides Zookeeper, we did the similar experiment in MemcacheD too, which is a widely used key-value storage system based on key-value data structure similar to Zookeeper but is simpler and inconsistency. We got a CDF with a coefficient of variation of 1.9 and a SLA of 98.3% with 2X mean bounder. This suggests that local resource management may be a general cause of anomalies which happens in a lot of key-value storage system, not just Zookeeper.

4.3.3 Deterministic Anomalies vs. Nondeterministic Anomalies

We also explored the SLA for browsing requests, which output a CDF with much smaller efficiency of variant of 1.2, but its mean response time increased dramatically when workloads got heavier and its performance anomalies increased as well. Hence, we can conclude that for those complicated storage system, their performance anomalies are more workload-independent, which means that they could not get much benefit from replication for predictability according to the principle #2 described in Section 4.1.

On the other hand, we set an experiment that run the same workload in two duplicates in the same time, collected percentile of each storage access and plotted them into one graph, as shown in the Figure 5. If anomalies of Zookeeper are dependent, either the plot's upper left part or the bottom right will be empty. But the result shows that anomalies in Zookeeper are independent because every quartile of the plot is touched evenly. This proves that Zookeeper is a workload independent system and its anomalies are nondeterministic, which indicates that the SLA of Zookeeper satisfies the principle #2, which allows its SLA to be improved by applying replication for predictability.

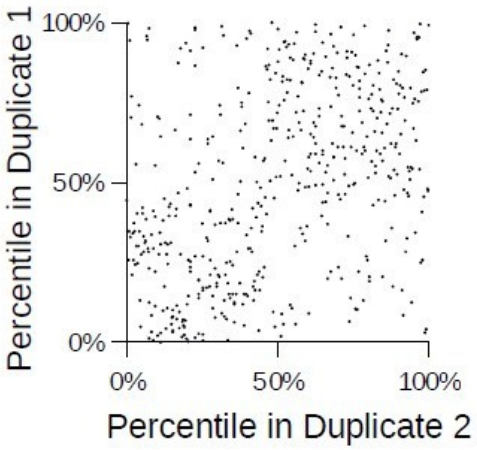

Figure 5: The proof of nondeterministic of anomaly in Zookeeper. Anomalies in one duplicates does not increase the possibilities of anomaly in the other

4.3.4 Queuing delay

In practical situation, queuing delay also slows down the response of storage systems, which also has to be taken into consideration. Here we set an experiment to test the queuing lengths of two running duplicates over time. In this experiment, the arrival rate of requests were set to be 500req/s with exponential distribution, so significant peak can still be caused although the peak throughput of each node was much larger (~2000req/s).

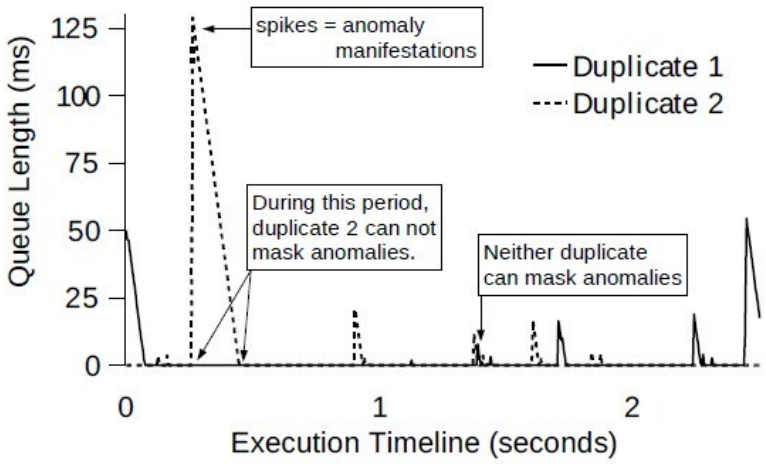

Figure 6: Waiting time of requests over time

The result is revealed in Figure 6. Spikes in this plot show that at some point of time when a performance anomaly happens, the subsequent request must have to wait until its former one to be finished. Even worse, if the queue grows long enough when anomalies keeps to happen for several times, one duplicate can totally be unable to mask the anomaly any more.

4.4 Model for SLA Prediction

Based on the analysis above in the Chapter 4, we propose the SLA prediction model of Zoolander. The model helps to characterize the effect from the nondeterministic performance anomalies for multiple duplicates running the same workloads. Notations used are shown in the Table 1, where the only output is $\hat{s}$ which is a percentile depicting the expected SLA and others are the inputs.

Table of notation	
Notation	Meaning
$\hat{s}$	Expected SLA
N	Number of duplicates
τ	Target latency bound
$\Phi_n(k)$	Percentile of service time in a duplicate n within latency below k
λ	Mean of the storage access rate
μ_{net}	Mean of the network latency (between duplicates and clients)
μ_n	Mean service time latency of duplicate n

Table 1: Notation of inputs and outputs of Zoolander

4.4.1 Basic Model

First we present the basic prediction model of SLA neglecting the network and queuing delay. Based on the two principles described above, since every duplicate runs the same workload with consistency guarantee, the duplicate that replies first can guarantee to return the correct data. So the possibility that all duplicates satisfy the SLA is just the possibility that the fastest duplicates return its response within the time constrain of the SLA. The probability can be computed by the follow formula:

$$\hat{s} = \sum_{n=0}^{N-1}[\Phi_n(\tau) * \prod_{i=0}^{n-1}(1-\Phi_i(\tau))]$$

Here, we denote duplicates from the number 0, so the tab of duplicates will be 0, 1, 2, 3 … When the number of duplicate is one, denoted as duplicate #0, the $\hat{s}$ will be $\Phi_0(\tau)$, which means the probability of getting service time latency below τ is just the probability of that only duplicate responds the accessed within the latency below τ .

Then, increasing the number of duplicate to be two, SLA violations happen when performance anomalies occur. Besides the situation that duplicate #0 responds fast within the latency below τ , accesses can also be responded fast by the whole system if the duplicate #1 masks the slow accesses in duplicate #0. Since in most cases resource maintained in each of the nodes in the cloud is the same, the percentiles of service time of all of the duplicates to finish processing a request within the latency below τ are also the same, meaning that their $\Phi_n(\tau)$ are the same too. Hence, we can get the $\hat{s}$ in this case to be $\Phi_n(\tau)$ plus $(1 - \Phi_n(\tau)) * \Phi_n(\tau)$ where n can be 0 or 1 and $(1 - \Phi_n(\tau))$ is just the probability of duplicate #0's performance anomalies to be masked by duplicate #1. We can achieve the same result by exchanging the tab of the two duplicates, so the order of duplicates will not actually affect the result.

Based on the analysis and condition above, recursively, we can conclude that every time adding a new duplicate, the $\hat{s}$ will be added by $(1- \Phi_n(\tau))^{N-1} * \Phi_n(\tau)$. So the formula above can be derived to a simpler form given that processing time distribution in each of the duplicates are the same, as shown below:

$$\hat{s} = \sum_{n=0} \Phi_n(\tau) * (1 - \Phi_n(\tau))^{n-1} = 1 - (1 - \Phi_n(\tau))^{N-1}$$

4.4.2 Queuing Delay & Network Delay

In real situation, to be practical, packets in the cloud cannot reaches their destination immediately because the bandwidth in the cloud is always finite. So the network latency should not be eliminated when designing the SLA model. More important, in real case,

since workload running in each of the duplicate is the same which decides their access patterns to also be the same, and the processing rate of each of the duplicates is finite, queuing delay will probably be caused when access rate increases. We can see the effect of queuing delay and the performance anomalies it causes in the Figure 6. Therefore, we must take these two kinds of latency into our model too, to make the SLA prediction reflect the real client's perceived latency, which consists of processing time, queuing delay and network latency.

We found the best way to fairly add the network latency and queuing delay is to deduct them from the targeted latency bound (τ), which requires all duplicates to reduce the expected SLA in proportion to the expected queuing delay and network latency as well. So can get the τ of each of the duplicates as formula below:

$$\tau_n = \tau - \mu_{net} - \mu_{queue}$$

Then the problem here is how to get the value of μ_{net} and μ_{queue}. We can simply get μ_{net} by collecting the distribution of the network latency cost by sending a packet from one node to another in the cloud. But to get the μ_{queue}, it is more complicated.

We studied prior theories of queuing delay to evaluate the effect of queuing delay in order to add it into our SLA prediction model. To help illustrate, we take Zookeeper as an example whose processing time has been well analyzed above. M/G/1 queuing model can be used here to help analysis which requires the inter arrival rate to be exponential distributed and the processing time to be general distributed. We then get the equation of computing μ_{queue} by the applying M/G/1 model, as shown in the formula below:

$$\mu_{queue} = \frac{1+C_v^2}{2} * \frac{\rho}{1-\rho} * \mu_n$$

The ρ denotes the function of system utilization and its distribution variance is denotes as C_v^2. We can use the mean processing time to divide the mean arrival rate to get the utilization. And the τ here is different in each of the Zookeeper duplicate (group).So, the SLA prediction model can finally be got as the equation below:

$$\hat{s} = \sum_{n=0}^{N-1}[\Phi_n(\tau_n) * \prod_{i=0}^{n-1}(1-\Phi_i(\tau_i))]$$

The model can be simplified by using M/M/1 model, which can eliminate the coefficient variance (denoted as C_v^2 above) for some storage system whose processing time is exponential distributed. And the model can also become more general by applying the G/G/1 model because it does not have the constraint of the M/G/1 model that needs the inter-arrival rate to be exponential distributed. But some constraint is necessary for the G/G/1 model to be applied. We leave the further analysis of the queuing delay as our future work.

4.5 Model-Driven Analysis

Based on the SLA prediction model, we can produce a more comprehensive tradeoff analysis on replication for prediction and replication for throughput. First we computed the mean of processing time from the CDF of the latency of write access to a single node Zookeeper duplicate shown in Figure 4 as “write, ZK=1”. The targeted bounded latency here was set to be 5ms, which was around two times larger than the mean service time

latency. Then we varied the utilization of the system into 3 different number as 1%, 5%, 15% by changing its arrival rate of the storage accesses for the configurations using replication for predictability. The similar work was done to configurations using replication for throughput too by changing the storage accesses arrival rate to each of the replication and allowing inconsistent write. After that, we put all of the data above as input into Zoolander and plotted its output.

The result is shown in the Figure 7, as the percentiles of service time latency below 5ms in configuration using replication for predictability or replication for throughput with different number of duplicates.

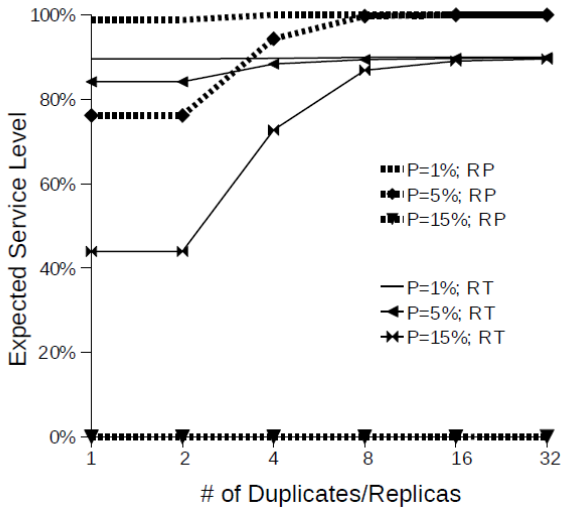

Figure 7: The SLA predicted by Zoolander for different configuration based on replication for predictability and replication for throughput

As can be seen in the Figure 7, when the utilization is heavy (15%), the configuration using replication for predictability produces really bad SLA (only 2%) and the SLA is improved little by adding the number of duplicates. This is caused by the

queuing delay: once a performance anomaly occurs in some duplicate, one or more accesses will be delayed and the queuing delay will be produced. Even worse, this queuing delay will probably keep increasing infinitely since the arrival rate is very high, which means that one duplicate may not be able to mask any performance anomalies of the other duplicates once a performance anomaly happens in it. This is also why the SLA still does not increase when adding duplicates. However, the situation above can be well avoided by using replication for throughput, which distributes accesses into different replicas so one storage access can be issued to another replica when some replica gets performance anomalies. The queuing delay is mitigated by using replication for throughput and the SLA grows to 90^{th} percentile when the number of replicas is increased to 32.

Then, in the situation that the utilization is smaller as 5%, the SLA of replication for throughput still exceeds that of replication for predictability when the number of duplicates (or replicas) is small. But then the SLA of replication for predictability overtakes since the point when the number of duplicates (or replicas) grows to 4. This is because that the queuing delay in every duplicate in this situation is finite this time because the processing rate is much larger than the arrival rate, so one duplicate can evenly finish all of its delayed access caused by the performance anomalies and masks the anomalies happening in the other duplicates, which improves the SLA. More anomalies can be masked by adding duplicates in this situation so the SLA can keep increasing to near 100^{th} percentile (99.99%). In the other hand, although the queuing delay can be mitigated, each of the replicas has its own maximum SLA, i.e., if the SLA of

one replica is 90%, the replica cannot finish 90% of its storage accesses shorter than 5ms; anomalies happens when processing the rest of the accesses. So actually the SLA of every replica becomes the bottleneck, and the SLA cannot be increased larger than 90% in this situation.

Afterwards, when system utilization reduces to 1%, approximately no queuing delay will be produced in this situation. So the SLA of configuration using replication for predictability exceeds that of configuration using replication for throughput at the very beginning. This is also because the bottleneck of the SLA of each of the replicas.

Based on the Zoolander, we provide a comprehensive comparison in the tradeoff between replication for prediction and replication for throughput. We can see that replication for throughput is more suitable for those systems with high utilization while replication for prediction can improve the system's SLA to an extremely high level in systems where queuing delay is not significant. Based on the specific environment of the system, managers can use Zoolander's prediction model to help them find the proper strategy of applying replication, to achieve the best performance.

Chapter 5: RP Zookeeper

As be pointed by the prior work as SCADS director [29], replication for prediction can be built on top of the existing systems while the SCADS director only supports the read accesses. We implemented our RP Zookeeper to validate our Zoolander SLA prediction model. RP Zookeeper is an extension of Zookeeper which implements the group level replication for prediction, with its guarantee of consistency and high SLA for both read and write accesses.

In the RP Zookeeper, write accesses are sent through the message repeater which maintains the FIFO order; while read accesses are directly sent to Zookeeper groups for the purpose of best performance. In every leader node of Zookeeper groups, there is a Zookeeper proxy that cooperates with the message repeater to ensure the consistency across all the groups. The results of the storage accesses are responded by the Zookeeper proxies directly to clients, which can save a network trip.

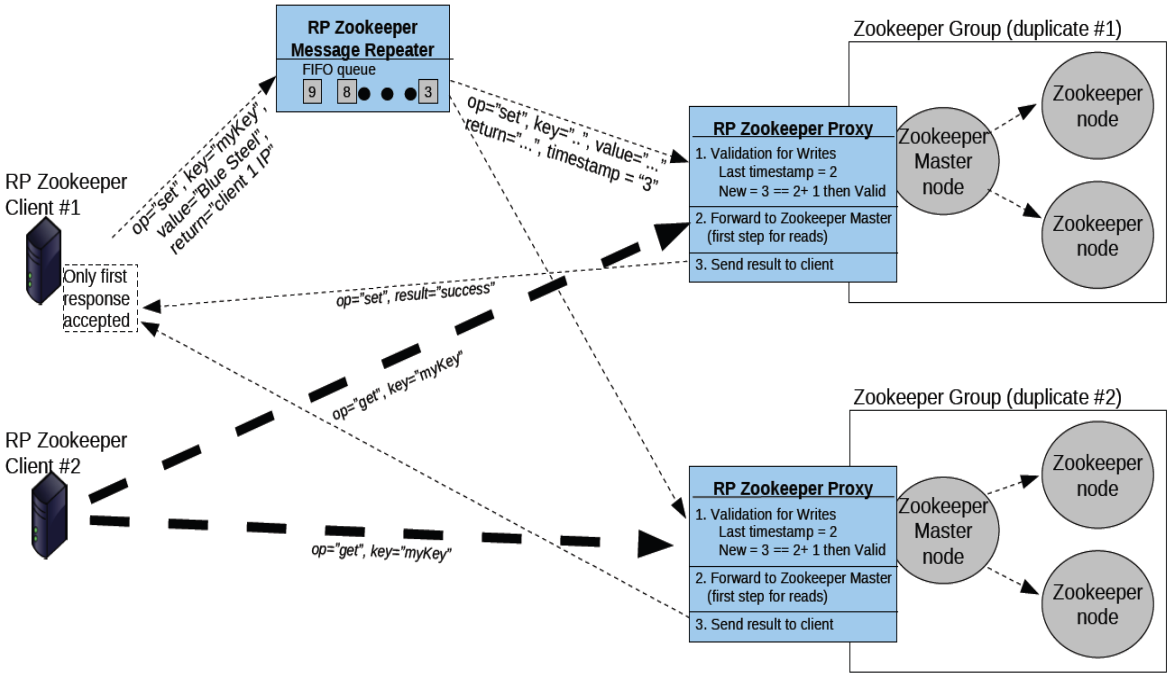

Figure 8: The components of RP Zookeeper. Replication for predictability occurs at the group level, and each of the groups consists of 3 nodes. Here the write accesses are sent to the multicast switch to be broadcast and the read accesses are sent directly to the Zookeeper groups.

In this chapter, we will first present the reason of choosing Zookeeper as the base to work on, and then will describe each components of RP Zookeeper in detail including the client, the Zookeeper proxy and the message repeater. We believe that the solution of RP Zookeeper can be borrowed to help improve the SLA in other storage systems which also suffer from performance anomalies such like the MemcacheD as analyzed in Chapter 4.

5.1 The reason of choosing Zookeeper

We choose Zookeeper as our storage system because of two typical implementation decision of Zookeeper. The first sound advantage of using Zookeeper is its enforcement of the FIFO order for its write accesses which forces all the write accesses to the leader node to be ordered and then to be broadcasted to slave nodes. One write access in Zookeeper cannot be committed until the majority of nodes updating their snapshot, which is on the Zookeeper's Zab Atomic Broadcast protocol [16, 20]. This implementation of Zookeeper allows to further improve the availability of Zoolander by just adding nodes to each of the Zookeeper groups without any concerning of the consistency; only thing we should do is just maintaining the consistency among all of the groups. Or if we choose other storage systems like MemcacheD without this enforcement, we must do extra work to maintain the consistency inside every group when adding nodes.

The other important reason motivates us to choose Zookeeper is its wait-free synchronization primitive based on the lock free implementation supported by its centralize FIFO order maintenance described above. In pure lock-based strong consistent systems, once a lock-hogging caused by some duplicate, the other duplicates will have to interact to maintain the consistency, which actually produces a deterministic anomaly violating the Zoolander's first principle discussed in Chapter 4. Some work has been proposed recently to avoid the serious effect due to the lock-hogging like using lease to thwart lock hoggers which is applied by Chubby and Farsite[11]. But even the leases can only bound the amount of time that one client delaying others. While the lock free design used in Zookeeper completely neglects this kind of resource sharing among nodes (i.e.

the locks) and avoids the slowdown caused by waiting for the lock release which actually eliminates the performance anomalies caused by locks.

5.2 Implementation

5.2.1 Message Repeater

The message repeater is the middle component of RP Zookeeper which sits between the clients and Zookeeper proxies. All of the write accesses will be broadcasted through it, to the Zookeeper proxies running in the leader node in each of the groups. We implement this message repeater in C programming language and set it to run in a separate server to achieve the maximum performance. The message repeater maintains one FIFO queue for all accesses which are sent through it and forwards them to Zookeeper proxies in the same order. But the accesses from different clients are not interleaved. This operation above guarantees that the write accesses to be linearized and the FIFO order in client level similar to Zookeeper. Specially, every time when the message repeater forwards a storage access, it attach it with a logical timestamp reflecting the global order of that access, then increases the timestamp by one after broadcasting the access.

5.2.2 Zookeeper Proxy

Zookeeper proxy is running in the leader node of each group and is also written in C programming language. It takes responsibility of storage access processing and timestamp validating. The proxy uses its main thread to listen on all the packets which may be from clients for read accesses or from the message repeater for write accesses.

Here the packets are defined by a struct including 4 fields as the type of access (read or write), address of client to respond, a key-value pare (denoting the key to read or to write with the written value), and a timestamp which is attached by the message repeater only for write access.

Besides, the Zookeeper proxy also contains a second thread that unpacks the packets and validates their timestamp by comparing with the most current timestamp. One timestamp is valid if it is one digit larger than the most current one and its attaching storage access in the packet will be processed. Otherwise, if the validation of the timestamp fails, the proxy will kill the duplicate for the potential failure that may have happened when transferring data in the network. A duplicate may also be killed by the Zookeeper proxy if local failure (e.g. disk error or the processing time exceed a timeout defined by the manager) happens when processing the storage access, which will be found by examining the output of the local Zookeeper group.

To make our implementation of RP Zookeeper more general, we also added an extra operation called write-all-duplicate which is similar to the synchronize request in Zookeeper. Write-all-duplicate will not be completed until all the duplicates finish the write access successfully. This operate can be issued by clients to guarantee their subsequent read to any of the duplicates to return an up-to-date data.

5.2.3 Client

The Client in RP Zookeeper just uses the library of RP Zookeeper. It packs all the storage accesses to duplicates into the packets with the format described above and sends them to Zookeeper proxies (for read) or the message repeater (for write). It also uses a second thread to listen to packets returned by Zookeeper proxies and detect inconsistencies if responses ever differ.

5.3 System overhead

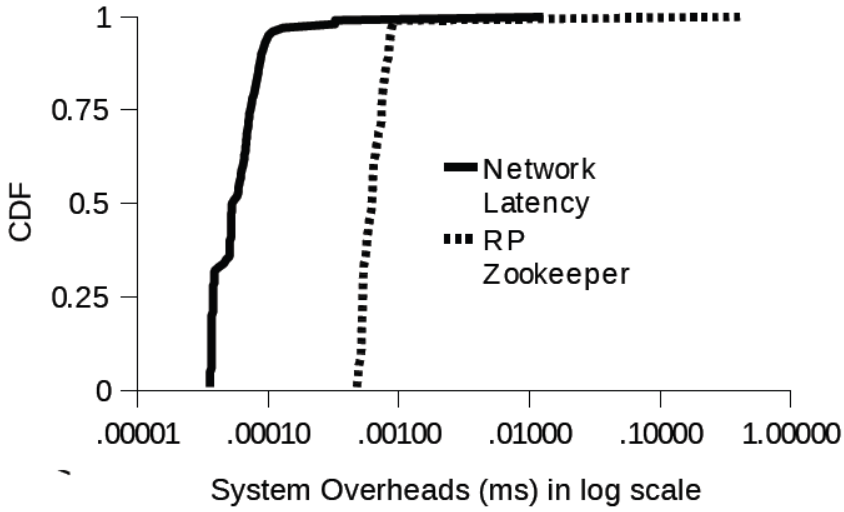

Figure 9: CDF of the network latency and whole system overhead of RP Zookeeper

To ensure the correctness of RP Zookeeper, its system overhead, which includes two parts as network latency and the extra processing time introduced by the system of RP Zookeeper must be light and reliable. We set two experiments to verify this requirement as shown in Figure 9: it turns out that our system fits the requirement well because the mean system overhead is very small (< 0.001ms) and its latency is less than 0.5ms (which

is much less than the Zookeeper processing time) even in the worst cast when network outlier occurs, which proves the reliability of RP Zookeeper.

5.4 Validation of Zoolander using on RP Zookeeper

We deployed our RP Zookeeper in our private cloud which is described above in the Chapter 4, to validate our Zoolander SLA prediction model. First we used a single node Zookeeper duplicate, sent it with 10,000 write accesses and collected its latency distribution as the input of Zoolander. The targeted latency bound was set to 5ms here which was the 90^{th} percentile value of the service time latency distribution (also be used as a Zoolander input).

We began this experiment by issuing write accesses to the system initially with one duplicate then added the number of duplicates by double each time with same write accesses issued, to see the SLA achieve. The result is plotted as shown the in the Figure 10. We can see that the SLA increase when the number of duplicates increases as predicted by the Zoolander. The SLA can reach 99.99% when the number of duplicates added to 16. Also we can see that the absolute error (i.e. differ between the observed SLA and the predicted one) is very low (even below 0.002 in all cases). This reveals that the prediction of the Zoolander is very actuate.

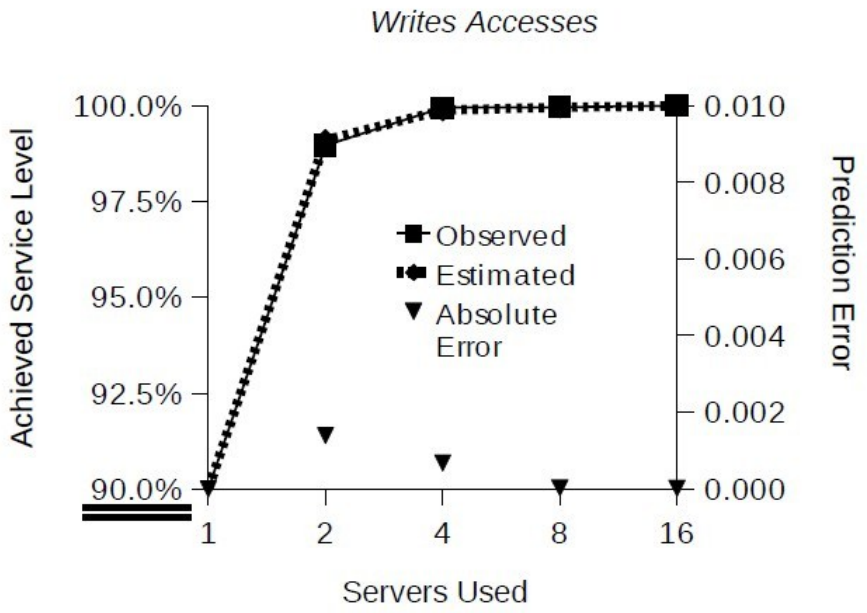

Figure 10: Result of the SLA observed by in the RP Zookeeper and predicted by Zoolander's prediction model.

Then we set the second experiment to see if the prediction of Zoolander that each percentiles of the SLA can be improved by adding the number of duplicate. In the experiment, we started with a RP Zookeeper configuration with 8 duplicates in it, and changed the targeted latency bound to vary from 75th to 99.5th then we compared the differ between actual SLAs we observed and those predicted by the Zoolander. As can be seen in the result shown in the Figure 11, the absolute error is still very low no matter which percentile is chosen. Figure 11 also depicts a strong gain for latency bound after 96th percentile which means the RP Zookeeper can still further improve the SLA as predicted by Zoolander by adding duplicates when start latency bound is set to an extreme high level. For example, if setting the targeted latency to 99% of the CDF of the

single node duplication, the SLA can be improved to a higher level (99.991%, which is 2 magnitude larger than 99%) by using 4 duplicates.

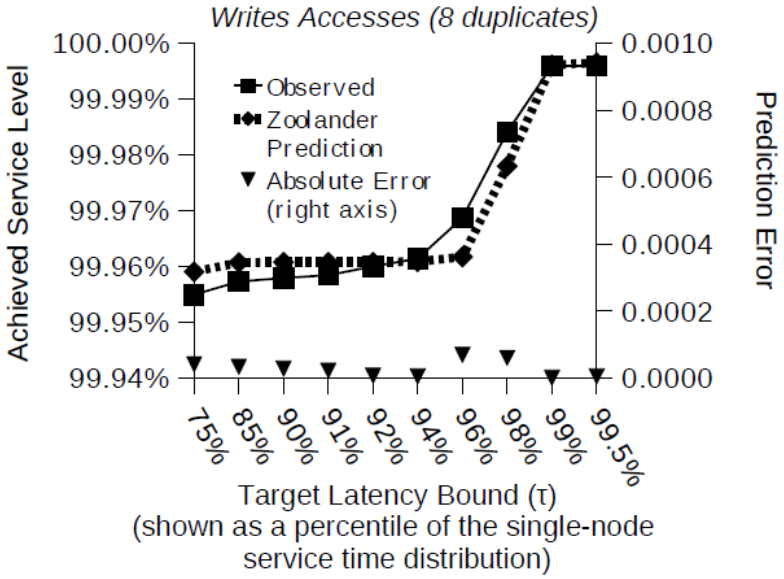

Figure 11: The absolutely errors of the prediction of Zoolander in different SLA

We also did the similar experiment to diverse workloads. Here we plot our result of two experiment as the form of numbers of nines achieved under read accesses and under larger Zookeeper group size (meaning that each of the groups contains more than one nodes), as shown in Figure 12and Figure 13. We focus the number of nines to be achieved because it is always the practical metric for SLA. The Zoolander's prediction is still very accurate and the error is less than 0.001.

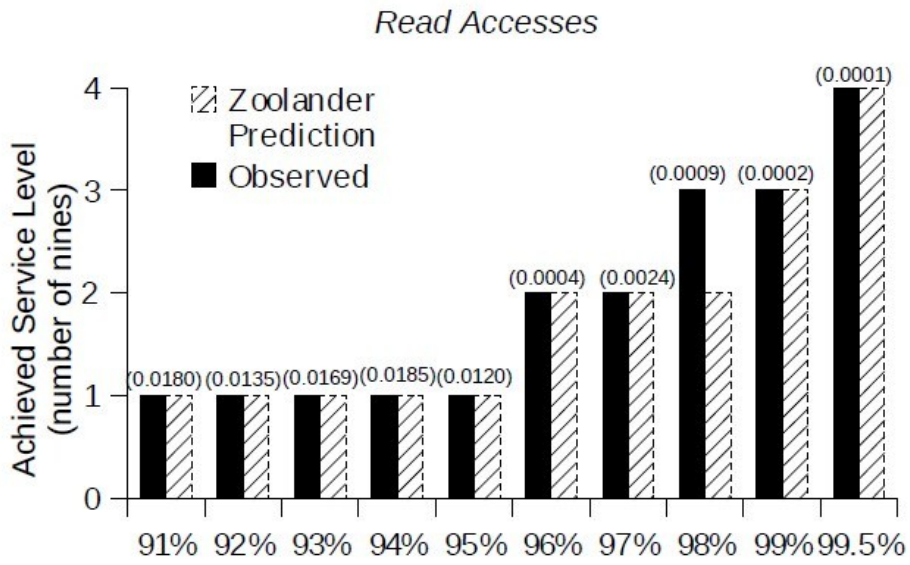

Figure 12: SLA achieved for only read accesses.

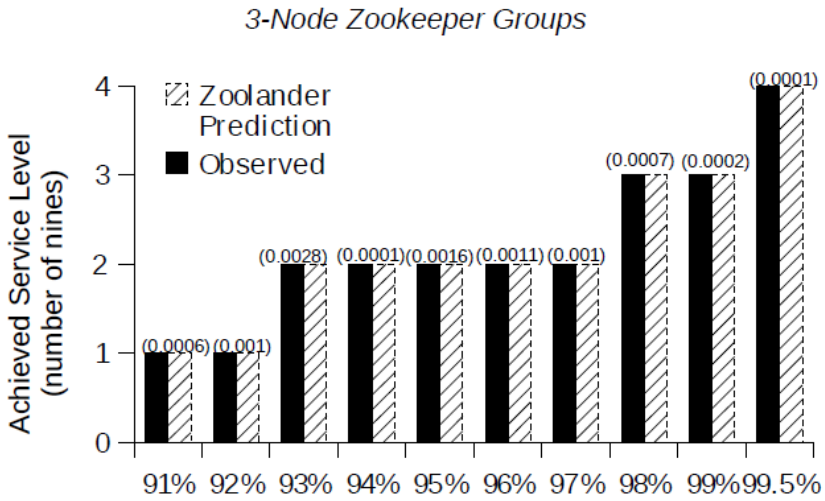

Figure 13: SLA achieved by using 3-node Zookeeper duplicate

Specially, Figure 12 shows that the read can be finished in our private cloud extremely fast (less than one microsecond). So it may happen that sometime read accesses get responded before they get broadcasted by the message repeater to all the

duplicates. And the Figure 13 shows that the SLA decreases when the group size increased. This is because the atomic broadcasting mechanism of Zookeeper that forces the nodes to communicate which produces performance anomalies. But not matter how we changed the workload, Zoolander continued to be able to produce a reliable and accurate prediction.

Finally, to make our validation more solid, we did the similar test in heterogeneous platforms. Recall that all of the experiments we did were run in our private cloud, so we were able to change the virtual environment in it. For example, we have done the experiment that changed to allow the L2 cache sharing in our virtual machines and created a new CDF for the single node duplicate. As a result Zoolander still predicted SLA accurately with less than 0.0002 errors. Then we changed the networking substrate in our cloud to be based on a user-level SLIRP device. This time the prediction of Zoolander were even more accurate with number of errors smaller than 0.0001.

Chapter 6 Zoolander in System Management

In this chapter, we present how the Zoolander can be used in the real system management, which means to get a highest SLA by using minimum resource. The challenge of improve the SLA in system management by using Replication can be concluded into the two questions below:

(1) Given fixed number of nodes, how to apply replication strategies to achieve the highest SLA for a certain workload?

(2) Given an expected SLA, how to achieve that target by using the minimum number of nodes?

We did our experiment here based on the two workload described in the Chapter 2, which are Inner-Join Olio and Gridlab-D.

The first experiment is based on the workload Inner-Join Olio. First we created the data of Inner-Join Olio in our RP Zookeeper by creating two tables as described below:

(1) A table stores users' information, which takes the username as the key and other information about the user (e.g. First name, Last name ..) as the value;

(2) An events' detail table that list all the detail of every event. This table takes the eventid as its key and other detail as its value.

We assume that each one Olio request:

SELECT username

From Person INNER JOIN eventandperson

ON eventandperson.eventid = 100 and

person.firstname = ‘A%” ;

is related to 30 events and 20 users in our storage system. Hence every time when issuing an inner join olio request, 50 parallel storage accesses (i.e. 30 reads to the event table first then 20 reads to the user table with a barrier between them) will be sent to our storage system with exponential distributed inter-arrival rate. Then, we allocated 4 nodes in our private cloud to config them with different replication strategy and see the final SLA them can achieve. Here we based on these 4 nodes, we can have 3 replication choices: first, all of the 4 nodes can use traditional replication for throughput; second, all of the 4 nodes can use replication for predictability; third, 2 of the nodes can use replication for throughput while the others can use replication for predictability (i.e. 2 replicas and 2 duplicates). The system utilization in this experiment is set to 8%.

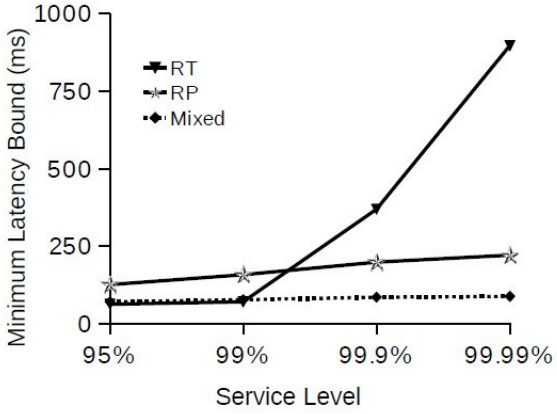

Figure 14: The SLA of the Inner-Join Olio workload achieved by using different replication strategy.

The result of the experiment is plotted in Figure14, where we can see that the mix of replication for predictability and replication for throughput achieves the best SLA because it is able to use the minimum latency bound in the majority of the 4 levels of percentile (3/4) compared to the other two replication configurations, but the peak throughput can be double by using this mixed replication configuration.

Then in the second experiment, we replayed the capture of the Gridlab-D workload by issuing 4 reads then 3 write for each of the simulation. We let these accesses to arrive in batches to reflect its intensive workload at the beginning and end of a simulation event. So here the G/G/1 model can be more suitable, as discussed in the Chapter 4. Then we did the similar experiment as the one did base on Inner-Join Olio workload, to see the highest SLA that can be achieved by using different replication strategies.

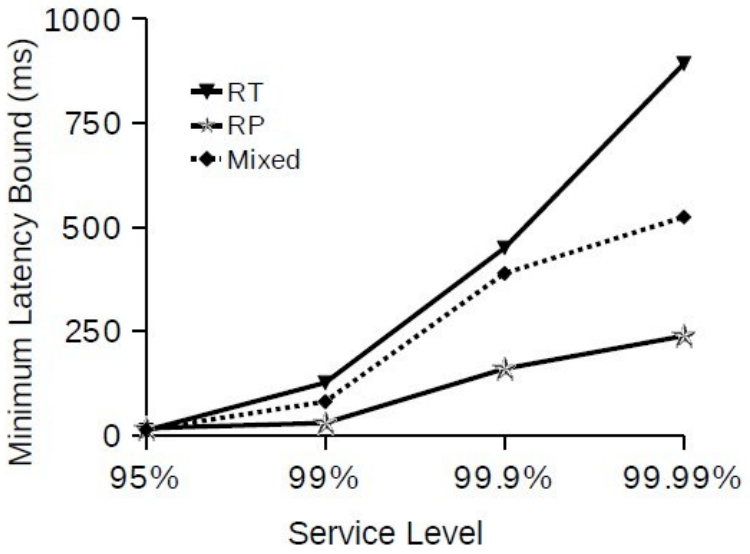

Figure 15: The SLA of the Gridlab-D workload achieved by using different replication strategy.

We can see the result as shown in the Figure 15. Here because of the heavy write workload, more duplicates are needed to a high SLA. So it is better to apply replication for predictability to all of the nodes.

Those two experiments above show examples about the way to find the answer of the question 1. Moreover, we did another experiment toward the second question that to find the minimum number of duplicates to achieve the expected SLA. To simplify, here we only did the comparison between the configuration that applied replication for throughput to all the nodes and the configuration that used a mix of replication for throughput and replication for predictability. We set the targeted SLA to be: 99% of all the accesses must finish within the latency below 10ms, and added 2 nodes per time to see the maximum throughput that can be support by that system.

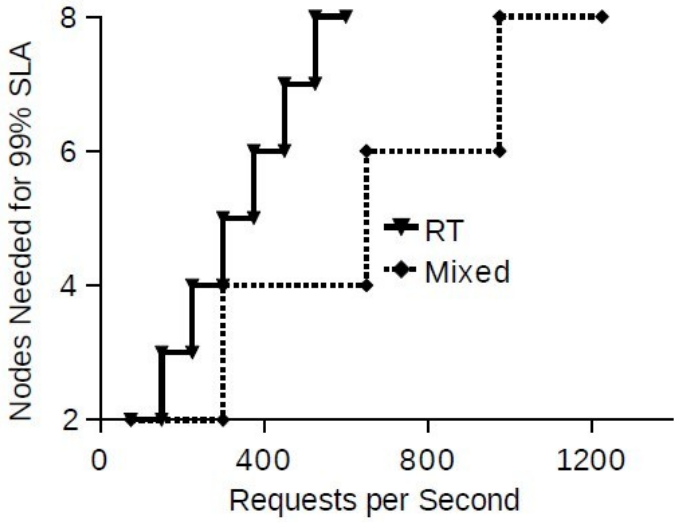

Figure 16: Maximum throughput that can be supported by using different replication strategy with different number of nodes

Figure 16 depicts the result of this experiment: we can see that using a mixed replication configuration can support larger throughput than only using replication for throughput. This is because although replication for predictability does not improve the throughput, it improves the "goodput" which is the number of accesses that can be processed with very reliable performance guarantee. And it can also be seen in this figure that with different number of access rate, the number of duplicates needed to achieve a target SLA is different, we can use Zoolander to get the best mixed strategy to achieve the best tradeoff.

Chapter 7 Conclusion

We studied comprehensively on replication for predictability here in this thesis, which is a mechanism that reduces the non-deterministic anomalies to help improve the SLA of networked systems. This meets the demand of new data-intensive workload that calls for storage systems with high SLAs to achieve the same performance as traditional static-content workloads. We analyzed in depth of the reason of this demand by studying two typical data-intensive workloads which are Inner-Join Olio and Gridlab-D. Then we proposed the Zoolander, which is a SLA prediction model that takes all the factors may affect the client's perceived latency including the network delay, the processing latency and the queuing delay into consideration and produces an accurate prediction of the SLA. Besides the core contribution of Zoolander, we implemented the RP Zookeeper on top of the Apache Zookeeper, which is a storage system supporting replication for predictability for a wider range of use (both the read and write are supported) compared to the SCADS director. Based on the Zoolander and RP Zookeeper we showed examples of using the Zoolander's prediction model to pick the best strategy of replication that achieves that highest SLA by using the minimum resource. Our results have two significant impacts. First, we proved that replication for predictability can scale SLA by increasing the number of duplicates, but this improvement must be based on the situation that both of the queuing delay and network latency are small. Second, by the prediction model of

Zoolander, we provided a general way to system managers to decide between replication for predictability and replication for throughput across different workload and system environment.

References:

[1] Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *In USENIX Symp. on Operating Systems Design and Implementation*, 2002

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. *In USENIX Symp. on Operating Systems Design and Implementation*, 2010.

[3] E. Anderson. Simple table-based modeling of storage devices.

[4] E. Brewer. Towards robust distributed systems (abstract). *In Symposium on Principles of Distributed Computing*, 2000.

[5] M. Burrows. The chubby lock service for looselycoupled distributed systems. *In Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI ’06, 2006.

[6] D. Chassin, K. Schneider, and C. Gerkensmeyer. In Transmission and Distribution Conference and Exposition.

[7] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. *In Autonomic Computing, 2006. ICAC ’06. IEEE International Conference*, 2006.

[8] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. *In USENIX Symp. on Operating Systems Design and Implementation*, pages 231–244, Dec. 2004.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. arno Jacobsen, N. Puz, D.Weaver, and R. Yerneni. Pnuts: Yahoo!s hosted data serving platform. *In VLDB*, 2008.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazons highly available key-value store. *In ACM Symp. On Operating Systems Principles*, 2007.

[11] J. Dike. User-mode linux.

[12] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. *In USITS*, Mar. 2003.

[13] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partitiontolerant web services. *SIGACT News*, 33, 2002.

[14] J. Gray. Transaction Processing: Concepts and Techniques. 1993.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internetscale systems. *In USENIX*, 2010.

[16] F. P. Junqueira and B. C. Reed. Zab: A practical totally ordered broadcast protocol. *In Brief Announcement in DISC*, 2009.

[17] C. A. Lee. A perspective on scientific cloud computing. *In Science Cloud Workshop colocated with the Symposium on High Performance Distributed Computing*, 2010.

[18] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. I*n Proceedings of the 21st Large Installation System Administration Conference (LISA)*, November 2007.

[19] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. *In EuroSys Conf.,* 2007.

[20] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. *In Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, 2008.

[21] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: managing server clusters on intermittent power. *In Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011.

[22] K. Shen, C. Stewart, C. Li, and X. Li. Referencedriven performance anomaly identification. *In ACM Int'l Conf. on Measurement and Modeling of Computer Systems*, 2009.

[23] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based Internet services. *In USENIX Symp. on Operating Systems Design and Implementation*, Dec. 2002.

[24] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson. Cloudstone: Multiplatform, multi-language benchmark and measurement tools for web 2.0. *In Workshop on Cloud Computing*, 2008.

[25] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In USENIX Symp. *On Networked Systems Design and Implementation*, May 2005.

[26] C. Stewart, K. Shen, A. Iyengar, and J. Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. *In IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2010.

[27] M. Stonebraker. Stonebraker on data warehouses. *Communcations of the ACM, 2011*.

[28] E. Thereska and G. R. Ganger. IRONModel: Robust performance models in the wild. *In ACM Int'l Conf. on Measurement and Modeling of Computer Systems*, Annapolis, MD, June 2008.

[29] B. Trushkowsky, P. Bodk, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. *In USENIX Conf. on File and Storage Technologies*, 2011.

[30] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. *In ACM Int'l Conf. on Measurement and Modeling of Computer Systems*, Banff, Canada, 2005.

[31] F. Uyeda, D. Gupta, A. Vahdat, and G. Varghese. Grassroots: Socially-driven web sites for the masses. *In Workshop on Online Social Networks (WOSN)*, Aug. 2009.