ADAPTIVE CONTROL FOR AVAILABILITY AND CONSISTENCY IN DISTRIBUTED KEY-VALUE STORES

BY

CANH SON NGUYEN BA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Associate Professor Indranil Gupta

ABSTRACT

The CAP theorem says that distributed key-value stores can only provide bounded consistency (C) and availability (A) under the presence of partition (P). Recent work has proposed the ability for applications of such stores to specify either an availability SLA or a consistency SLA. In this paper, we propose an adaptive algorithm that automatically controls the underlying storage system in real-time to meet such an SLA while optimizing the other C/A metric. We also present an implementation of the algorithm based on the popular key-value store Riak. Our experiments with the modified system, under realistic workloads, show that the control technique is able to change the system's configurations to quickly and stably satisfy the SLAs.

To my parents, brothers and sisters, for their love and support.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser Prof. Indranil Gupta for the advice and support he has given me along the way. His knowledge, insight, and attention to detail have been instrumental in helping me finish this work.

I would like to extend my utmost gratitude to Muntasir R. Rahman, who was my mentor over the past year. With his invaluable experience, Muntasir has guided me through many challenges of conducting research as a first-year graduate student.

I am also deeply appreciative to Jump Trading for their generous scholarship that has gone a long way in supporting my research.

Finally, I am eternally grateful to my parents, brothers and sisters, who have shown me unconditional love and support to pull through the tough times.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The CAP theorem [1–4] states that it is impossible for a distributed system to provide consistency, availability and partition-tolerance at the same time. Since partition-tolerance is critical in a distributed system and therefore must be satisfied, system designers have to decide between consistency and availability. Distributed key-value stores such as Riak [5], Cassandra [6, 7], Dynamo [8, 9], and Voldemort [10] favor the latter, offering only eventual consistency in exchange for extremely high availability. Nevertheless, these systems also make an effort of giving applications some control over the trade-off between the two guarantees via built-in settings. For example, applications of such stores can modify *replication factor* or *consistency level* to tune the availability and consistency. Replication factor indicates the number of replicas containing an entry. A higher replication factor would make the system more available in case of failures while having lower consistency. Consistency level governs the number of responses a coordinator needs to wait for before it returns to the client. A write with the consistency level of TWO would require the coordinator to wait for at least two replicas to acknowledge the write, giving more consistency but also making the client wait longer than a write whose consistency level is ONE.

However, in these systems, it is the application's responsibility to pick the right configurations (e.g., replication factor, consistency level, etc.) so that the system achieves the desired consistency or availability. A change in network conditions or system load would require the application to change these configurations in order to maintain the same performance. For example, if an application requires all its reads to be returned within some given time, a rise in network latency might entail a reduction in consistency level in order for the system to still satisfy this availability level. To the best of our knowledge, current versions of these distributed key-value stores do not have the ability to dynamically choose the right configurations and adapt to system load or network changes. As a result, applications running on these stores require human attention to cope with unexpected changes in network conditions or system status. In addition, this limitation makes these systems less attractive to inexperienced users as they have to manually modify low level configurations to accommodate high level requirements of availability or consistency.

This paper presents an adaptive control layer on top of such distributed key-value stores to free applications from low level configurations. This layer provides necessary encapsulation between configuration parameters and performance requirements from application level. Instead of working directly with these parameters, applications just need to specify their Service Level Agreement, or SLA, for availability or consistency. The idea of SLAs for availability and consistency has recently been proposed and studied [8, 11–13]. Given an availability SLA, the adaptive techniques make sure that it meets the availability requirement, while optimizing consistency. Similarly, in case of a consistency SLA, the system meets the consistency while optimizing availability. In other words, the control layer continuously adapts to network conditions and system status to guarantee that requirements of one metric are met while giving the best possible performance for the other. Availability SLAs are useful when fast response time is critical while some staleness is tolerable such as in the Netflix video tracking application [14], online advertising [15], or shopping cart applications [12]. On the other hand, consistency SLAs are beneficial to applications that desire bounded staleness in their results but also want best-possible response time. A web search engine [12] would be a notable example of such applications.

As a proof of concept, we implemented our strategy into Riak [5]. Our experiments with YCSB [16], under realistic workloads, shows that the algorithm is able to quickly modify its configurations to meet the availability or consistency SLA. When network conditions change, the system also quickly adapts in order to maintain the SLA. On the other hand, if the network is stable, the system's consistency and availability measurements converge to their respected equilibriums.

We make the following technical contributions:

- We propose an adaptive strategy to automatically control distributed key-value stores. It acts as a layer on top of these systems, allowing applications to specify an availability or consistency SLA. Under changing network conditions and system load, the algorithm guarantees that the SLA is met while optimizing the other metric.

- We integrate this technique into the popular key-value store Riak as a proof of concept.

- We present experimental results from benchmarks done using YCSB against the above system under realistic network conditions. With this, we find that the system is able to converge quickly to requirements specified by the SLA despite rapid changes in the network.

CHAPTER 2

DESIGN OF THE ADAPTIVE CONTROLLER

This chapter shows how to design an adaptive control layer for distributed key-value stores. We first explain how SLAs fit into our control strategy. We then elaborate on how the algorithm tunes its configuration knob to control the system. Finally, we talk about our considerations about selecting a control knob.

2.1 Service Legal Agreements

Similar to [11, 12], in order to take advantage of the system, an application first needs to specify its SLA. If the SLA is availability, our adaptive layer will optimize the consistency while meeting the SLA requirement. Similarly, if the SLA is consistency, it will optimize availability while meeting the SLA.

Availability and consistency can be measured in different ways using different metrics. On the other hand, metrics depend on the application. Therefore, we decide not to enforce any metrics into our adaptive control layer. Instead, we give applications the ultimate choice as to whether the system is meeting their requirements. In detail, instead of merely supplying a number as an SLA, applications need to supply an *SLAObject* (in Python) that implements the two functions below:

1. **isAvailabilitySLA()** function to indicate whether the SLA is availability or consistency.

2. **isSLASatisfied()** function that takes a list of read and write operations (and their results), measures the system's performance based on these operations, and returns whether the SLA is met.

Our controller treats such an SLA object as a black box to determine the system's performance and make changes accordingly. The first function is called once during initialization to obtain the type of SLA. The second function is then called routinely with the updated operation logs to check if the SLA is currently satisfied. Since our system is not bounded to

any particular metrics, it can fit into a much bigger range of applications. The prototype of SLAObject is shown in Prototype 1.

Prototype 1 SLAObject

```
function isAvailabilitySLA():
  return True if SLA is availability, False if consistency

function isSLASatisfied(log L):
  return True if SLA is met, False otherwise
```

An example for an availability SLA is percentage of *up time* on average [17]. A definition of up time for distributed stores can be when the round trip time of an operation is within a specified limit. Consider the case when an application demands 99% up time with a limit of 200 ms. This means that 99% of all reads and writes need to be answered within 200 ms. The application would need to supply an SLAObject whose isAvailabilitySLA() always returns True and isSLASatisfied() computes whether this condition is satisfied.

Read My Writes [18,19] can be used as an example for consistency SLA. Read My Writes guarantees that a read from a client only returns its previous writes or writes that happened after its latest write. An application can take advantage of Read My Writes by requiring that at least a certain percentage, say 90%, of its reads satisfy the property. For this SLAObject, isAvailabilitySLA() always returns False and isSLASatisfied() needs to check if 90% of the reads is Read My Writes.

2.2 Control Algorithm

2.2.1 Overview

Given an SLAObject, our adaptive algorithm works as a loop. At each iteration, it calls the supplied isSLASatisfied() function with the operation log to check the system's performance and changes its control knob appropriately to ensure convergence to the SLA. To understand how the algorithm interacts with the SLAObject, consider the case of availability SLA discussed in Section 2.1. Let us say that the application still requests 99% up time with a limit of 200 ms. If the system is violating the SLA, say, achieving only 95% up time, the call to isSLASatisfied() will return False. Our algorithm will then turn the control knob in a way to move availability closer to the 99% SLA. However, if the system is performing better than required, having 99.5% up time for example, isSLASatisfied() will return True. In this case,

the algorithm will turn the knob to the opposite direction, giving up some availability for better consistency. Note that in both cases, the control algorithm does not have any quantitative information about the actual performance of the system, just whether it is meeting the SLA or not.

In our design, the control layer sends operations to the underlying key-value store during each iteration. The operation log used by isAvailabilitySLA() contains only these reads and writes. This active measurement scheme has been suggested by [11]. It gives us more control over the injection rate, which results in more uniform convergence rate over time. In contrast, a passive scheme where the control layer instead uses only operations from the application might result in volatility in injection rate. In an extreme case, this might cause starvation of the control layer, during which it does not have enough operations to reliably determine the system's performance. This means either the controller would need to inject more queries, or convergence would slow down.

Procedure 1 depicts the control loop for the availability SLA case. The control loop for a consistency SLA is shown in Procedure 2.

procedure 1 Adaptive control for availability SLA

Input: Database D, SLAObject sla: sla.isAvailabilitySLA()=**true**

```
1: while true do
2:     Run n operations on D, collect log L
3:     if sla.isSLASatisfied(L) then
4:         Change control knob to reduce availability, thus improving consistency
5:     else
6:         Change control knob to improve availability, thus reducing consistency
7:     end if
8: end while
```

procedure 2 Adaptive control for consistency SLA

Input: Database D, SLAObject sla: sla.isAvailabilitySLA()=**false**

```
1: while true do
2:     Run n operations on D, collect log L
3:     if sla.isSLASatisfied(L) then
4:         Change control knob to reduce consistency, thus improving availability
5:     else
6:         Change control knob to improve consistency, thus reducing availability
7:     end if
8: end while
```

2.2.2 Read Delay as a Control Knob

Figure 2.1 shows the design of Riak's pipeline during a read or write operation. When a *client* (front-end) issues such a request, it selects one node in the cluster to send the request to. This node, called *coordinator*, acts as a point of contact for the client throughout the operation. Based on the key found in the request, the coordinator forwards it to the nodes (typically 3) that are holding the actual data. These nodes are called *replicas*. There is a total of three replicas in the figure, but this number might vary. Upon receiving the request from the coordinator, each replica processes the request and returns to the coordinator with the result. If the request is a write, the replicas will apply the change to their local data. If the request is a read, each of them will send its local data to the coordinator. The coordinator will then merge these responses using some given merge rule and return the final result to the client. An example of a rule for merging is to select the response with highest timestamp. While the figure only depicts the pipeline for Riak, we find that many other systems [6–10] use a similar design.

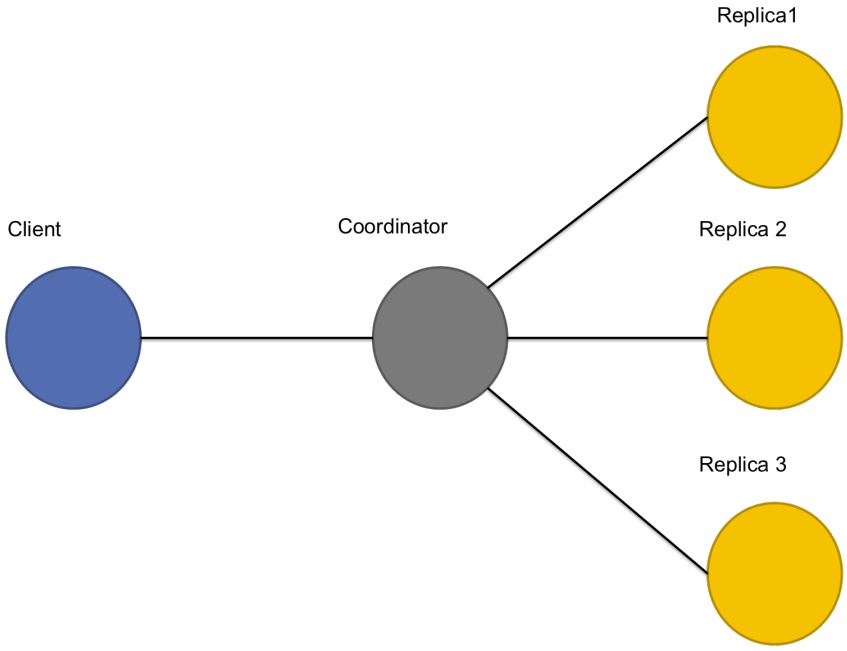
For our control strategy, we primarily rely on *read delay*, a knob that has been studied in literature [20–22]. Read delay affects the read path by making the coordinator artificially wait for a specified amount of time before forwarding the read request to the replicas. The idea of read delay is to give the replicas some time to converge to the latest write. A higher read delay value gives the replicas more time which translates to more consistency. However, the client also needs to wait longer for a response, hence lower availability. On the other hand, a lower value of read delay leads to lower consistency and higher availability.

Controlling the system's performance is rather straightforward with read delay. Consider the case where the SLA is availability. If the isSLASatisfied() function returns $True$, indicating that we are satisfying the SLA, we need to increase read delay to optimize for consistency. If the isSLASatisfied() function instead returns $False$, suggesting that the SLA is being violated, the algorithm will decrease read delay to improve availability. Similarly, when the SLA is consistency, we need to decrease read delay when isSLASatisfied() returns $True$ and increase it otherwise.

2.2.3 Control Algorithm in Detail

Suppose that we have two functions, $increaseRD$ and $decreaseRD$, which modify the current read delay appropriately so that the system performance moves closer to a given SLA. We can then use these two functions and combine procedures 1 and 2 to make a generalized version for both availability and consistency SLA as shown in procedure 3.

Figure 2.1: Riak pipeline



procedure 3 Generalized adaptive control

Input: Database D, SLAObject sla, Initial read delay rd

if sla.isAvailabilitySLA() **then** # sla is availability
    $improve \leftarrow decreaseRD$
    $reduce \leftarrow increaseRD$
else # sla is consistency
    $improve \leftarrow increaseRD$
    $reduce \leftarrow decreaseRD$
end if
while true do
    Run n operations on D, collect log L
    if sla.isSLASatisfied(L) **then**
        $reduce()$ # Use rd to reduce the performance
    else
        $improve()$ # Use rd to improve the performance
    end if
end while

Next, we discuss how to implement increaseRD and decreaseRD. The naive approach of changing read delay by the smallest unit increment (e.g., 1 ms) would be ideal to avoid high fluctuation at convergence. However, this approach would take a long time to converge if the SLA is far from the current performance. Therefore, we employ a multiplicative approach for changing read delay.

The multiplicative strategy starts with unit steps and exponentially increases the step size to grow the rate at which the performance approaches the SLA. When it crosses the SLA, the step size is reset back to unit increment and its direction is changed. The step size then again grows exponentially until it crosses SLA, and so on. The multiplicative strategy makes the performance approach the SLA faster. However, it is also subject to overstep which brings the performance too far on the opposite side of the SLA, resulting in longer convergence time. To prevent such large step sizes, we put a limit on the maximum size of a step. When a step reaches this maximum size, it does not grow anymore and read delay is changed incrementally. Procedure 4 extends procedure 3 by implementing multiplicative strategy into increaseRD and decreaseRD.

At convergence in a perfectly stable network, read delay will oscillate within the [-1,+1] range of the optimal read delay value. For example, if a read delay of 10.5 milliseconds would match the SLA exactly, the multiplicative strategy will alternate the actual read delay between 10 and 11 milliseconds. Figure 2.2 depicts an example of such case. The growth factor for step size is 2. Read delay starts from 0 ms and gradually converges to its equilibrium, oscillating between 10 and 11 ms.

However, it is important to note that if the multiplicative strategy is strictly followed, the system might be prevented from reaching convergence. For example, let us say the optimal value for read delay is instead 7.5 ms and we again start with a read delay of 0 ms. Since the growth factor for step size is 2, read delay will be sequentially changed to 1, 3 and 7ms, before crossing the optimal SLA at 15 ms. After that, it changes direction and starts from step size of 1 ms again, reaching 14, 12, and 8 ms. Since the step size is 8 ms at this point, read delay again crosses the optimal SLA to reach the original value of 0 ms. The system will continue doing so indefinitely without converging. Figure 2.3 depicts this case. To resolve this problem, we add some unit steps randomly in between multiplicative ones.

2.3 Read Delay vs. Other Potential Knobs

This section explains why read delay is chosen as our main control knob over some other interesting configurations, including consistency level, replication factor and read repair chance.

procedure 4 Generalized adaptive control, multiplicative strategy

Input: Database D, SLAObject sla, Initial read delay rd, Growth factor γ, Step size limit MAX_SS

$step\_size \leftarrow 0$
if sla.isAvailabilitySLA() **then** # sla is availability
    $improve \leftarrow decreaseRD$
    $reduce \leftarrow increaseRD$
else # sla is consistency
    $improve \leftarrow increaseRD$
    $reduce \leftarrow decreaseRD$
end if
while true do
    Run n operations on D, collect log L
    if sla.isSLASatisfied(L) **then**
        $reduce()$ # Use rd to reduce the performance
    else
        $improve()$ # Use rd to improve the performance
    end if
end while
function INCREASERD
    if $step\_size \leq 0$ **then** # rd is currently decreasing
        $step\_size \leftarrow 1$ # change direction, reset $step\_size$
    else # rd is currently increasing
        $step\_size \leftarrow \min(step\_size \times \gamma, MAX\_SS)$
    end if
    $rd \leftarrow rd + step\_size$
end function
function DECREASERD
    if $step\_size \geq 0$ **then** # rd is currently increasing
        $step\_size \leftarrow -1$ # change direction, reset $step\_size$
    else # rd is currently decreasing
        $step\_size \leftarrow \max(step\_size \times \gamma, -MAX\_SS)$
    end if
    $rd \leftarrow \max(rd + step\_size, 0)$ # We do not want rd to be negative
end function

Figure 2.2: Changes of read delay over time

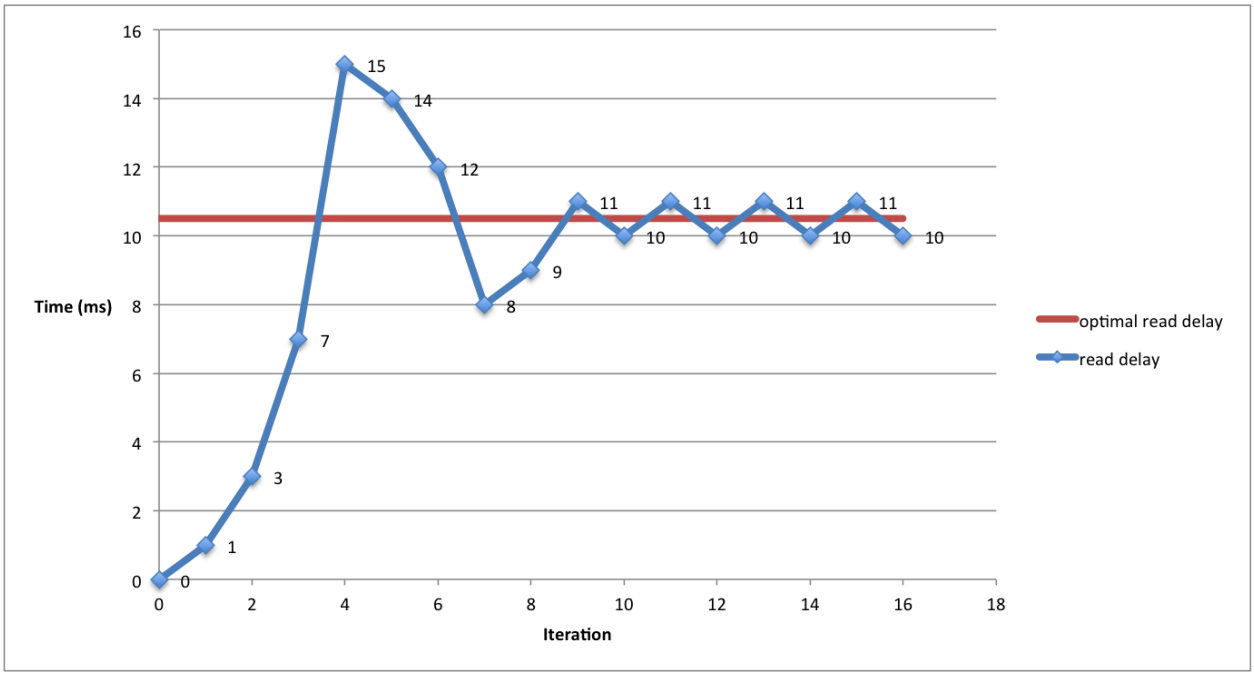
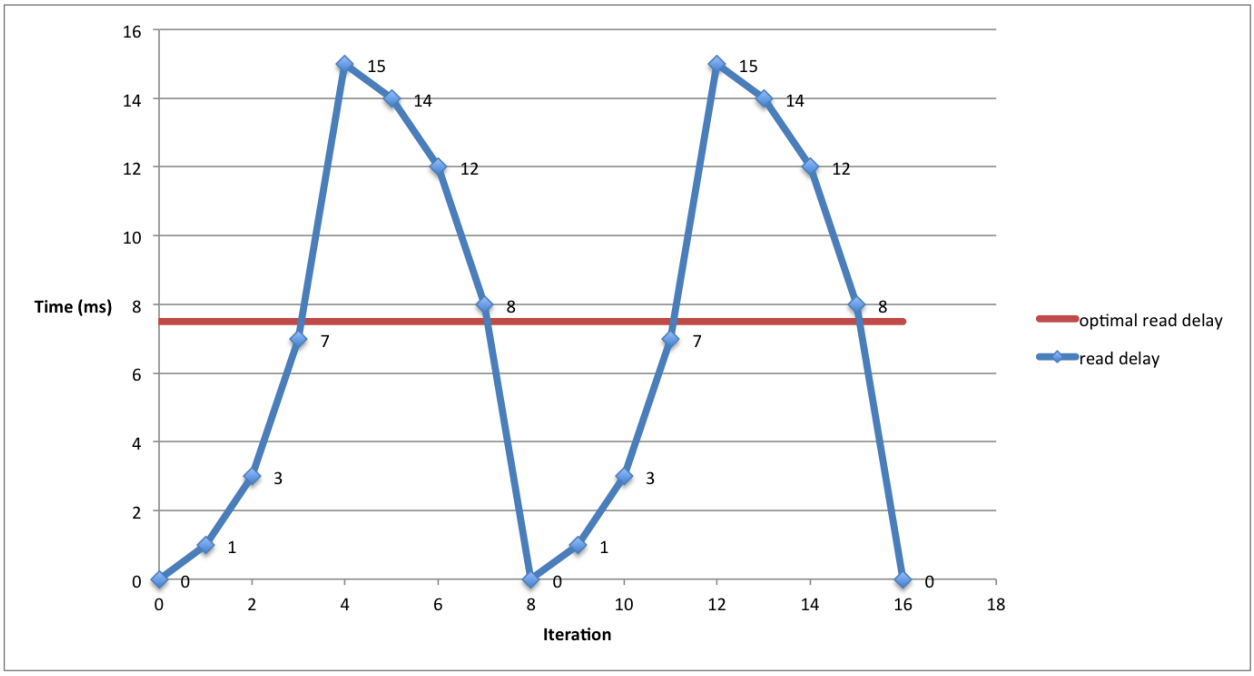


Figure 2.3: Read delay, nonconvergent case



In our implementation, we pick read delay over *consistency level* due to two reasons. First, read delay is fine-grained in time, and can be set to any non-negative values. Read delay gives us a larger set of possible configurations compared to other knobs. For example, there are only three possible values for consistency level if replication factor is three. Second, read delay is non-blocking, whereas consistency level is a blocking knob. For example, if a client issues a read with the consistency level set to ALL and some replicas for the corresponding key are on failed nodes, the coordinator will have to either be blocked waiting or return an error. So a higher consistency level does not guarantee a "better" value in response.

Some other knobs that we have taken into consideration are *replication factor* and *read repair chance*, both of which have some major drawbacks. Replication factor is mostly set only once during initialization. Modification of replication factor is potentially a very expensive process as it would involve transferring a huge amount of data between replicas. Replication factor also doesn't give a large set of possible configurations as the recommended values for most distributed key-value stores are between three and five. Because of these reasons, replication factor doesn't play any role in our adaptive algorithm.

Meanwhile, read repair is the mechanism for the coordinator to actively push the most recent version to any out-of-date replicas. This process generally happens in the background after a read whose responses from the replicas disagree with each other. Read repair chance governs the probability at which the coordinator performs this repairing process. Thus, a read repair chance of 0 implies no read repair, and replicas will be updated only by subsequent writes. A read repair of 1 implies that all stale reads result in a read repair. Read repair chance is available in most key-value stores such as Cassandra and Riak. A high value of read repair chance would result in more consistent system, whereas a low read repair chance would give less consistency. However, since read repair does not directly affect the read path, it does not affect availability. Therefore, we currently don't take advantage of read repair chance.

As stated above, the current version of our adaptive strategy mainly uses read delay as a control knob. However, read delay has its own disadvantage that it cannot be negative. This means that when read delay is zero, the system reaches its peak availability (or worst consistency) performance as we cannot further decrease read delay. This warrants an interesting direction for future work to incorporate other control knobs into the adaptive algorithm to achieve wider, finer-grained scales of availability and consistency.

CHAPTER 3

IMPLEMENTATION

In this chapter, we discuss how to implement the adaptive strategy into distributed key-value stores. To illustrate the idea, we have chosen Basho's Riak as the based store and integrated the strategy into it.

3.1 Central Controller

In order to incorporate the adaptive layer to key-value stores, we use a central controller that will be running adaptive control loop discussed in the previous chapter. This central server continuously issues reads and writes via the clients. It also collects operation logs from all clients and parses the logs to select k latest operations. The list of these operations will be used as the parameter for the SLAObject (Section 2.1) to compare the system's current performance to the SLA. k is a system setting and can be configured by applications. In our implementation, we use $k = 20000$.

Since the central controller is just one server, it is a single point of failure. When the controller crashes, the system will stop working properly. In addition, if the operations logs are big enough, processing them might take a long time using just one server. Furthermore, some SLA metrics might require expensive computation. Therefore, this approach is rather limited. In future work, we would like to improve this aspect of the system, taking advantage of a cluster of servers as the controller instead of just a single node.

3.2 Availability and Consistency SLAs

As discussed above, we provide applications with the ability to have their own SLA metrics. For our experiments of the Riak implementation, however, we use the probabilistic availability and consistency metrics as mentioned in [11].

For availability, given a number t_a, the probability of unavailability of the system, p_{ua}, is calculated as the ratio of operations that are *not* returned within t_a milliseconds. For

example, if t_a is 150 ms and 15000 out of the 20000 operations are returned within 150 ms, then $p_{ua} = 0.75$. For all experiments, we use $t_a = 150$. The function isSLASatisfied() for this availability SLAObject compares the current performance p_{ua} with a given unavailability SLA, $p_{ua}(sla)$, and returns $True$ if and only if $p_{ua} \leq p_{ua}(sla)$.

For consistency, we serialize read and write operations based on their timestamps. Let's define a *fresh read* to be a read that returns the value of the latest write before it. All other reads that do not return the latest write value are *stale reads*. Then the probability of inconsistency of the system, p_{ic}, is the ratio of stale reads over all read operations. The consistency SLAObject implements its isSLASatisfied() to compare the current inconsistency measurement p_{ic} with a given inconsistency SLA, $p_{ic}(sla)$, and return $True$ if and only if $p_{ic} \leq p_{ic}(sla)$.

3.3 Modification to Riak

We use YCSB [16] as our clients for the experiments. There are 2 ways to send requests to the Riak cluster. The first one is to send HTTP REST queries via Riak's webmachine - the system's HTTP interface. The second one is to send binary requests via the protobuf interface. For our experiments, we take advantage of the first method due to its simplicity. Based on Riak v1.4.2, we introduce a system wide parameter for read delay. This parameter is passed from YCSB clients to the Riak cluster as a standard field in the query string for each read (i.e.:http://ReadURL?read_delay=...). We modified Riak code to accept this parameter and cause the coordinator to delay the read accordingly. This modification requires less than 50 lines of Erlang code in Riak.

Since YCSB does not have a built-in connector for Riak, we also develop a separate Riak client for YCSB from scratch. This client implements standard YCSB operations (read, insert, scan, update) using Riak HTTP interface as discussed above. The connector is about 250 lines of Java code.

CHAPTER 4

EVALUATION

This chapter discusses the evaluation of our modified Riak system. We first explains the setup of our experiments. We then present the results of the system under both Sharp Network Jump and Realistic Delay Distributions.

4.1 Experiment Setup

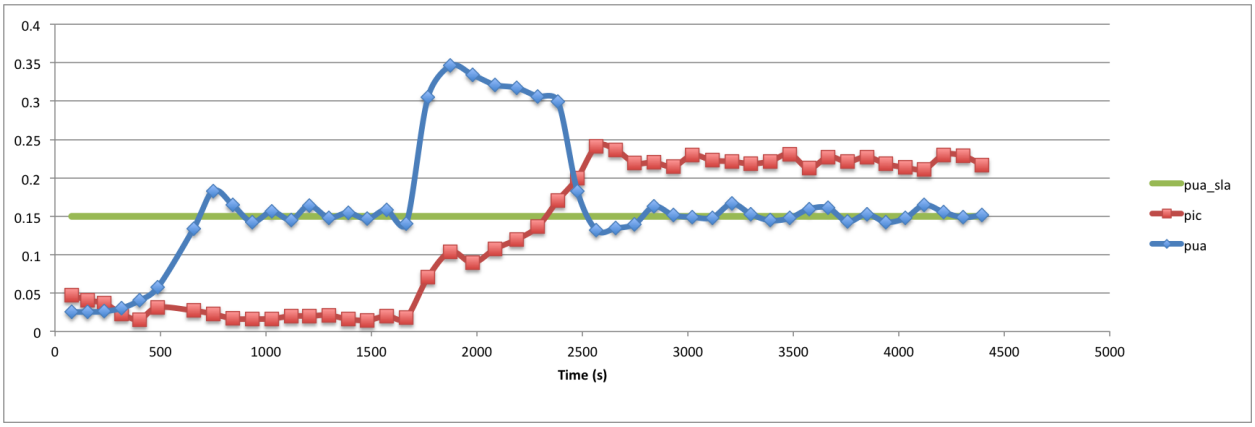
We use YCSB v0.1.4 to inject operations to Riak. The consistency level for all read and write operations is ONE. The cluster contains 5 Emulab pc3000 servers [23], each with a 64-bit Xeon processor, 2GB RAM, and 146GB disk space. We use a LAN (star topology) network, with 100 Mbps bandwidth and initial server-to-server delay of 0 ms, dynamically controlled using traffic shaping [24].

Each server in the cluster is a Riak node. In addition, 8 YCSB clients are run on each of these servers. For each operation, a client selects a random Riak node as the coordinator to send the query. We use a zipfian, read heavy distribution with 80% reads and 20% writes. The size of each record is 2048B and the size of each key is 13B. We keep the default replication factor of 3. The throughput is 1000 ops/s.

One server is selected to run the central controller as discussed in section 3.1. We also use NTP [25] to synchronize clocks within 1 ms - reasonable for a single datacenter. This clock skew can be made tighter by using atomic or GPS clocks [26]. However, the synchronization is needed just for calculation of the consistency SLA function in 3.2. The adaptive layer alone does not require clock synchronization.

For the multiplicative strategy, we use growth factor $\gamma = 2$ and cap the step size at $MAS\_SS = 10$ (Section 2.2.3). This ensures fast convergence rate without being subject to huge overstep.

Figure 4.1: Availability SLA under Sharp Network Jump



4.2 Experiments with Sharp Network Jump

We present the results for our experiments with the system under Sharp Network Jump. The network is set up such that there is no artificial delay between nodes at first. Later, we inject a node-to-LAN delay of 26 ms for 3 out of 5 nodes in the cluster. This means that out of the 10 node-to-node connections:

- 3 connections will have a 52 ms delay
- 6 connections will have a 26 ms delay
- 1 connections will have a 0 ms delay

Figure 4.1 depicts the performance of the system for an availability SLA. We use $p_{ua}(sla) = 0.15$. Initially, the system detects that p_{ua} is better than $p_{ua}(sla)$, so it quickly pushes that up to optimize for consistency. The plot also shows that the adaptive strategy is able to keep p_{ua} very close to $p_{ua}(sla)$ under a stable network from time 900 s to 1700 s. At time 1700 s, we use Emulab traffic sharping tool to add the mentioned delay, resulting in higher values for both p_{ua} and p_{ic}. The system again quickly brings p_{ua} back to $p_{ua}(sla)$, giving up some consistency this time. Again, p_{ua} is maintained along the $p_{ua}(sla)$ line.

The result is similar for an consistency SLA, as shown in Figure 4.2. We use $p_{ic}(sla) = 0.15$. Network delay is injected at around time 2100 s. Both before and after the delay is injected, our system is able to meet the SLA requirement. For both experiments under Sharp Network Jump, we observe that the system takes about 500 s to 700 s to converge to the SLA. Currently the system is implemented such that it only measures the current performance and modifies the control knob accordingly when there are enough operation logs as specified by k (section 3.1). Therefore, a lower value for k would result in shorter convergence time.

Figure 4.2: Consistency SLA under Sharp Network Jump

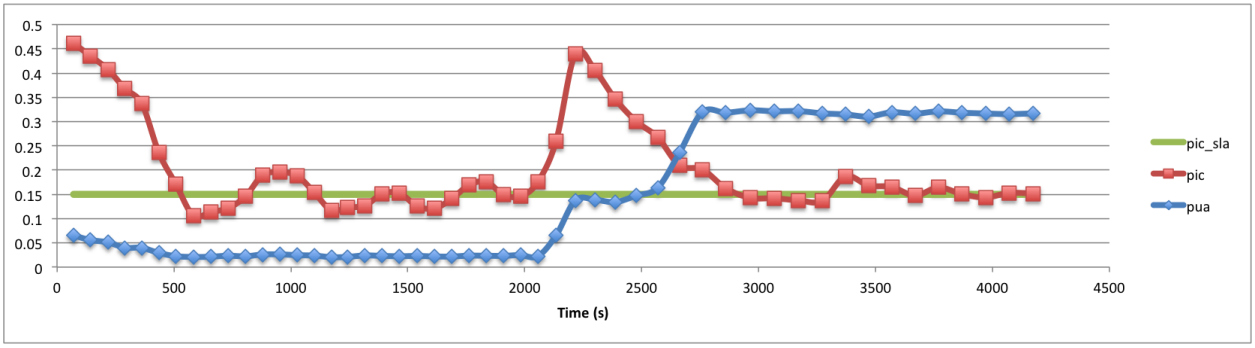


Figure 4.3: Availability SLA under Sharp Network Jump, Scatter plot

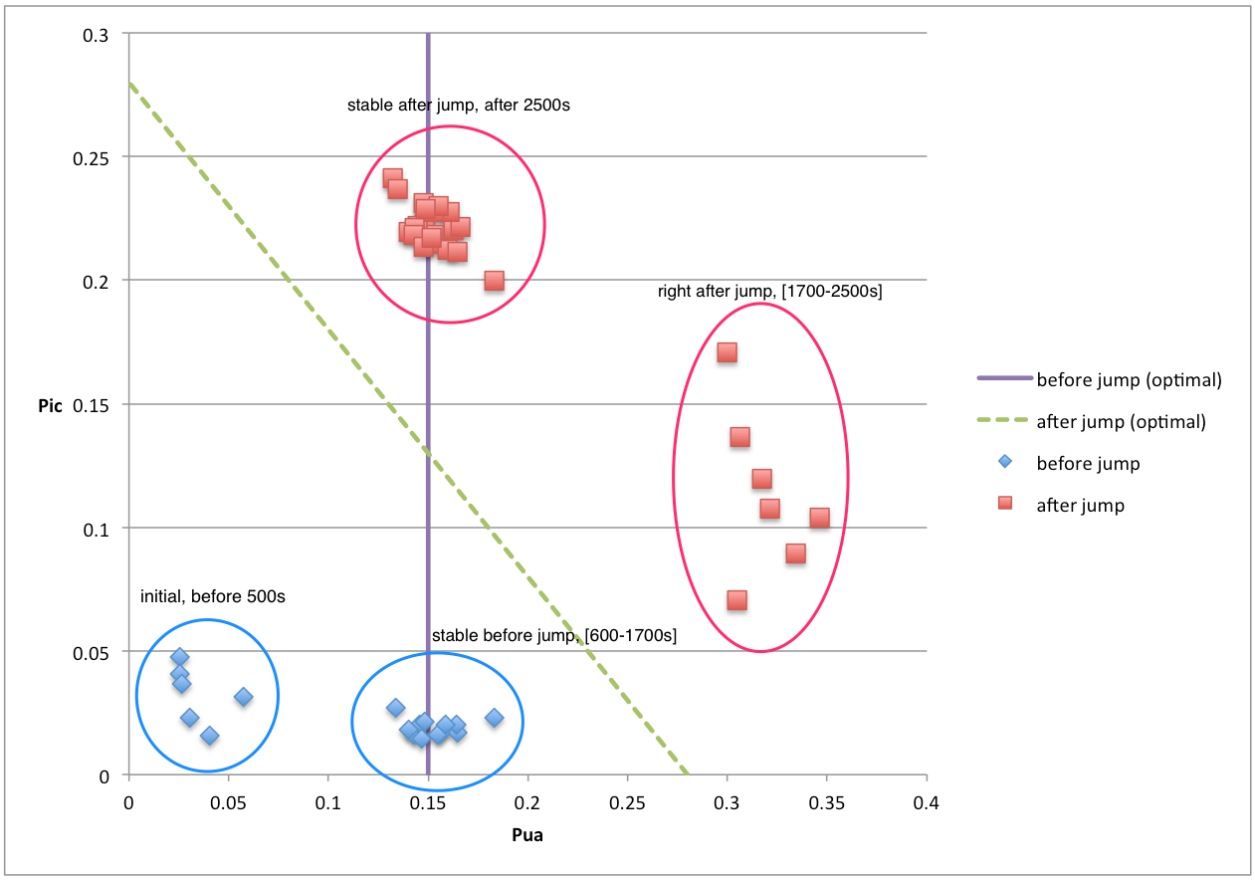


Figure 4.4: Consistency SLA under Sharp Network Jump, Scatter plot

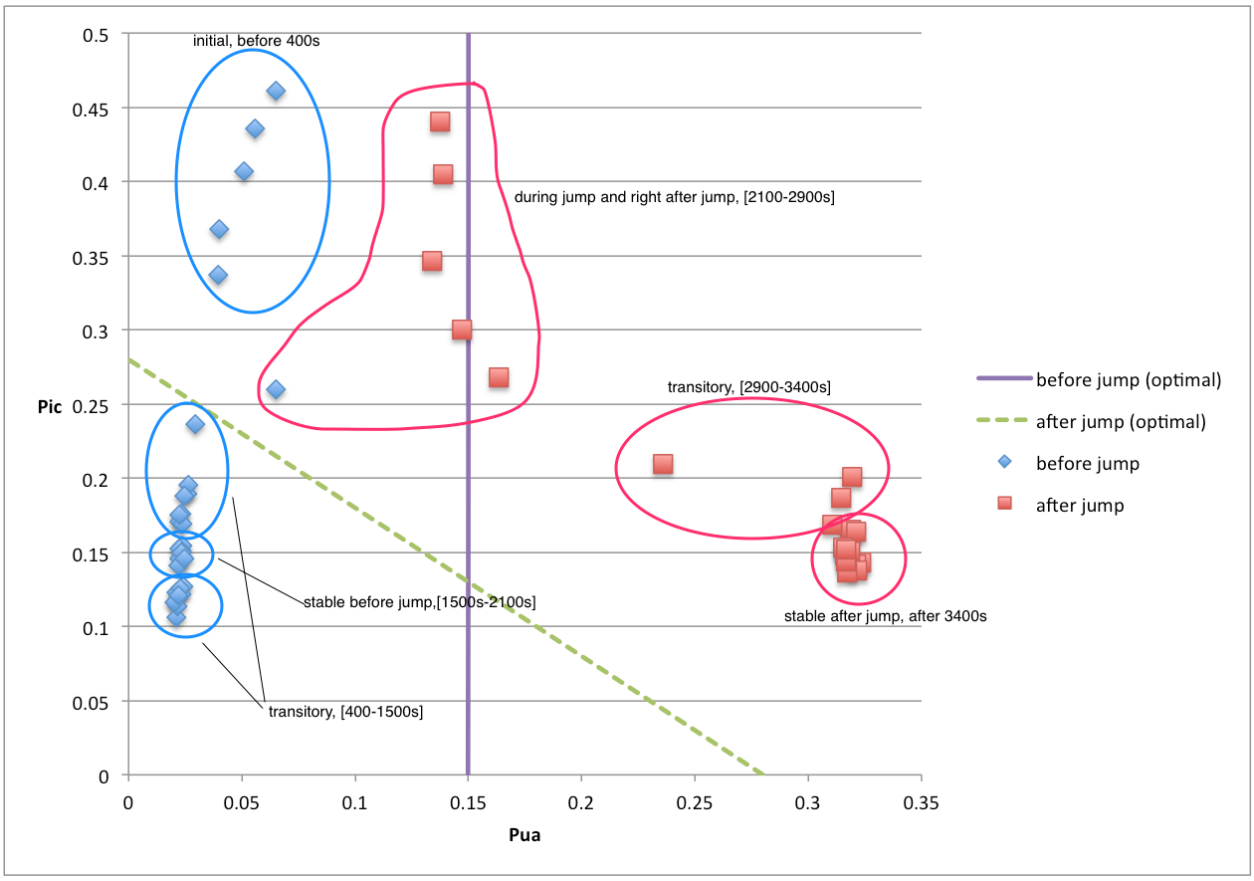


Figure 4.5: Availability SLA under Realistic Delay Distribution

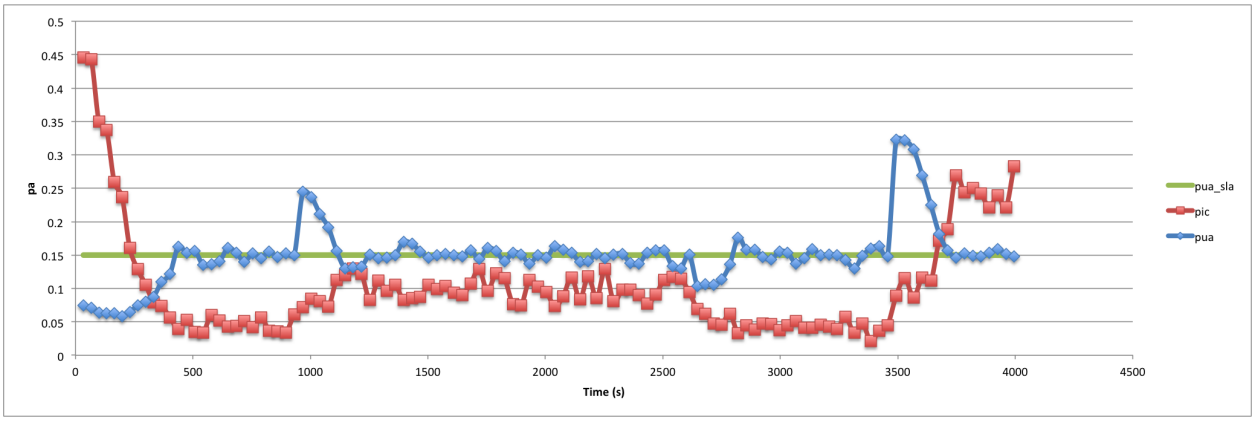
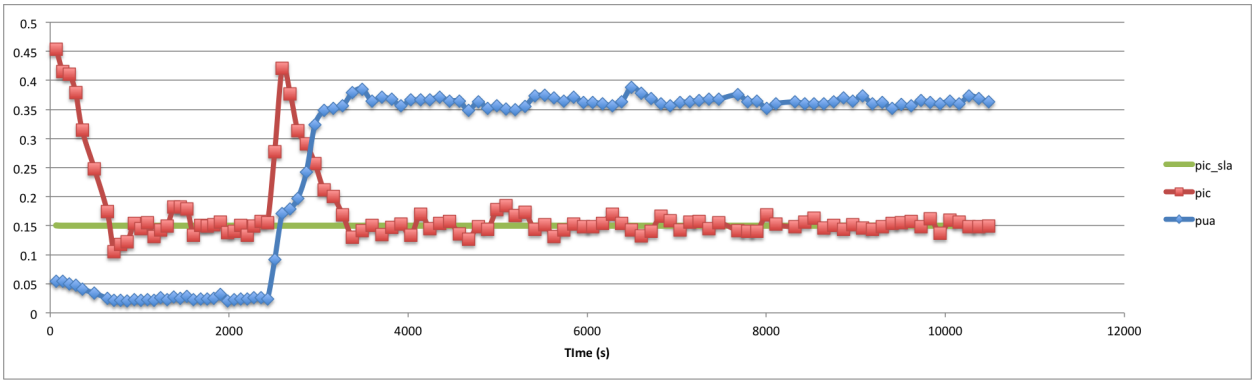


Figure 4.6: Consistency SLA under Realistic Delay Distribution



[11] proves that for under a specific network condition, there is a limit on how much availability and consistency a distributed store can provide. Figure 4.3 and Figure 4.4 help to visualize how close our system is to this theoretically optimal-achievable envelope. Figure 4.3 shows the two achievable envelopes as piecewise linear segments (named before and after the jump) and the (p_{ua}, p_{ic}) data points from our run in Figure 4.1. Similarly, Figure 4.4 shows the same but with data taken from Figure 4.2. The figures annotate the clusters of data points by their time interval. We observe that the system's performance is close to the optimal-achievable envelopes.

4.3 Experiments with Realistic Delay Distributions

This section discusses the performance of the system under more realistic delay distributions. We use lognormal distributions for network delay for our experiments in this sections. In order to do this, at each node, we delay each outgoing packet from the network interface so

that these delays form a lognormal distribution with a given pair of μ and σ.

Figure 4.5 presents a run with availability SLA $p_{ua}(sla) = 0.15$. Initially, $\mu = 1$ and $\sigma = 0.2$. As we can see, the system quickly brings p_{ua} up along the $p_{ua}(sla)$ line, resulting in lower p_{ic}. At time 950 s, we change μ to 2.8 (while keeping the same σ). The change in network delay causes a jump in both p_{ua} and p_{ic}. The system is able to quickly fix this, pushing $p_{ua}(sla)$ to the SLA again. At time 2600 s, we revert μ back to 1 and at time 3450 s, we set μ to a higher value of 3.1, both of which are handled well by the adaptive strategy. The plot shows that the system takes about 200 s to converge for these changes. In addition, as under Sharp Network Jump, when there are not any changes in the network, p_{ua} and p_{ic} lines are stable.

Next, a case with consistency SLA is shown in Figure 4.6. For this experiment, $p_{ic}(sla) = 0.15$. We begin with $\mu = 1$ and $\sigma = 0.1$. At time 2500 s, μ is changed to 4. Similar to previous experiments, we observe that our system quickly meets the SLA and it expectedly optimizes for availability if the current consistency is better than required.

CHAPTER 5

RELATED WORK

There have been some research regarding adaptive key-value stores. FRACS [27] is a self-tuned consistency enforcer based on buffering updates at replicas up to some slateness. AQuA [28] controls consistency by dividing replicas into "strong" and "weak" consistency groups and continuously moving them between these groups. TACT [29] is a middleware layer that enforces arbitrary consistency bounds among replicas by limiting the number of outstanding writes at the replicas (*order error*) and bounding write propagation delay (*staleness*). All the mentioned systems provide *best-effort* behavior for consistency, within the latency bounds. Consistency measurements in these systems are limited to some given metrics. In comparison, our system allows applications to supply any availability or consistency SLAObjects and treats them as black boxes to determine if the requirements are met. Consistency level has been used in [30] as a control knob to deal with node failures and changes in network conditions. However, as discussed in Section 2.3, consistency level is a weaker knob than read delay for various reasons.

The need to control the tradeoffs between consistency and availability has been suggested by [31]. Many distributed storage systems are able to offer multiple levels of consistency [32–35]. It has been studied that different applications might benefit from different consistency guarantees [36, 37]. PBS [20] proposed a probabilistic consistency model for quorum-based stores, but did not focus on availability. Pileus [12] considered families of consistency and availability SLAs, but only for geo-distributed settings. To the best of our knowledge, none of these systems allow applications to define their own SLA metrics. In general, researchers have proposed a wide range of consistency models for distributed databases [19, 29, 38–41].

SLA is a concept that has been explored in many areas, including differentiated services [42], cloud services [8], and utility computing [43]. Several algorithms for checking consistency in distributed stores have been developed [44, 45]. In general, tradeoffs have been studied between storage cost and repair bandwidth [46], between energy cost, fairness and delay [47], and between throughput and delay in wireless network [48].

CHAPTER 6

CONCLUSION AND FUTURE WORK

We have presented an adaptive approach for distributed stores to control the trade-off between consistency and availability. Our approach allows applications to specify high-level consistency or availability requirements by supplying their own SLA functions. It guarantees that the SLA is met while trying to give the best possible performance for the other metric, even under changing network conditions. This approach has been shown to be practically viable through our implementation based on the popular distributed database Riak. Our experiments with the implementation demonstrated that the algorithm converges fast and stably. At equilibrium, the system met its SLAs and the overall performance was close to its theoretical limits.

Future work could be directed toward several directions. First, we believe that the approach can be improved by incorporating more control knobs, instead of just using read delay. This would give us a finer-grained, wider range of achievable consistency and availability. Second, the use of just one physical node as a controller is rather limited as discussed in Section 3.1. A multi-node cluster could be utilized for this purpose to achieve better fault-tolerance and processing time. Another solution to improve fault-tolerance is to use a secondary node as a backup controller (similar to Hadoop's Namenode [49, 50]).

Section 2.2 explains why we have chosen to actively measure the performance of the system. Future work could involve exploring how to effectively incorporate a passive measurement scheme [11] and comparing the two approaches. In addition, a hybrid approach in which the control layer uses operation logs from the application but also injects reads and writes in case of starvation could potentially provide the system with more stability and efficiency.

Last but not least, the control layer is currently designed exclusively for distributed key-value stores whose all nodes live in the same datacenter. Further studies could be directed toward satisfying SLA in geo-distributed settings [12].

REFERENCES

[1] E. Brewer, "A certain freedom: Thoughts on the CAP theorem," in *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2010, pp. 335–335.

[2] E. A. Brewer, "Towards robust distributed systems (Invited Talk)," in *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2000.

[3] S. Gilbert and N. A. Lynch, "Perspectives on the CAP theorem," *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[4] N. Lynch and S. Gilbert, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[5] "Basho Riak," http://basho.com/riak/.

[6] "Cassandra," http://cassandra.apache.org/.

[7] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 205–220.

[9] "DynamoDB," http://aws.amazon.com/dynamodb/.

[10] "Project Voldemort, A distributed database," http://www.project-voldemort.com/voldemort/.

[11] M. R. Rahman, L. Tseng, S. Nguyen, I. Gupta, and N. Vaidya, "Probabilistic cap and timely adaptive key-value stores," 2014.

[12] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 309–324.

[13] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh, "SCADS: scale-independent storage for social computing applications," in *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009. [Online]. Available: http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_86.pdf

[14] A. Cockcroft, "Dystopia as a service (Invited Talk)," in *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2013.

[15] "Read-time online advertising with Turn," https://www.turn.com/whyturn#technology.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2010, pp. 143–154.

[17] A. B. Brown and D. A. Patterson, "Towards availability benchmarks: A case study of software RAID systems," in *Proceedings of the General Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA*, 2000. [Online]. Available: http://www.usenix.org/publications/library/proceedings/usenix2000/general/brown.html pp. 263–276.

[18] D. Terry, "Replicated data consistency explained through baseball," *Commun. ACM*, vol. 56, no. 12, pp. 82–89, 2013. [Online]. Available: http://doi.acm.org/10.1145/2500500

[19] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30, 1994*, 1994. [Online]. Available: http://dx.doi.org/10.1109/PDIS.1994.331722 pp. 140–149.

[20] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *VLDB Endowment*, vol. 5, no. 8, pp. 776–787, 2012.

[21] "Read-After-Write Consistency in Amazon S3," http://shlomoswidler.com/2009/12/read-after-write-consistency-in-amazon.html.

[22] W. Golab and J. J. Wylie, "Providing a measure representing an instantaneous data consistency level," Jan 2014, uS Patent Application 20140032504.

[23] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2002, pp. 255–270.

[24] "Emulab Traffic Shaping," http://www.uky.emulab.net/tutorial/docwrapper.php3?docname=advanced.html&printable=1.

[25] "Emulab NTP," http://goo.gl/SMk2uZ.

[26] J. C. Corbett et al., "Spanner: Google's globally-distributed database," in *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 251–264.

[27] C. Zhang and Z. Zhang, "Trading replication consistency for performance and availability: an adaptive approach," in *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, 2003. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICDCS.2003.1203520 pp. 687–695.

[28] S. Krishnamurthy, W. H. Sanders, and M. Cukier, "An adaptive quality of service aware middleware for replicated services," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 11, pp. 1112–1125, 2003. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TPDS.2003.1247672

[29] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Trans. Comput. Syst.*, vol. 20, no. 3, pp. 239–282, 2002. [Online]. Available: http://doi.acm.org/10.1145/566340.566342

[30] A. Davidson, A. Rubinstein, A. Todi, P. Bailis, and S. Venkataraman, "Adaptive hybrid quorums in practical settings," 2013, http://www.cs.berkeley.edu/~kubitron/courses/cs262a-F12/projects/reports/project12_report_ver2.pdf.

[31] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini, "Flex-kv: Enabling high-performance and flexible kv systems," in *Proceedings of the 2012 Workshop on Management of Big Data Systems*, ser. MBDS '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2378356.2378361 pp. 19–24.

[32] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, and C. T. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," *ACM Trans. Comput. Syst.*, vol. 27, no. 3, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629087.1629088

[33] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, 2011. [Online]. Available: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf pp. 223–234.

[34] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008. [Online]. Available: http://www.vldb.org/pvldb/1/1454167.pdf

[35] "Choosing consistency," http://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html.

[36] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: Pay only when it matters," *PVLDB*, vol. 2, no. 1, pp. 253–264, 2009. [Online]. Available: http://www.vldb.org/pvldb/2/vldb09-759.pdf

[37] X. Wang, S. Yang, S. Wang, X. Niu, and J. Xu, "An application-based adaptive replica consistency for cloud storage," in *GCC 2010, The Ninth International Conference on Grid and Cloud Computing, Nanjing, Jiangsu, China, 1-5 November 2010*, 2010. [Online]. Available: http://dx.doi.org/10.1109/GCC.2010.16 pp. 13–17.

[38] R. Alonso, D. Barbará, and H. Garcia-Molina, "Data caching issues in an information retrieval system," *ACM Trans. Database Syst.*, vol. 15, no. 3, pp. 359–384, 1990. [Online]. Available: http://doi.acm.org/10.1145/88636.87848

[39] D. Barbará and H. Garcia-Molina, "The demarcation protocol: A technique for maintaining constraints in distributed database systems," *VLDB J.*, vol. 3, no. 3, pp. 325–353, 1994. [Online]. Available: http://www.vldb.org/journal/VLDBJ3/P325.pdf

[40] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li pp. 265–278.

[41] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, 2011. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043593 pp. 401–416.

[42] C. Courcoubetis and V. Siris, "Managing and pricing service level agreements for differentiated services," in *Proc. International Workshop on Quality of Service (IWQoS)*, 1999, pp. 165–173.

[43] M. Buco, R. Chang, L. Luan, C. Ward, J. Wolf, and P. Yu, "Utility computing SLA management based upon business objectives," *IBM Systems Journal*, pp. 159–178, 2004.

[44] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie, "What consistency does your key-value store actually provide?" in *Proceedings of the Sixth Workshop on Hot Topics in System Dependability, HotDep 2010, Vancouver, BC, Canada, October 3, 2010*, 2010. [Online]. Available: https://www.usenix.org/conference/hotdep10/what-consistency-does-your-key-value-store-actually-provide

[45] W. M. Golab, X. Li, and M. A. Shah, "Analyzing consistency properties for fun and profit," in *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993806.1993834 pp. 197–206.

[46] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," in *IEEE Transactions on Information Theory, vol. 56, no. 9*, September 2010.

[47] S. Ren, Y. He, and F. Xu, "Provably-efficient job scheduling for energy and fairness in geographically distributed data centers," in *Proc. International Conference on Distributed Computing Systems (ICDCS)*, 2012, pp. 22–31.

[48] G. Sharma, R. Mazumdar, and N. B. Shroff, "Delay and capacity trade-offs in mobile ad hoc networks: A global perspective," *IEEE/ACM Transactions on Networking (ToN)*, vol. 15, no. 5, pp. 981–992, 2007.

[49] "Apache Hadoop," http://hadoop.apache.org/.

[50] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.