Zoolander: Modelling and Managing Replication for Predictability

Daiyi Yang and Christopher Stewart
The Ohio State University

Abstract

Social networking and scientific computing workloads access networked storage much more frequently than traditional, e-commerce workloads. In these workloads, issuing storage accesses in parallel offers speedup, as long as the slowest parallel access is fast. This paper studies replication for predictability, an approach to speed up slow storage accesses by running the same workload on duplicate servers and using the results from the fastest server. We created Zoolander, an analytic model that predicts the percentage of accesses that will complete quickly— i.e., a service level. Zoolander considers the effects of replication strategies, heavy tails in access time distributions, and queuing delays. We validated Zoolander against RP Zookeeper, an enhanced Zookeeper Coordination Service that supports replication for predictability. Zoolander was precise, predicting achieved service levels within 0.002 across diverse workloads and platforms. We used Zoolander to manage service levels of two parallel, data-intensive workloads running on RP Zookeeper. Exploiting replication for predictability, we achieved speedups of 373% and reduced the cloud servers needed by 50%.

1 Introduction

Service level agreements (SLAs) set bounds on the percieved latency for networked storage accesses at different percentile levels. For example, a traditional, storage SLA may read, "99% of all storage accesses should complete within 300ms provided peak load is less than 500 accesses per second [10, 27, 30]." The *service-level* portion in the above SLA (i.e., 99% of all accesses) may suffice for static content and e-commerce workloads that nest only a few storage accesses within a single user request. However, data-intensive workloads like social networking and cloud-based scientific computing may nest many parallel, storage accesses within a user request. Response times for such workloads depend on the slowest parallel storage access— not the mean. These workloads need storage systems that can provide service levels of several nines (e.g., 99.99% of all accesses) while maintaining low latency.

Partitioning and replication are widely used to manage service levels in traditional storage workloads [10, 14]. However, parallel, data-intensive workloads may demand stronger service levels than small partitions under light workloads can achieve, creating a need for additional approaches. Recently researchers have begun reigning in performance anomalies, i.e., making access times follow a normal distribution with low variance. Such research has taken two approaches: 1) debugging the underlying root causes of anomalies [23] and 2) avoiding anomalies via online management [2, 27, 30]. In very recent works, Trushkowsky et al. [30] and Ananthanarayanan et al. [2] used *replication for predictability* to avoid anomalies in their systems. Specifically, the SCADS manager [30] deploys two copies of the exact same storage partition, sending reads to both copies in parallel. The copy that replies first (normally not an anomaly) provides the result. This approach, called replication for predictability, is unlike traditional replication for throughput. Here, replication masks nondeterministic performance anomalies. It reduces variance but does not improve throughput. The prior work [2, 30] used replication for predictability only sparingly with conservative, ad-hoc goals. For example, in both works, the approach was limited to no more than two copies and supported only read accesses. Neither work quantified the expected performance gains. This work studies replication for predictability in greater detail by 1) modelling its benefits (at scale) and 2) using it to manage service levels for the read and write storage accesses of data-intensive, parallel services.

We present Zoolander, an analytic model that predicts service levels under replication for predictability. Zoolander accepts the following inputs: 1) a target latency bound, 2) service-time distributions, 3) storage access patterns, and 4) the network latency distribution. Zoolander outputs the percentage of accesses with response time below the target latency (i.e., the service level). Online management can query Zoolander with what-if questions about the service level under hypothetical changes, e.g., what will the service level become if the request arrival rate doubles or how many duplicates are needed if the SLA is strengthened to 99.99%?

We validated Zoolander by adding replication for predictability to the Zookeeper Coordination Service [15], creating a storage system that we call RP Zookeeper. We chose Zookeeper for three reasons. First, it is used in production at Yahoo where it underlies the Yahoo Message Broker and PNUTS. Second, Zookeeper implements wait-free coordination as a service, meaning the latency of any storage access depends only on Zookeeper servers, not on the computation of user workloads. Such wait-free coordination reduces the variance of service times within Zookeeper. Our third reason for choosing Zookeeper is that it enforces write-order consistency that is stronger than many published key-value systems [9, 10]. RP Zookeeper maintains these relatively strong consistency guarantees. Viewed as a case study, RP Zookeeper shows that replication for predictability can support stringent consistency. The name Zoolander, for a service-level model, comes from a popular movie about fashion models and plays on the first syllable of Zookeeper.

Zoolander predicts the service levels of RP Zookeeper with less than 0.002 error across diverse workloads, complex Zookeeper setups, and heterogeneous hardware. Using just 4 shared-nothing duplicates, RP Zookeeper can serve reads and writes within 15ms at a 4-nines service level (99.995%). Comparitively, using traditional, replication for throughput achieves a service level of only 99%—RP Zookeeper improves by 2 orders of magnitude. Using Zoolander to guide replication strategy, RP Zookeeper can reduce the execution time of a scientific computing workload by 373%. Or, it can meet an preset SLA using 50% fewer servers.

This paper is organized as follows: Section 2 overviews the emerging data-intensive workloads that we target, explaining the need for richer SLAs. Section 3 distinguishes our work from prior performance models and storage systems. Section 4 details the first principles of replication for predictability for service-level management and presents Zoolander. Section 5 summarizes the implementation of RP Zookeeper. We also present results on it's service-level scalability and on the accuracy of Zoolander. Section 6 demonstrates Zoolander in the online management of two data-intensive benchmarks. Section 7 concludes.

2 Background: Data-Intensive Workloads

Our research provides support for transforming data-intensive workloads into services on cloud-based storage. Today, these workloads are commonly batch processed, e.g., as MapReduce jobs, without stringent response time constraints. Moving toward a service model with predictably fast response times could benefit end users and workload managers alike. End users could flexibly perform a wider range of operations in real time [32]. Managers, especially for scientific computing workloads, can benefit from lower costs by dynamically provisioning cloud resources instead of investing in dedicated hardware [18].

In this section, we first describe the hardware and software architectures underlying cloud storage. Then we discuss the storage workloads that arise from data-intensive services and the impact of slow storage accesses on such services. Finally, we present the two data-intensive services used in this paper as representative benchmarks.

Hardware: Large datacenters comprising thousands of commodity servers enable new, in-memory system designs for networked storage. Instead of buying a few custom servers with enormous secondary storage capacity, these new designs exploit the main memory of many commodity servers. These servers share only the datacenter's power delivery and network resources—hence, these designs are often called *shared nothing*. Since the servers are independent, simple fail-over redundancy can mask hardware failures.

One storage access on shared-nothing systems can be extremely fast, needing only 1) a network roundtrip, 2) light computation, and 3) a memory access. However, if the data layout or consistency model requires communicatation between servers, the performance won't scale. One approach to reduce such inter-server communication is to carefully partition data. Traditional replication can also be used, provided the storage system can support relaxed consistency models. Partitioning and replication are easy to manage in key-value storage systems. A partition copies a subset of keys from a server to a new server and replication carbon copies an entire server's data. The challenge is understanding when to apply each approach.

Data-Intensive Workloads: Static-content and e-commerce workloads famously use shared-nothing storage to support interactive Internet services, using partitioning and replication to produce service level guarantees on 99% of storage accesses. In these workloads, only a few storage accesses must complete quickly to achieve good results—for example, a 99% SLA on storage accesses often translates to a 99% SLA on static webpage requests. However, data-intensive services present workloads that need many fast storage accesses. Even a few slow storage accesses can degrade performance for the whole workload. Below, we describe such workloads, explaining the outsized effect of a few slow storage accesses.

Inner-Join Olio: This workload represents a request type within a social networking service. It presents users with a new view of their data based on a non-indexed search [28, 32]. Specifically, it allows users of Olio, a social calendar benchmark [25], to find events posted by users with fuzzy (non-indexed) properties, e.g., find the hostess of tagged events posted by users born in Oakland, CA. In others words, it computes an inner join:

```
SELECT event.hostess
FROM event INNER JOIN user
ON event.taggedBy includes MY.NAME and
event.id in user.events and user.origin = Oakland;
```

Like other Olio requests, this workload must return quickly to satisfy end users. Unlike other requests, this workload requires many storage accesses. First, key-values corresponding to the tagged events must be retrieved, then values that match the search criteria spark another access to the user data. Even services with small user databases will generate hundreds of storage accesses on these requests. In a key-value storge system, the join predicate can be parallelized at a fine granularity and spread across many cloud nodes, but the result is incomplete until all parallel accesses finish. Thus, the slowest parallel accesses are a lower bound on the response time of the entire request.

Cloud Gridlab-D: This workload runs smart-grid simulations [6] on a shared-nothing cloud. Gridlab-D uses event-driven simulation to model a wide range of energy producing/consuming devices from power plants to air conditioners. Created in C++, each class implements an simulation agent that first waits for a triggering event, then updates its local state, and finally triggers other agents. Figure 1 depicts a workload that tracks the energy usage for a household, personal choices within it, and a water heater. Agents in brackets depend only on the storage system; they can execute in parallel (here, storage system also maintains global variables). These agents must complete before agents later in the pipeline can trigger, even if every agent is given its own compute resources. Slow storage accesses can affect many dependent agents, inflating response time linearly.

As shown by the above workloads, data-intensive services must complete many storage accesses within tight response time constraints—all of their accesses must complete quickly. We instrumented the Gridlab-D workload and injected a random delay into 2% of its storage accesses during a 5-minute simulation. Figure 2 plots the slowdown caused by the few slow requests against the injected delay per access. If Gridlab-D were like static-content workloads, the slowdown after a 5-minute test would be proportional to the injected delay per access, i.e., the expected increase in response time. Unfortunately, injecting a few slow requests made the simulation run 53–133% slower than a static-content workload would. The outsized effect of a few slow storage accesses on data-intensive workloads raises a new research challenge: Is it possible to exploit the abundance of commodity cloud servers to improve service levels by several orders of magnitude while maintaining low latency? In this paper, we will demonstrate that replication for predictability provides a scalable solution.

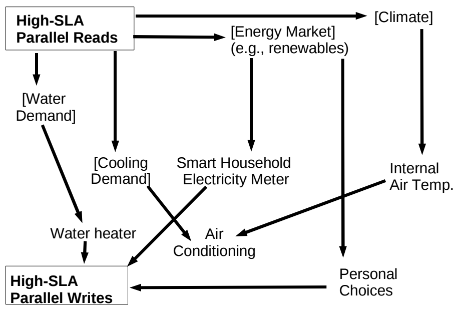


Figure 1: Data dependence in a Gridlab-D simulation for 1 time step. Unframed text blocks represent simulation agents. Lines indicate data dependence. Dependences across time steps are not shown.

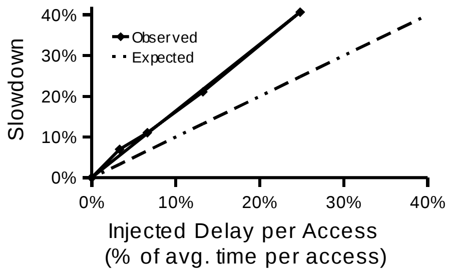


Figure 2: An experiment with the Gridlab-D workload. Random delay was injected into 2% of storage accesses. The X-axis shows the amount of delay inserted per access. The dotted line plots a slope of 1, i.e., the expected slowdown for workloads that make few storage accesses. The solid line plots the observed slowdown in Gridlab-D.

3 Related Work

Parallel, data-intensive services will produce diverse and demanding storage workloads that need high service levels and low latency. Our goal to manage networked storage for these workloads distinguishes our efforts from prior work—motivating us to rigorously study replication for predictability and consider it in online management. Prior work that used replication for predictability targeted specific workloads with less stringent service-level demands, allowing them to apply the approach sparingly based on ad-hoc rules of thumb. Prior performance models and anomaly classification tools have focused on the service-level benefits of partitioning and traditional replication.

The SCADS manager is the most related system [30]. SCADS achieved service levels of 99.5% for the EBates.com workload by 1) dynamically partitioning a key-value store and 2) by using 1 duplicate copy. We distinguish a duplicate, used in replication for predictability, from a replica, used in traditional replication for throughput. Specifically, SCADS observed that even small data partitions under light load incurred performance anomalies that violated SLAs. However, since their target workloads did not demand several-nines SLA, SCADS used only 1 duplicate to avoid anomalies. From an implementation persepective, SCADS also benefitted from a relaxed-consistency data model that effectively made their workload read only. The Gridlab-D workload described in Figure 1 needs consistent writes in over 40% of its storage accesses. Further, it demands a stronger SLA, as a 99.5% SLA would translate to to only 96.5% SLA for an individual simulation time step. Our contribution is Zoolander, an analytic model that answers the question, "How many duplicates are needed to get a 99.9% SLA for Gridlab-D?"

Mantri [2] applied replication for predictability to map-reduce applications. Working at a higher level, they traded the costs of a map-reduce task being run twice (due to failure) against the lost throughput from replication for predictability. Mantri's decisions varied depending on 1) the amount of input data that would be recomputed in pipelined map-reduce jobs and 2) the cost of replicating data. An important distinction is that Mantri provided only best-effort outlier removal, justifying their usage of only 1 duplicated job. In contrast, our goal is to provide predictable and strong SLAs to data-intensive services. These services may not migrate to cloud storage otherwise, and many consider clouds cost effective compared to supercomputers anyway [18]. Zoolander permits reasoning about the precise performance benefits of many duplicates (we have tested up to 16).

Our service-level modelling approach builds on the prior success of performance models from queuing theory [12, 26, 31] to machine-learned models [3, 8] to control theory [7, 20, 24] (to cite only a few). Such research has been broadly impactful in the development of the foundations for cloud computing, motivating successful start ups like RightScale and demanding a place in core system designs (Amazon Queuing service). While rigorous, traditional queuing models focus on the service time to arrival rate at a single node. To date, we are unaware of a work that has extended this theory to replication for predictability in Internet services. Machine learning can predict a broader range of performance mechanism, albeit with fewer core insights. Indeed, we adopt a hybrid approach in this paper. Using machine-learned service times [30] and a first principles model of replication for predictability.

Finally, our work focuses on nondeterministic performance anomalies. Other related work has studied deterministic performance anomalies caused by performance bugs. EntomoModel [27] and IronModel [29] both use machine-learned models of the system to select configurations that avoid anomalies during runtime. Reference-Driven Anomaly Detection [23] uses the manifestation patterns of performance bugs to find hints on their underlying root causes.

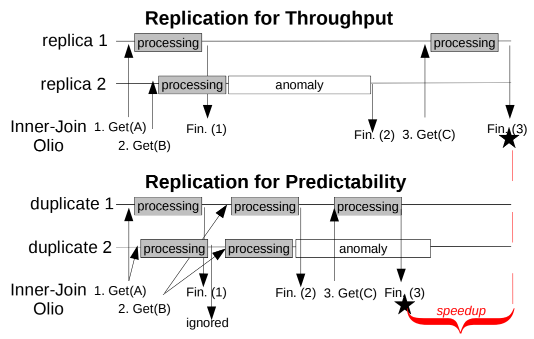
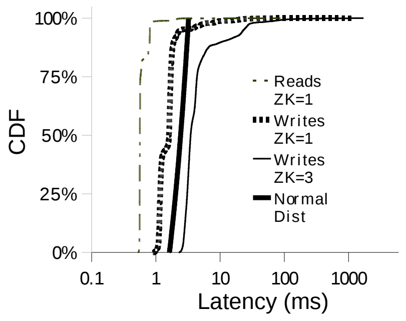


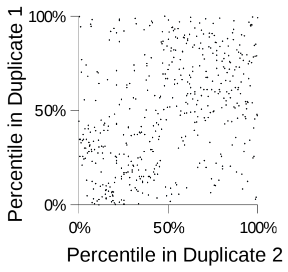
Figure 3: Replication for predictability versus replication for throughput. In this example, only predictability speeds up the inner-join Olio workload. Horizontal lines reflect the local time of a cloud server. The numbered commands reflect Olio access patterns. The terms replica and duplicate refer to servers that store the same data, *but*, unlike replicas, duplicates process the exact same workload—A feature which masks nondeterministic anomalies. Star means the workload is complete. Note, get #3 depends on gets #1 and #2.

4 Zoolander

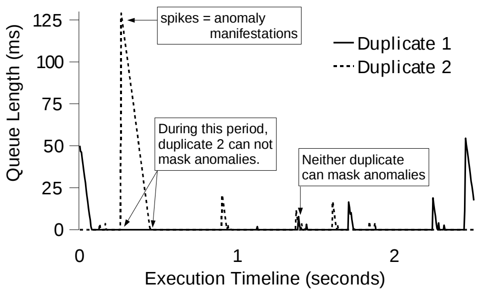
Figure 3 compares replication for predictability against traditional, replication for throughput. In traditional replication, each request is processed on just one cloud server; adding replicas increases the number of requests that can be processed. This parsimonius approach suffers when the processing server incurs a performance anomaly, i.e., the perceived processing time is much longer than expected. In networked storage, many factors can induce anomalies, from the OS scheduler to



(a) CDF of access latencies in Zookeeper. Each experiment looks at 1 Zookeeper group containing 1 node (ZK=1) or 3 nodes (ZK=3). Writes and reads are separated.



(b) Nondeterministic anomaly manifestations. An anomaly in either duplicate does significantly increase the probability of anomaly in the other. Correlation coefficent equals 0.25.



(c) Divergent waiting times over time.

Figure 4: Empirical evidence in support replication for predictability.

garbage collectors to background data copying [30] to misconfigurations [27].

Performance anomalies have less impact under replication for predictability; N duplicates can mask $N-1$ anomalies. However, duplicates sacrifice throughput, achieving only $1/N$ throughput of tradtional replication. When anomalies— not workload access patterns— are the root cause of SLA violations, replication for predictability can help to reduce response times. Figure 3 shows that the parallel storage accesses issued by inner-join Olio can benefit from this approach.

This section describes Zoolander, an analytic model that predicts service levels given a replication strategy. Zoolander is based on the following principles of replication for predictability:

1. *Duplicates operate independently.* To mask performance anomalies, slow accesses on 1 duplicate can not spread to others. Duplicates should have their own (virtual) resources, and they should process storage accesses without depending on other duplicates.

2. *Duplicates mask nondeterministic performance anomalies.* Low response times and high throughput are common goals in production storage systems. They have been designed to meet this goals, so few deterministic performance bugs make it onto production servers [27]. However, nondeterministic anomalies are hard to fix and are known to reach production systems. Here by nondeterministic, we mean that these anomalies manifest independently of workload patterns.

In the remainder of this section, we confirm these first principles on a cloud platform. Then we present the modelling approach for Zoolander, and show that Zoolander alone is useful to understand basic replication tradeoffs. But first, we discuss data consistency under replication for predictability.

4.1 Consistency Issues

If duplicates operate independently (principle #1), a failed duplicate can not make storage unavilable. According to the CAP Conjecture [4, 13], networked storage with high availability must sacrifice either data consistency or partition tolerance. The former ensures that a storage access returns only data that is up to date, whereas the latter allows the system to grow by adding networked servers. In traditional replication, partition tolerance scales throughput with the number of servers, so data consistency is often sacrificed [1,4,10,30]. However, as shown in Figure 3, replication for predictability is not intended to scale throughput; predictable responses (both in service level and data value) are the key. Here, we describe a simple approach to sacrifice partition tolerance by funnelling storage accesses through one machine. We connect all duplicates to a message repeater, i.e., a device that broadcasts network messages to all connected devices in FIFO order. Workloads access duplicates only through the repeater. If a duplicate misses any broadcast message or fails to complete a storage access, it stops running. The repeater should achieve higher throughput than any single duplicate and incurr few anomalies. These traits can be achieved with hardware or software repeater. Section 5 describes our implementation of such a software repeater between Zookeeper duplicates.

4.2 First Principles

We studied the service times (i.e., processing time for a storage access) in our own local, private cloud. By running on a private cloud, we gained repeatability and control in our experiments. We use a 16-processor, 32-core Dell cluster, where each core operates at 2.66 GHz with a 3MB L2 cache. Only one virtual machine can run on each processor, eliminating L2 cache sharing (this is relaxed in Section 5). Idle cores use p-states to lower their power draw. Our virtualization software is User-Mode Linux (UML) [11], a port of the Linux operating system that runs in user space of any X86 Linux system. Thus, RedHat Linux (kernel 2.6.18) serves as our virtual machine monitor. Custom PERL scripts designed in the mold of Usher [19] allow us to 1) run predefined virtual machines on server hardware, 2) stop virtual machines, 3) create private networks, and 4) expose public IP addresses. Our cloud infrastructure is compatable with any public cloud that hosts X86 Linux instances, including Amazon EC2.

For this work, we chose Zookeeper as a storage substrate [15]. Zookeeper retreives values (data) given a key (some string). Such key-value storage provides fast access and loose structure that can help data-intensive services grow. In this paper, we use the following terms to describe a Zookeeper setup: 1) a *Zookeeper group* is a group of cloud servers that host the same subset of keys, and 2) a *Zookeeper node* is one cloud server in a Zookeeper group. Zookeeper allows for inconsistent reads but enforces consistency in writes. For reads, Zookeeper sends the result from only 1 Zookeeper node, improving throughput as described above. On the other hand, writes must contact a majority of Zookeeper nodes in a group. If a network partition blocks communication within the group, the write service becomes unavailable.

We created an initialization script that automatically sets up Zookeeper in our cloud. The script takes the IP addresses of all Zookeeper nodes within each Zookeeper group. Experiments in this section focus on all-read or all-write storage accesses. Each key stores a value that is 2KB. In total, there are 1,000 keys so the total dataset fits within L2. Accesses were issued one after another with no concurrency or according to a preset rate. For simplicity, duplication is performed at the group level throughout this paper.

Figure 4(a) plots the cumulative distribution function (CDF) for Zookeeper reads and writes in our cloud. All of the experiments followed a low-variance, normal distribution for fast service times, i.e., latency below the 70^{th} percentile. But variance across the entire distribution was much higher, producing coefficients of variation ($\frac{\sigma}{|\mu|}$) ranging from 1.5–8. To visually highlight the heaviness of the tails, Figure 4(a) also plots a normal distribution with standard deviation and mean that were 25% larger than the fastest 90% of write times in ZK=1. The tails for both reads and writes under ZK=1 overtake the normal distribution, even though the normal distribution has a larger mean. We also found that the distribution tails increased as requests accessed more system resources. Writes in a single-node Zookeeper lead to local disk accesses that didn't happen under reads. Writes in 3-node Zookeeper groups send network messages for consistency. This behavior suggests that local resource management, e.g., handling I/O interrupts, may induce some anomalies. Finally, Figure 4(a) also provides evidence to motivate this study. Even when we set a lax latency bound of 2 times the mean, Zookeeper only provided a service level of 98.8%,95.7%,91.5% for reads, writes to 1 node, and writes to 3 nodes respectively. Alternatively, to provide a service level of 99.99% (needed for parallel, data-intensive services), the latency bound would have risen to 16X, 26X, and 99X relative to the respective means.

We also measured read access times in Memcached, a simple, inconsistent key-value store often used in practice [22]. We saw a coefficient of variation of 1.9, and, under a lax latency bound, only 98.3% service level was achieved. This result suggests that anomalies due to local resource management afflict other key-value stores as well, not just Zookeeper. We also looked at the service time distribution for browsing requests in RUBiS, a benchmark for online auction services. The tail here was much less heavy with a coefficient of variation of only 1.2. As a more complicated service, mean response times in RUBiS were much larger than Zookeeper, making issues like interrupt scheduling less impactful relative to the total processing time. This result shows that while there is potential for replication for predictability in computation-intensive services [2], storage infrastructure can gain a lot from mitigating heavy-tailed access distributions. Further, complicated services are more likely to be afflicted by workload-dependent, deterministic performance bugs [27] compared to more simple key-value stores.

Figure 4(b) highlights principle #2. Across two duplicates that run the same workload, we show the percentile of each storage access. If anomalies were workload dependent, either the bottom right or upper left quartiles of this plot would have been empty. Instead, every quartile was touched evenly. Figure 4(c) plots the queue lengths of these two duplicates over time. In this experiment, storage accesses arrived at a set rate of 500 requests per second, below our peak throughput of 2,000 requests per second but enough to cause significant queuing. Performance anomalies caused the plot to spike. After an anomaly, subsequent requests processed on the same cloud node must wait to be processed. At several points, one of the duplicates is unable to mask an anomaly because its queue is too long. Queuing delay can pass anomalies across storage accesses and even servers. We must consider queuing in Zoolander.

4.3 Service-Level Model

In this subsection, we will reference the symbols defined in Table 1. Our model seeks to characterize the service level provided by N independent duplicates running the exact same workload under nondeterministic performance anomalies. In our model, an anomaly is defined as an SLA violation. The storage system manager provides target latency (τ ms) for the SLA.

	Model Inputs
$\hat{s}$	Expected service level
N	Number of duplicates used to mask anomalies
τ	Target latency bound
$\Phi_n(k)$	Percentage of service times in Zookeeper duplicate n with latency below k
λ	Mean interarrival rate for storage accesses
μ_{net}	Mean of network latency between Zookeeper duplicates and storage clients
μ_n	Mean service time for Zookeeper duplicate n (derived)

Table 1: Zoolander inputs.

Using principles #1 and 2, Zoolander first models the probability that the fastest duplicate will respond within SLA. Recall, all duplicates run the exact same workload and can provide consistent/acceptable data, allowing the fastest duplicate to provide results. This probability is computed with the following series:

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

To provide intuition into this result, consider $\Phi_0(\tau)$ is the probability that duplicate 0 meets the τ ms latency bound in SLA. If $N = 2$, $\Phi_1(\tau) * (1 - \Phi_0(\tau))$ is the probability that duplicate 1 masks an anomaly for duplicate 0. Note, the order in which we consider Zookeeper duplicates does not affect the service level estimate. When all duplicates have the same service time distribution, we can reduce the above equation to a geometric series. Then the N^{th} moment of the series is easily computed below: (Note, as N approaches infinity, the expected service level converges to 1.)

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

Queuing and Network Delay: SLAs reflect a client's perceived latency which may include processing time, queuing delay, and network latency. Since duplicates execute the same workload, they share access arrival patterns and their respective queuing delays are correlated. Similarly, networking problems can affect all duplicates. As shown in Figure 4(c), queuing delay can significantly increase execution times.

Here, we lean on prior work on queuing theory to answer two questions. First, does the expected queuing level completely inhibit replication for predictability? And second, how many duplicates are needed to overcome the effects of queuing? The key idea is to deduct the queuing delay from τ in the base model. Intuitively, requiring all duplicates to reduce their expected service time in proportion to the expected queuing delay.

$$\tau_n = \tau - (\frac{1 + C_v^2}{2} * \frac{\rho}{1 - \rho} * \mu_n) - \mu_{net}$$

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

We used an M/G/1 queuing model to derive the expected queuing delay, reflecting the heavy-tail service times observed in Figure 4(a). To breifly explain the first equation above, an M/G/1 models the expected queuing delay as a function of system utilization (ρ), distribution variance (C_v^2), and mean service time. Utilization is the mean arrival rate divided by the mean service time. Note, that the new τ may be different for each Zookeeper group (parameterizing it by n). An M/G/1 assumes that inter-arrivals are exponentially distributed. This may not be the case in all data-intensive services. A G/G/1 with some constraints on inter-arrival may be more accurate. Alternatively, an M/M/1 would have simplified our model, eliminating the need for the squared coefficient of variance (C_v^2). Prior work has shown that multi-class M/M/1 can sometimes capture the first-order effects of M/G/1. We leave empirical analysis of the right queuing network to future work. We also deduct the mean time lost to network latency. Here, network latency is the average delay to send a TCP message between any two cloud servers.

4.4 Model-Driven Analysis

Figure 5 uses Zoolander to compare the expected service levels achieved via traditional replication versus replication for predictability. The expected queuing delay affects which approach is best. We used the CDF from "writes, ZK=1" in Figure 4(a) to obtain a mean service time and the squared coefficient of variation (no queuing at all). We set the target latency bound to 5 ms. We varied the utilization (i.e., ρ) by changing the arrival rate of storage accesses. Traditional replication divides the arrival rate across all replicas evenly (in this case, allowing for inconsistent writes [10, 30]). We plugged all of these parameters into Zoolander to get expected service levels.

Figure 5 shows that replication for predictability is not appropriate under even moderate queuing delay. The service time skew due to heavy-tail anomalies magnifies expected queuing delay by a factor of 3. Such large queuing

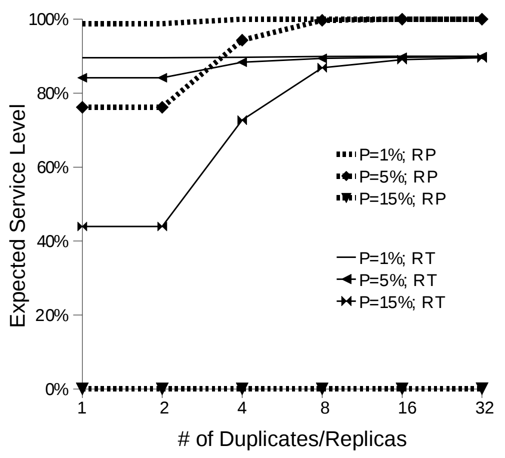


Figure 5: Trading throughput for predictability. Zoolander predicts the service level achieved under different replication strategies and workloads.

delay reduces the target latency for Zoolander duplicates to the 2nd percentile, producing the flat line at the bottom of the plot. Fortunately, traditional replication mitigates queuing delay. Recall, replicas balanced the arrival rate evenly, greatly reducing the per-node ρ. We observe that 15% utilization is mitigated using 8 replicas (ρ=2%).

Unfortunately, traditional replication can not improve service levels under light workloads with little queuing. In this example, 5 ms corresponds with the 90^{th} percentile. Replication for predictability is not bound by the service time distribution. Under light queuing, it is expected to achieve a service level of serveral nines. Of course, these approachs are not exclusive. When queuing is significant, storage managers can choose traditional replication (or partitioning). When anomalies are prevailent, replication for predictability is appropriate. Zoolander helps managers find the sweet spots for their system.

5 RP Zookeeper

Figure 6 depicts RP Zookeeper, our add-on to the Zookeeper coordination service. RP Zookeeper enables replication for predictability for both read and write key-value accesses with consistency guarantees similar to Zookeeper. Writes are processed in FIFO order, but reads may be inconsistent. This section describes implementation details of each component that were key in the design of RP Zookeeper. First, we make a case for Zookeeper as the underlying storage substrate. Then, we describe our message repeater that ensures that state-changing accesses arrive at each duplicate in the same FIFO order. The RP Zookeeper proxy ensures write consistency across duplicates and responds directly to clients, reducing network roundtrips. Subsection 5.1 uses RP Zookeeper to empirically validate Zoolander across workloads, platforms, and SLA bounds.

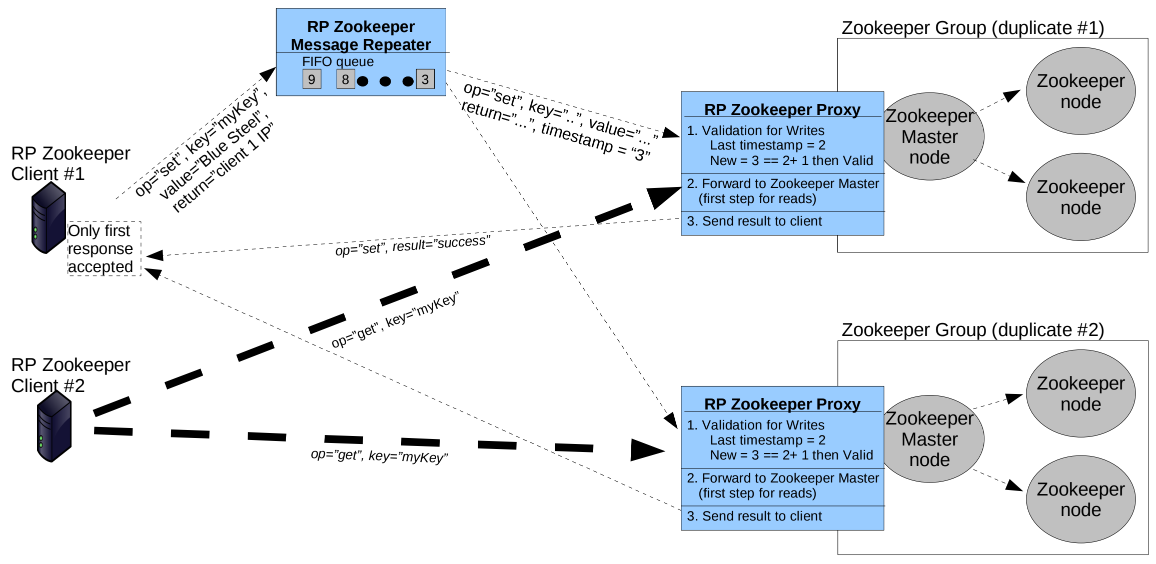
RP Zookeeper was built as an extension of Zookeeper, even borrowing its name. However, we believe that other key-value systems can also benefit from replication for predictability. For instance, our results in Section 4 suggest that Memcached also suffers from nondeterministic, heavy-tailed anomalies. Prior work [30] has show that replication for predictability can be built on top of other key-value systems (SCADS), albeit prior implementations have been more limited than ours.

Zookeeper [15]: Replication for predictability benefits from two implementation decisions in Zookeeper. First, Zookeeper avoids locks in favor of wait-free synchronization primitives. A storage access in Zookeeper can not be delayed because another client holds a lock. This reduces the resource sharing (i.e., no locks) in Zookeeper, which reduces performance anomalies. The tradeoff is that storage accesses in Zookeeper can fail, unlike pure lock-based systems.

Wait-free primitives also simplify Zoolander, our service-level model. In a pure lock-based system, anomalies due to lock-hogging clients could force other duplicates to interact to maintain consistency. This would violate Zoolander's first principles. It would possibly force us to consider the effect of lock-hogging. Lock-hogging can be a serious problem. Most production systems, like Chubby [5] and Farsite [1], now use leases to thwart lock hoggers. Leases bound the amount of time that 1 client can delay other clients. Wait-free synchronization eliminates the problem.

Zookeeper also enforces first-in-first-out (FIFO) order on its write accesses. This allows clients to issue multiple accesses at once. The Zookeeper leader node orders their arrival into the system, only committing writes in that order. To commit a write, a majority of nodes must be contacted using the Zab Atomic Broadcast protocol [16, 17, 21]. In RP Zookeeper, we use a software-based repeater to ensure that all duplicates see writes in the same order.

Message repeater: Our message repeater sits between the client and Zookeeper duplicates. All state changing accesses, e.g., write, create, and delete, go through the message repeater. We implemented the message repeater in C. It runs on its own server to maximize its throughput. It maintains one FIFO queue for all accesses sent through it. It forwards accesses to all duplicates in the same order, sending one access at a time. Specifically, accesses from multiple clients are *not* interleaved. The above operation guarantees both linearizable writes and



RP Zookeeper Message Repeater
FIFO queue
9
8
3
RP Zookeeper Client #1
Only first response accepted
op="set", key="myKey", value="Blue Steel", return="client 1 IP"
op="set", key="...", value="...", return="...", timestamp = "3"
RP Zookeeper Proxy
1. Validation for Writes
Last timestamp = 2
New = 3 == 2+ 1 then Valid
2. Forward to Zookeeper Master (first step for reads)
3. Send result to client
Zookeeper Group (duplicate #1)
Zookeeper Master node
Zookeeper node
Zookeeper node
op="set", result="success"
op="get", key="myKey"
RP Zookeeper Client #2
op="get", key="myKey"
Zookeeper Group (duplicate #2)
RP Zookeeper Proxy
1. Validation for Writes
Last timestamp = 2
New = 3 == 2+ 1 then Valid
2. Forward to Zookeeper Master (first step for reads)
3. Send result to client
Zookeeper Master node
Zookeeper node
Zookeeper node

Figure 6: The components of RP Zookeeper. Directed lines indicate data paths across the network for reads (bold) to writes (dotted). The depicted Zookeeper groups comprise 3 nodes each. Replication for predictability occurs at the group level. For visual clarity, return values from the Get operation are omitted.

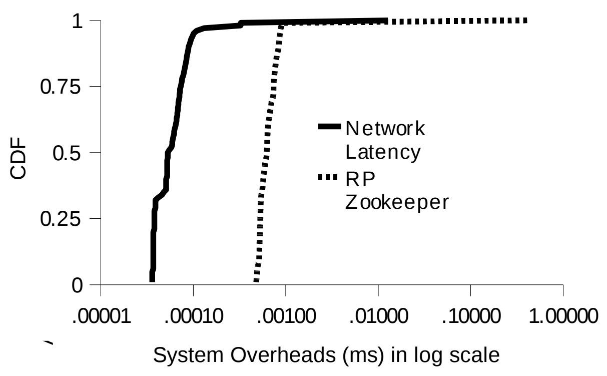


Figure 7: The cdf of the network delay and the whole system overhead of RP Zookeeper

FIFO client order similar to zookeeper. When the repeater forwards a message, it attaches a logical timestamp, reflecting the “ticket number” of the access. The timestamp is increased by 1 after each message.

Zookeeper Proxy: Zookeeper proxy is a C program running on the leader node of each Zookeeper group. The main thread of proxy listens on the connection from the message repeater for change-state accesses like write, and from clients for read accesses, then selects the first upcoming packet to process. Packets are defined by a struct that contains fields for request type (read, write, create or delete), client address to send the result to, a key and value, and also a timestamp that denotes the order of the change-state request. Besides, the proxy also contain a second thread which dequeues packets from the queue, unpacks them, then validate their timestamps. The Zookeeper proxy validate one timestamp by comparing it with the most current timestamp. If it is one digit larger then the most current one, it is valid and the request will be processed. If not, the proxy assumes that a failure has occurred and kills the duplicate. Packets with valid timestamps will then be sent to the local Zookeeper leader node to be processed; then their result will be taken and forwards to the client by a third thread. The proxy also examines Zookeeper outputs before forwarding to the client. If somehow, Zookeeper fails locally (e.g., many disk crashes), the proxy kills the entire duplicate. Such local failures include extremely slow access times that exceed a lax timeout.

Finally, we note that RP Zookeeper provides one additional operation type: write-all-duplicates. The operation parallels Zookeeper’s sync command by ensuring that a write to all duplicates has completed successfully. A write-all-duplicates message occurs on the client side, but guarantees that a subsequent read from a specific node will return an up-to-date value.

Client: Clients use a special library for RP Zookeeper. It packs all of the Zookeeper request into the packet format as described above. It sends state-change accesses to the message repeater and non-state-change accesses as read directly to the Zookeeper proxy. The client also listens for the response from the Zookeeper proxy by opening a port with a second thread. Finally, the client can detect subtle inconsistencies if duplicate responses ever differ, killing offending duplicate.

Overhead: We also tested the system overhead of the RP Zookeeper, which consists of two parts as the network delay and the Zookeeper process time. As shown in Figure 7, the mean system overhead is very small ($\leq$0.001ms) in RP Zookeeper and reliable and it maintain less then 0.5ms even in the worst situation when network outlier occurs.

5.1 Model Validation & System Results

We deployed RP Zookeeper on the cloud platform described in Section 4. Zookeeper group size was set to 1 node. Initially, there were no duplicates. We then issued 100,000 writes one after another (no concurrency), getting a distribution of service times. We used this single-node distribution as input to Zoolander. Also, we used the 90^{th} percentile of this distribution as the default latency bound (τ=5ms) in our experiments (also a Zoolander input). We then added duplicates to RP Zookeeper one at a time, issuing the same write workload each time we added a duplicate. Figure 8(a) shows RP Zookeeper's performance, i.e., achieved service level, as duplicates increase. Specifically, the achieved service level grew as duplicates were added. For example, under 8 nodes, RP Zookeeper could support the following SLA: 99.9% of write accesses will complete within 5ms. The graph also shows that Zoolander had absolute error (i.e., actual service level minus predicted) below 0.002 in all cases.

In our next test, we set the number of duplicates to 8 and collected a service time distribution. We then changed the latency bound (τ) to different percentiles in the single-node distribution, from the 75^{th} to 99.5^{th}. High percentiles led to several-nine service levels in RP Zookeeper, forcing Zoolander to be accurate with high precision. Low percentiles required Zoolander to accurately model more accesses. Figure 8(b) shows Zoolander's accuracy as the latency bound increased. Absolute error was within 0.001 for high and low percentiles. In Figure 4(a), we observed that write access times had a heavy-tail distribution that started around the 96^{th} percentile. Figure 8(b) shows a steeper slope (strong gains) for latency bounds after the 96^{th} percentile. For instance, setting the latency bound to the 99^{th} percentile of single-node distribution (τ=15ms), RP Zookeeper achieved 99.991% service level using only 4 duplicates. In other words, 4 duplicates improved the service level by two orders of magnitude.

Diverse Workloads Figures 8(c) and 8(d) shows the number of nines achieved under read accesses and under larger Zookeeper group size. On our cloud platform, reads completed in microseconds [15]. Sometimes our software repeater had not finished broadcasting accesses before a duplicate finished the job. Figure 8(c) shows the results with just 2 duplicates. As we varied the latency bound, Zoolander accurately estimated service level. We focus on the number of nines because it is a common metric in practice for SLAs. In one case, Zoolander misestimates the number of nines, but the error in that case was only 0.0009.

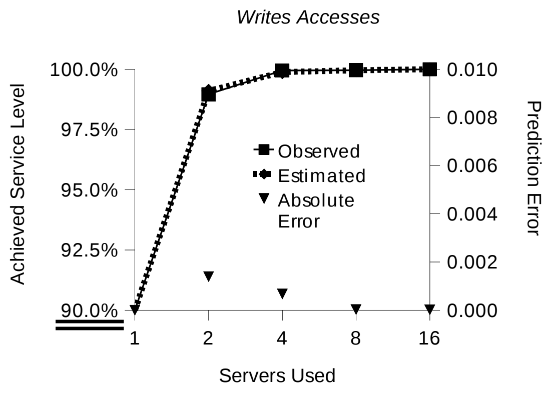
Figure 8(d) shows results where we set the group size to 3 under a write workload. Recall, Zookeeper uses an atomic broadcast to issue group writes. Communication between group nodes increases anomalies. Despite this increase, Zoolander accurately predicted the number of nines across all tested latency bounds.

Heterogeneous Platforms We were also able to change our cloud platform, allowing virtual machine instances to share the L2 cache. We created a new single-node distribution of service times for this architecture. We found that Zoolander still accurately predicted service levels with less than 0.0002 error. We also changed the networking substrate, making virtual machines communicate through a user-level SLIRP device. Zoolander still predicted service levels with less than 0.001 error.

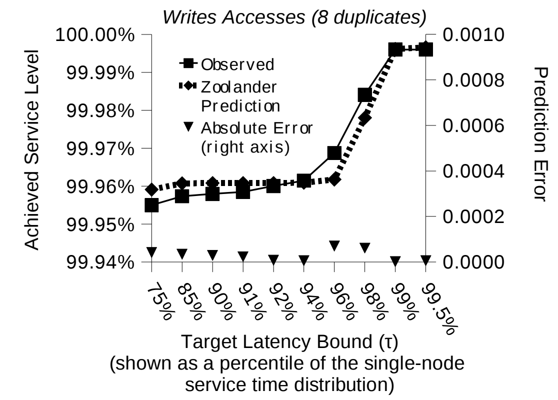
6 Zoolander in System Management

Section 2 presented two targeted workloads: Inner-Join Olio and Gridlab-D. We implemented those workloads and captured a storage access traces from each. Below, we describe the access patterns of each. We then use replication for predictability to manage SLAs and costs for these workloads.

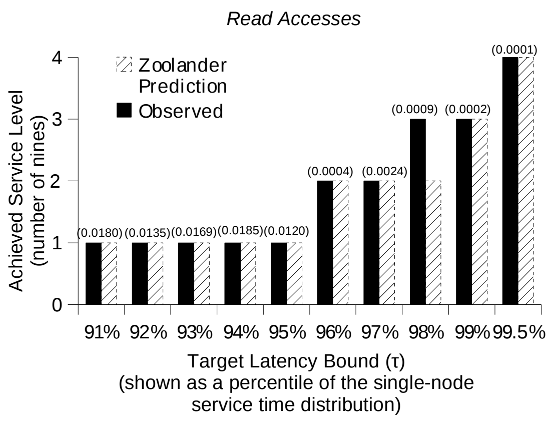
For Inner-Join Olio, we created an Olio request type that reads a user's tagged events, then for each event that matches a user-supplied keyword, we look up the user that posted the event. This workload typically produced about 30 parallel read accesses to get tagged events and another 20 parallel accesses to get user data from matching events. For RP Zookeeper, we essentially used two tables. The first was indexed by event IDs and the second by user IDs. A key was an ID and the value held all pertinent information. This workload issues only read accesses. Since these requests come for independent users, we assume that storage accesses appear exponentially distributed at the Zookeeper side. To be sure, we use the term request to refer to 50 (30 then 20) parallel storage accesses.



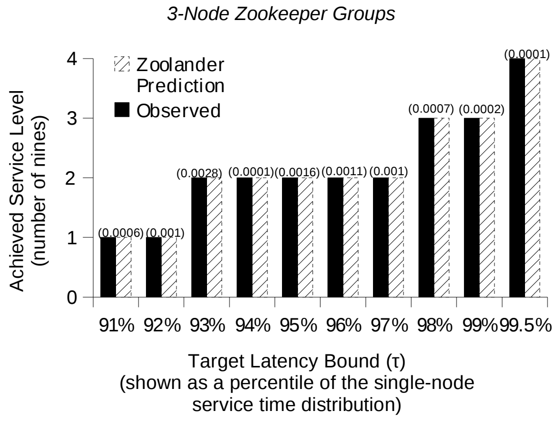
(a) Achieved service levels against Zoolander predictions as duplicates increase.



(b) Service levels as the target latency bound changes. 8 duplicates used.



(c) Service levels achieved on read-only accesses. 2 duplicates used.



(d) Service levels achieved under 3-node Zookeeper deployment. 2 duplicates used.

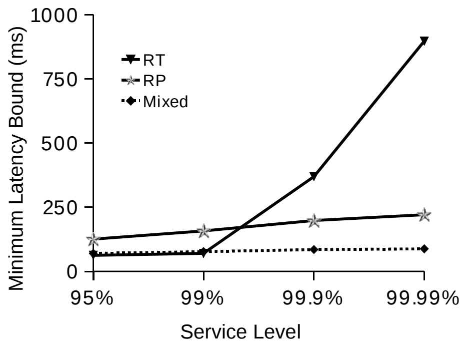
Figure 8: Validation Experiments

For Cloud-based Gridlab-D, we used the instrumentation described in Section 2. We replayed the captured trace, issuing 4 reads and 3 writes in parallel for each simulation event. The requests arrived in batches under this workload, reflecting surges in storage accesses at the beginning and end of a simulation event. Note, this defied Zoolander's assumption that request inter-arrivals are exponentially distributed. In this sense, our M/G/1 model could only approximate that actual G/G/1 queuing that batched arrivals produced.

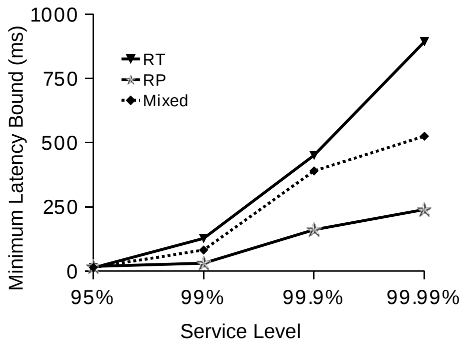
Our first goal was to answer the following question, *"What replication strategy produces the highest SLA under these workloads?"* Specifically, using 4 servers, RP Zookeeper can be configured in 3 ways. First, all four servers can use traditional replication. Second, all four servers can use replication for predictability. Third, two servers can use traditional replication while having 1 duplicate a piece (i.e., 2 replicas and 2 duplicates).

Figures 9(a) and 9(b) show that by using replication for predictability, we can support much stronger SLAs than by using only replciation for throughput. For read-only, exponentially distributed Olio workload (Figure 9(a)), the mixed solution provides the best tradeoff. Peak throughput doubled (via replication for throughput) at the cost of a small increase in the SLA latency bound. However, the batched and write-heavy workload Gridlab-D required more duplicates for high service level. Replication for predictability needed to be scaled to all for nodes to meet a stringent (99.9%) SLA.

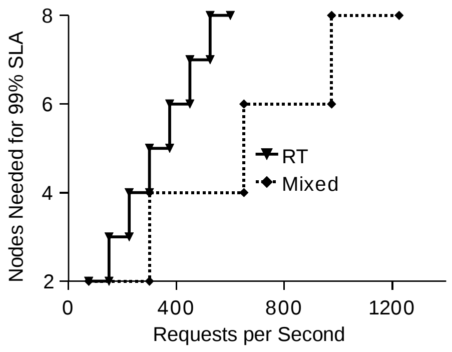
Our second goal was to answer the question, *"what if we used dynamic cloud provisioning, how many servers are needed to meet a set SLA?"* We set the SLA to 99% of all accesses must complete with 10ms. Figure 9(c)) plots the number of servers need to meet the above SLA as the request arrival rate increased. We compared using only replication for throughput against using a mixed approach. For simplicity, all replicas had the same number



(a) Service level agreements for inner-join Olio. This is a read-only workload. Inter-arrival times for storage accesses were exponentially distributed. System utilization was 8%. Group size is 1 throughout this section.



(b) Service level agreements for Gridlab-D workload. This workload includes writes. Inter-arrivals are batched with each simulation event. System utilization was 8%.



(c) Nodes used as request rate increases for Gridlab-D. SLA was 99% of all accesses must complete within 10ms.

Figure 9: Management Experiments

of duplicates, forcing the mixed approach to provision by factors of 2. We used Zoolander to decide the proper mix of duplicates and replicas.

The approach that used only replication for throughput provisioned one server for every 75 requests. The mixed approach provisioned one server for approximately every 150 requests. Even though replication for predictability does not improve throughput, it can improve *goodput* substantially. Further, the best mixture of replicas and duplicates changed with request arrival and latency bound. For instance, 6 servers could contain two mixtures 1) 2 replicas with each duplicated 3 times or 2) 3 replicas duplicated twice. When the latency bound was 5 ms, the former provided better performance, but when the latency bound was 10 ms, the latter was best.

7 Conclusion

In this paper, we've examined replication for predictability as a mechanism to provide sufficiently high service levels for data-intensive workloads. Unlike e-commerce and static content workloads, these workloads, found in social networking and scientific computing, issue multiple storage accesses for each user request. We collected traces from two benchmarks, Olio and Gridlab-D, to confirm such access patterns. These workloads suffer from non-deterministic performance anomalies, especially when such anomalies cause heavy-tailed latency distributions. Replication for predictability avoids such anomalies online by executing the same storage accesses across many machines simultaneously. The fastest node provides the result, improving overall service levels. A key contribution in this paper was Zoolander, a model that characterizes the service level under replication for predictability using a geometric series. Zoolander considers the service times of each duplicate as well as the impact of queuing delay and network latency. We validated Zoolander on RP Zookeeper, an add-on to the Zookeeper Coordination Service that supports replication for predictability. Our results have two significant impacts. First, they show that replication for predictability can scale service levels as the number of nodes is increased, provided queuing delay and network latency are small. Second, they shows that Zoolander can help system managers decide between traditional replication and replication for predictability. This is important because the best management policy differs across workloads.

References

[1] A. 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 loosely-coupled 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 on*, 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, partition-tolerant 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 internet-scale systems. In *USENIX*, 2010.

[16] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Conference on DSN*, 2011.

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

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

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.