Characterizing and Adapting the Consistency-Latency Tradeoff in Distributed Key-value Stores [*]

Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, Nitin Vaidya
University of Illinois at Urbana-Champaign
{mrahman2, ltseng3, nguyenb1, indy, nhv}@illinois.edu

Paper Type: Research, Long

Abstract

The CAP theorem is a fundamental result that applies to distributed storage systems. In this paper, we first present probabilistic models to characterize the three important elements of the CAP theorem: consistency (C), availability or latency (A), and partition-tolerance (P). Then, we provide *quantitative* characterization of the tradeoff among these three elements. Next, we leverage this result to present a new system, called PCAP, which allows applications to specify either a latency SLA or a consistency SLA. The PCAP system automatically adapts, in real-time and under changing network conditions, to meet the SLA while optimizing the other C/A metric. We incorporated PCAP into two popular key-value stores – Apache Cassandra and Riak. Our experiments with these two deployments, under realistic workloads, reveal that the PCAP system satisfactorily meets SLAs, and performs close to the bounds dictated by our tradeoff analysis.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords Consistency, Adaptivity, Distributed Storage

1. Introduction

Distributed key-value stores (e.g., Cassandra [4], Riak [3], Dynamo [24], Voldemort [8]) are preferred by applications for whom eventual consistency suffices, but where high availability and low latency (i.e., fast reads and writes [11]) are paramount. Latency is a critical metric for such cloud services because latency is correlated to user satisfaction – for instance, a 500 ms increase in latency for operations at Google.com can cause a 20% drop in revenue [1]. At Amazon, this translates to a $6M yearly loss per added millisecond of latency [2].

These systems provide a weak notion of consistency called eventual consistency [14, 45]. At the same time, clients in such applications expect freshness, i.e., data returned by a read to a key should come from the latest writes done to that key by any client. For instance, Netflix uses Cassandra to track positions in each video [19], and freshness of data translates to accurate tracking and user satisfaction. This implies that clients care about a *time-based* notion of eventual consistency.[1]

The CAP theorem was proposed by Eric Brewer [17, 18], and later formally proved by Gilbert and Lynch [29, 40]. It essentially states that a system can choose at most two of three desirable properties: Consistency (C), Availability (A), and Partition tolerance (P). Fox and Brewer later proposed the Weak CAP Principle [27], which characterizes a finer-grained tradeoff between availability and consistency. Recently, Abadi [11] proposed to study the consistency-latency tradeoff, and unified the tradeoff with the CAP theorem. The unified result is called PACELC. It states that when a network partition occurs, one needs to choose between Availability and Consistency, otherwise the choice is between Latency and Consistency. We focus on the latter tradeoff as it is the common case. These prior results provided qualitative characterization of tradeoff between consistency and availability/latency, while we provide *quantitative* characterization of the tradeoff.

Concretely, traditional CAP literature tends to focus on situations where "hard" network partitions occur and the designer has to choose between C or A, e.g., in geo-distributed

[*] This work was supported in part by the following grants: NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, and a VMware Graduate Fellowship.

[1] Such client centric consistency models [43] differ from a system-centric consistency view, where consistency is related to restrictions on operation ordering [12].

Submission to SoCC '15, October, 2015, Hawaii.

datacenters. However, individual datacenters themselves suffer far more frequently from "soft" partitions [23], arising from periods of elevated message delays or loss rates (i.e., the "otherwise" part of PACELC). Neither the original CAP theorem nor the existing work on consistency in key-value stores (e.g., [15, 24, 30, 36, 38, 39, 42, 44, 45]) address such soft partitions for a single datacenter.

In this paper we first present probabilistic[2] models to characterize the three important elements: a soft partition, latency requirements, and consistency requirements. We then state and prove two impossibility theorems. All our models take timeliness into account. Our latency model specifies soft bounds on operation latencies, as might be provided by the application in an SLA (Service Level Agreement). Our consistency model captures the notion of data freshness returned by read operations. Our partition model describes propagation delays in the underlying network. The resulting theorems show which combinations of the parameters in these three models (partition, latency, consistency) make them impossible to achieve together.

Next, we build a system called PCAP (short for Probabilistic CAP) that allows applications of key-value stores to specify either a probabilistic latency SLA or a probabilistic consistency SLA. Given a probabilistic latency SLA, PCAP's adaptive techniques meet the specified operational latency requirement for client reads and writes, while optimizing the consistency. Similarly, given a probabilistic consistency SLA, PCAP meets the consistency while optimizing operational latency. PCAP does so under real and continuously changing network conditions. There are known use cases that would benefit from an latency SLA – these include the Netflix video tracking application [19], online advertising [9], and shopping cart applications [44] – each of these needs fast response times but is willing to tolerate some staleness. A known use case for consistency SLA is a Web search application [44], which desires search results with bounded staleness but would like to minimize the response time.

We have integrated our PCAP system into two key-value stores – Apache Cassandra [4] and Riak [3]. Our experiments with these two deployments, under realistic workloads from YCSB [20], reveal that the PCAP system satisfactorily meets a latency SLA (or consistency SLA), optimizes the consistency metric (respectively latency metric), performs reasonably close to the bounds dictated by our tradeoff analysis, and it scales well.

2. Consistency-Latency Tradeoff

We first present our probabilistic models for soft partition, latency and consistency. Then we present our impossibility results.

[2] By probabilistic, we mean the behavior is statistical over a long time period.

2.1 Models

To capture consistency, we defined a new notion called *t-freshness*, which is a form of eventual consistency. Consider a single key (or read/write object) being read and written concurrently by multiple clients. An operation O (read or write) has a start time $\tau_{start}(O)$ when the client issues O, and a finish time $\tau_{finish}(O)$ when the client receives an answer (for a read) or an acknowledgment (for a write). We assume that at time 0 (initial time), the key has a default value.

DEFINITION 1. $<$ **t-freshness and t-staleness**$>$: *A read operation R is said to be t-fresh if and only if R returns a value written by any write operation that starts at or after time* $\tau_{fresh}(R,t)$*, which is defined below:*

1. *If there is at least one write starting in the interval* $[\tau_{start}(R)-t, \tau_{start}(R)]$*: then* $\tau_{fresh}(R,t)$ *is the start time of the earliest starting write operation in this interval, and*
2. *If there is no write starting in the interval* $[\tau_{start}(R)-t, \tau_{start}(R)]$*, then there are two cases:*
   (a) *No write starts before R starts: then* $\tau_{fresh}(R,t)=0$.
   (b) *Some write starts before R starts: then* $\tau_{fresh}(R,t)$ *is the start time of the last write operation that starts before* $\tau_{start}(R)-t$.

A read that is not t-fresh is said to be t-stale.

Note that the above characterization of $t_{fresh}(R,t)$ only depends on *start times* of operations.

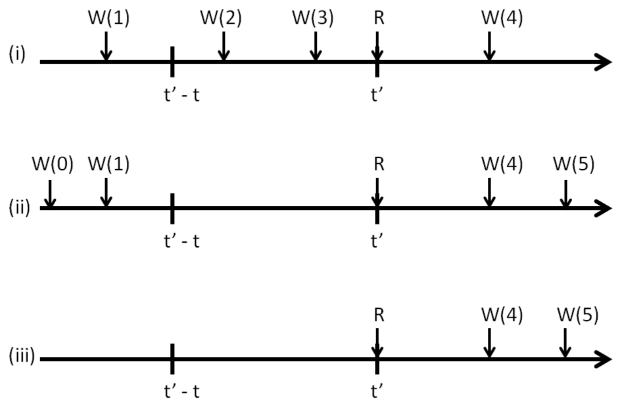

Figure 1: *Examples illustrating Definition 1. Only start times of each operation are shown.*

Fig. 1 shows three examples for t-freshness. The figure shows the times at which several read and write operations are issued (the time when operations complete are not shown in the figure). $W(x)$ in the figure denotes a write operation with a value x. Note that our definition of *t-freshness* allows a read to return a value that is written by a write issued after the read is issued. In Fig. 1(i), $\tau_{fresh}(R,t)=\tau_{start}(W(2))$; therefore, R is t-fresh if it returns $2,3$ or 4. In Fig. 1(ii), $\tau_{fresh}(R,t)=\tau_{start}(W(1))$; therefore, R is t-fresh if it returns $1,4$ or 5. In Fig. 1(iii), $\tau_{fresh}(R,t)=0$; therefore, R is t-fresh if it returns $4,5$ or the default.

DEFINITION 2. <**Probabilistic Consistency**>: *A key-value store satisfies (t_c, p_{ic})-consistency if in* <ins>*any execution*</ins> *of the system, the fraction of read operations satisfying t_c-freshness is at least $(1 - p_{ic})$.*

> Intuitively, p_{ic} is the *likelihood of returning stale data*, given the time-based freshness requirement t_c.

The closest definition was proposed by the t-visibility in Probabilistically Bounded Staleness (PBS) model [15] and Δ-atomicity [31]. These two metrics do not require a read to return the latest write, but provide time bound on the staleness of the data returned by the read. The main difference between t-freshness and these two previous metrics is that we consider the start time of write operations rather than the end time. This allows us to characterize consistency-latency tradeoff precisely. While we prefer t-freshness, yet our PCAP system (Section 3) is modular and could instead use the PBS t-visibility or Δ-atomicity.

We define our probabilistic notion of latency as follows:

DEFINITION 3. <**Probabilistic Latency**>: *A key-value store satisfies (t_a, p_{ua})-latency if in* <ins>*any execution*</ins> *of the system, the fraction of read operations that complete within t_a time units of their start time is at least $(1 - p_{ua})$.*

> Intuitively, given response time requirement t_a, p_{ua} is the *likelihood of a read violating the t_a*.

Finally, we capture the notion of a soft partition of the network by defining a probabilistic partition model. For our theoretical part, we assume that propagation delay follows the same partition model over time, and the parameters do not change over time. (Later, our implementation and experiments in Section 5 will measure the effect of time-varying partition models.)

In a key-value store, data can propagate from one client to another via the other nodes using different approaches. For instance, in Apache Cassandra [4], a write might go from a writer client to a coordinator to a replica or from a replica to another replica in the form of read repair [24]. Our partition model characterizes the delay of all propagation approaches.

DEFINITION 4. <**Probabilistic Partition**>: *An execution is said to suffer (t_p, α)-partition if $\geq \alpha$ fractions of writes suffer propagation delay $> t_p$.*

Intuitively, α is the likelihood that delay over all the propagation paths from a writer client to a reader client will be over the delay deadline t_p. We do not assume eventual delivery of message. Messages delayed too long are lost.

2.2 Impossibility Results

We now present two theorems that characterize the consistency-latency tradeoff. **Readers may skip the proofs without loss of continuity**.

First, we consider the case when the client expectations are stringent, i.e., the client expects all data to be fresh within a time bound, and all reads need to be answered within a time bound.

THEOREM 1. *If $t_c + t_a < t_p$, then it is impossible to implement a read/write data object under a $(t_p, 0)$-partition while achieving $(t_c, 0)$*-consistency*, and $(t_a, 0)$*-latency*, i.e., there* <ins>*exists an execution*</ins> *such that these three properties cannot be satisfied simultaneously.*

Proof: The proof is by contradiction. In a system that satisfies all three properties in all executions, consider an execution with only two clients, a writer client and a reader client. There are two operations: (i) the writer client issues a write W, and (ii) the reader client issues a read R at time $\tau_{start}(R) = \tau_{start}(W) + t_c$. Due to $(t_c, 0)$-consistency, the read R must return the value from W.

Let the delay of the write request W be exactly t_p units of time (this obeys $(t_p, 0)$-partition). Thus, the earliest time that W's value can arrive at the reader client is $(\tau_{start}(W) + t_p)$. However, to satisfy $(t_a, 0)$-latency, the reader client must receive an answer by time $\tau_{start}(R) + t_a = \tau_{start}(W) + t_c + t_a$. However, this time is earlier than $(\tau_{start}(W) + t_p)$ because $t_c + t_a < t_p$. Hence, the value returned by W cannot satisfy $(t_c, 0)$-consistency. This is a contradiction. $\square$

Essentially, the above theorem relates the clients' expectations of freshness (t_c) and latency (t_a) to the propagation delays (t_p). If client expectations are too stringent when the maximum propagation delay is large, then it may not be possible to guarantee both consistency and latency expectations.

However, if we allow a fraction of the reads to return late (i.e., after t_a), or return t_c-stale values (i.e., when either p_{ic} or p_{ua} is non-zero), then it may be possible to satisfy the three properties together even if $t_c + t_a < t_p$. Hence, we consider non-zero p_{ic}, p_{ua} and α in our second theorem.

THEOREM 2. *If $t_c + t_a < t_p$, and $p_{ua} + p_{ic} < \alpha$, then it is impossible to implement a read/write data object under a (t_p, α)-partition while achieving (t_c, p_{ic})*-consistency*, and (t_a, p_{ua})*-latency*, i.e., there* <ins>*exists an execution*</ins> *such that these three properties cannot be satisfied simultaneously.*

Proof: The proof is by contradiction. In a system that satisfies all three properties in all executions, consider an execution with only two clients, a writer client and a reader client. The execution contains alternating pairs of write and read operations $W_1, R_1, W_2, R_2, \ldots, W_n, R_n$, such that:

- Write W_i starts at time $(t_c + t_a) \cdot (i - 1)$,
- Read R_i starts at time $(t_c + t_a) \cdot (i - 1) + t_c$, and
- Each write W_i writes a distinct value v_i.

By our definition of (t_p, α)-partition, there are at least $n \cdot \alpha$ written values v_j's that have propagation delay $> t_p$. By a similar argument as in the proof of Theorem 1, each of their corresponding reads R_j are such that R_j cannot both satisfy t_c-freshness and also return within t_a. That is, R_j is either t_c-stale or returns later than t_a after its start time. There are $n \cdot \alpha$ such reads R_j – let us call these "bad" reads.

By definition, the set of reads S that are t_c-stale, and the set of reads A that return after t_a are such that $|S| \leq n \cdot p_{ic}$ and $|A| \leq n \cdot p_{ua}$. Put together, these imply:

$n \cdot \alpha \leq |S \cup A| \leq |S| + |A| \leq n \cdot p_{ic} + n \cdot p_{ua}$.

The first inequality arises because all the "bad" reads are in $S \cup A$. But this inequality implies that $\alpha \leq p_{ua} + p_{ic}$, which violates our assumptions. $\square$

3. PCAP Key-value Stores

We now convert our probabilistic models for consistency and latency from Section 2 into SLAs, and show how to design adaptive key-value stores that satisfy such probabilistic SLAs. We call such systems as PCAP systems.

Assumptions about underlying Key-value Store: First, we assume each key is replicated on multiple servers. Second, we assume the existence of a "coordinator" server that acts as a client proxy in the system, forwards client queries to replicas, and finally relays replica responses to clients – most key-value stores feature such a coordinator [3, 4]. Third, we assume the existence of a background mechanism (e. g. , read repair [24]) for reconciling divergent replicas. Finally, we assume that the system is synchronized so that we can use timestamps to detect stale data (most key-value stores already require this assumption, e. g. , to decide which updates are fresher).

SLAs: We consider two scenarios, where the application specifies either: i) a probabilistic latency requirement in the SLA, or ii) a probabilistic consistency requirement in the SLA. In the former case our adaptive system optimizes the probabilistic consistency while meeting the SLA requirement, while in the latter case it optimizes probabilistic latency while meeting the SLA. These SLAs are probabilistic in the sense that they give statistical guarantees to operations over a long time duration.

Our latency SLA (i) looks as follows[3]:

Given: Latency $SLA = < p_{ua}^{sla}, t_a^{sla}, t_c^{sla} >$;
Ensure that: Fraction p_{ua} of reads, whose finish and start times differ by more than t_a^{sla}, is such that: p_{ua} stays below p_{ua}^{sla} ;
Minimize: Fraction p_{ic} of reads which do not satisfy t_c^{sla}-freshness.

As an example, consider a shopping cart application (from [44]) that desires that at most 10% of the operations may take longer than 300 ms, but wishes to minimize staleness. In our system this requirement can be specified as the following PCAP latency SLA:

$< p_{ua}^{sla}, t_a^{sla}, t_c^{sla} > = < 0.1, 300\ ms, 0\ ms >$.

Our consistency SLA (ii) looks as follows:

Given: Consistency $SLA = < p_{ic}^{sla}, t_a^{sla}, t_c^{sla} >$;
Ensure that: Fraction p_{ic} of reads, which do not satisfy t_c^{sla}-freshness, is such that: p_{ic} stays below p_{ic}^{sla} ;
Minimize: Fraction p_{ua} of reads whose finish and start times differ by more than t_a^{sla}.

As an example, consider a web search application that desires no more than 10% of the search results return data that is over 500 ms old, but wishes to minimize the fraction of operations taking longer than 100 ms [44]. This requirement can be specified as the following PCAP consistency SLA:

$< p_{ic}^{sla}, t_a^{sla}, t_c^{sla} > = < 0.10, 500\ ms, 100\ ms >$.

Our PCAP system leverages three control knobs to meet these SLAs: 1) *read delay*, 2) *read repair rate*, and 3) *consistency level*. The last two of these are present in most key-value stores. The first (read delay) has been discussed in literature [5, 15, 26, 32]. To the best of our knowledge, we are the first to experimentally explore the impact of using read delay to satisfy SLAs for arbitrary key-value storage systems.

Increased Knob	Latency	Consistency
Read Delay	Degrades	Improves
Read Repair Rate	Unaffected	Improves
Consistency Level	Degrades	Improves

Figure 2: *Effect of Various Control Knobs.*

3.1 Control Knobs

Table 2 shows the effect of our three control knobs on latency and consistency. We discuss each of these knobs and explain the entries in the table.

The knobs of Table 2 are all directly or indirectly applicable to the read path in the key-value store. The placement of these knobs in the Cassandra query path is shown in Fig. 3. We show the four major steps involved in answering a read query from a front-end to the key-value store cluster: (1) Client sends a read query for a key to a coordinator server in the key-value store cluster; (2) Coordinator forwards the query to one or more replicas holding the key; (3) Response is sent from replica(s) to coordinator; (4) Coordinator forwards response with highest timestamp to front-end; (5) Coordinator does *read repair* by updating replicas, which had returned older values, by sending them the freshest timestamp value for the key. Step (5) is done in parallel with (4).

A *read delay* involves the coordinator artificially delaying the read query for a specified duration of time before forwarding it to the replicas. i.e., between step (1) and step

[3] This SLA is similar to latency SLAs used in industry today.

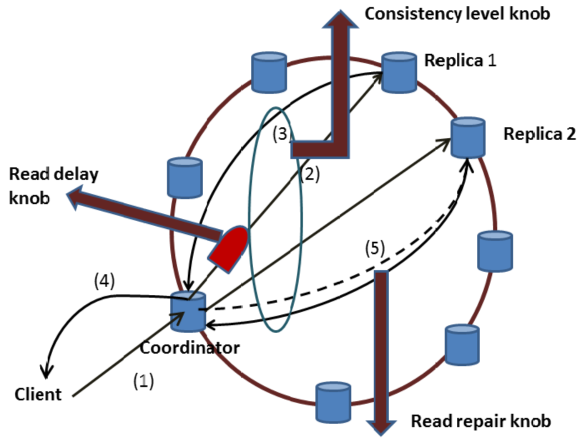

Figure 3: *Cassandra Read Path and PCAP Control Knobs.*

(2). This gives the system some time to converge after previous writes. Increasing the value of read delay improves consistency (lowers p_{ic}) and lowers latency (increases p_{ua}). Decreasing read delay achieves the reverse. Read delay is an attractive knob because: 1) it is non-intrusive to client expectations such as consistency level, and 2) it is fine-grained in time, and can be set in milliseconds. Our PCAP system inserts read delays only when needed to satisfy the specified SLA.

However, read delay has the disadvantage that it cannot be negative, as one cannot speed up a query and send it back in time. This brings us to our second knob: read repair rate. Read repair was depicted as distinct step (5) in our outline of Fig. 3, and is typically performed in the background. The coordinator maintains a buffer of recent reads where some of the replicas returned older values along with the associated freshest value. It periodically picks an element from this buffer and updates the appropriate replicas. In key-value stores like Apache Cassandra and Riak, read repair rate is available as a configuration parameter per column-family.

Our read repair rate knob is the probability with which a given read that returned stale replica values will be added to the read repair buffer. Thus, a read repair rate of 0 implies no read repair, and replicas will be updated only by subsequent writes. Read repair rate = 0.1 means the coordinator performs read repair for 10% of the read requests.

Increasing (respectively decreasing) the read repair rate can improve (respectively degrade) the consistency. Since the read repair rate does not directly affect the read path (Steps (4) and (5) described earlier, are parallel), it does not affect latency – Table 2 summarizes this behavior.[4]

The third potential control knob is consistency level. Key-value stores allow the client to specify, along with each read or write operation, how many replicas the coordinator should wait for (in step (3) of Fig. 3) before it sends the reply back in step (4). For instance, some consistency levels offered in Cassandra are: `ONE`, `TWO`, `QUORUM`, `ALL`. These are along a spectrum: as one increases consistency level from `ONE` to `ALL`, reads are delayed longer (latency decreases) while the chance of returning the latest write rises (consistency increases).

Our PCAP system primarily relies on read delay and repair rate as the control knobs. Consistency level as a control knob in PCAP can be used only for applications in which reads do not specify any consistency level at all. That is, if a read specifies a higher consistency level, it would be prohibitive for the PCAP system to degrade the consistency level as this may violate client expectations. Further, read delay and repair rate are *non-blocking* control knobs under replica failure, whereas consistency level is a *blocking* knob. For example, if a Cassandra client sets consistency level to `QUORUM` with replication factor 3, then the coordinator will be blocked if two of the key's replicas are on failed nodes. On the other hand, under replica failures read repair rate does not affect operation latency, while read delay only delays reads by a maximum amount.

3.2 Selecting A Control Knob

For the primary control knob, we prefer read delay over read repair rate. This is because the former allows us to influence both consistency and latency, while the latter affects only consistency. The only exception occurs when during our adaptation process, we reach a state where we need to degrade consistency (e.g., increase p_{ic} to be closer to the SLA) but the read delay value is already zero. Since read delay cannot be lowered further, in this instance we switch to using the secondary knob of read repair rate, and start decreasing this instead.

Read repair rate is not a good choice for the primary knob as it takes longer to obtain an estimate of p_{ic} for read repair rate compared to read delay. Because read repair rate is a probability, we need a larger number of samples (in the operation log) to accurately estimate the actual p_{ic} resulting from a given read repair rate. For example in our experiments, we observed that we need to inject $k \geq 3000$ operations to obtain an accurate estimate of p_{ic}, whereas only $k = 100$ suffices for the read delay knob.

3.3 PCAP Control Loop

Our PCAP system uses the control loop depicted in Fig. 4, for the consistency SLA case. The control loop for a latency SLA is analogous and not shown.

The control loop runs at a standalone server called the PCAP Coordinator.[5] This server runs an infinite loop of iterations. In each iteration, the coordinator: i) injects k operations into the store (line 6), ii) collects the log $\mathscr{L}$ for the k re-

[4] Although read repair rate does not affect latency, it introduces some background traffic and can impact propagation delay. Our model ignores such small impacts, however our experiments reflect the net effect of this.

[5] The PCAP Coordinator is a special server, and is different from Cassandra's use of a coordinator for clients to send reads and writes.

```
1:  procedure CONTROL($\mathscr{SLA} = < p_{ic}^{sla}, t_c^{sla}, t_a^{sla} >, \varepsilon$)
2:      $p_{ic}^{sla'} = p_{ic}^{sla} - \varepsilon$;
3:      Select control_knob; // (Sections 3.1, 3.2)
4:      $inc = 1$;
5:      $dir = +1$;
6:      while (true) do
7:          Inject $k$ new operations (reads and writes)
8:            into store;
9:          Collect log $\mathscr{L}$ of recent completed reads
10:           and writes (values, start and finish times);
11:         Use $\mathscr{L}$ to calculate
12:           $p_{ic}$ and $p_{ua}$; // (Section 3.4)
13:         $new\_dir = (p_{ic} > p_{ic}^{sla'})? + 1 : -1$;
14:         if $new\_dir == dir$ then
15:             $inc = inc * 2$; // Multiplicative increase
16:             if $inc > MAX\_INC$ then
17:                 $inc = MAX\_INC$:
18:             end if
19:         else
20:             $inc = 1$; // Reset to unit step
21:             $dir = new\_dir$; // Change direction
22:         end if
23:         $control\_knob = control\_knob + inc * dir$;
24:     end while
25: end procedure
```

Figure 4: *Adaptive Control Loop for Consistency SLA.*

cent operations in the system (line 8), iii) calculates p_{ua}, p_{ic} (Section 3.4) from $\mathscr{L}$ (line 10), and iv) uses these to change the knob (lines 12-22).

The behavior of the control loop in Fig. 4 is such that the system will converge to "around" the specified SLA line. Because our original latency and consistency SLAs require the system to stay below the line, we use a *laxity* parameter ε, subtract ε from the target SLA, and treat this as the target SLA in the control loop. Concretely, given a target consistency SLA $< p_{ic}^{sla}, t_a^{sla}, t_c^{sla} >$, where the goal is to control the fraction of stale reads to be under p_{ic}^{sla}, we control the system such that p_{ic} quickly converges around $p_{ic}^{sla'} = p_{ic}^{sla} - \varepsilon$, and thus stay below p_{ic}^{sla}. Small values of ε suffice to guarantee convergence (for instance, our experiments use $\varepsilon <= 0.05$).

Continuously changing the control knob needs to be done in such a way that it makes the system quickly converge. We found that the naive approach of changing the control knob by the smallest unit increment (e.g., always 1 ms changes in read delay) resulted in a long convergence time. Thus, we opted for a *multiplicative* approach (Fig. 4, lines 12-22) to ensure quick convergence.

We explain the control loop via an example. For concreteness, suppose only the read delay knob (Section 3.1) is being changed, and that the system has a consistency SLA. Suppose p_{ic} is higher than $p_{ic}^{sla'}$. The multiplicative-change strategy starts incrementing the read delay, initially starting with a unit step size (line 3). This step size is exponentially *increased* from one iteration to the next, thus multiplicatively increasing read delay (line 14). This continues until the measured p_{ic} just goes under $p_{ic}^{sla'}$. At this point, the *new_dir* variable changes sign (line 12), so the strategy reverses direction, and the step size is reset to unit size (lines 19-20). In subsequent iterations, the read delay starts being *decreased* by the step size. Again the step size is increased exponentially until p_{ic} just goes above $p_{ic}^{sla'}$. Then its direction is reversed again, and this process continues similarly thereafter. Notice that (lines 12-14) from one iteration to the next, as long as p_{ic} continues to remain above (or below) $p_{ic}^{sla'}$, we have that: i) the direction of movement does not change, and ii) exponential increase continues. At steady state, the control loop keeps changing direction with a unit step size, and the metric stays converged under the SLA.

In order to prevent large step sizes, we cap the maximum step size (line 15-17). For our experiments with read delay we cap at 10 ms, with unit step size 1 ms.

We preferred active measurement (wherein the PCAP Coordinator injects queries to calculate p_{ua}, p_{ic}) rather than passive measurement due to two reasons: i) the active approach gives the PCAP Coordinator better control on the convergence, thus the convergence rate is more uniform over time, and ii) in the passive approach if the client operation rate were to become low, then either the PCAP Coordinator would need to inject more queries, or convergence would slow down. Nevertheless, in Section 5.3.6, we show results using a passive measurement approach.

Overall our PCAP controller satisfies SASO (Stability, Accuracy, low Settling time, small Overshoot) control objectives [33].

3.4 Complexity of Computing p_{ua} and p_{ic}

We show that the computation of p_{ua} and p_{ic} (line 10, Fig. 4) is efficient. Suppose there are r reads and w writes in the log, thus log size $k = r + w$. Calculating p_{ua} requires a linear pass of the read operations, comparing the difference of their finish and start times to t_a. This takes $O(r) = O(k)$.

p_{ic} is calculated as follows. We first extract and sort all the writes according to start timestamp, inserting each write into a hash table under the key of <object value, write key, write timestamp>. We make a second pass wherein, for each read operation, we extract its matching write by using the hash table key (the third entry of the hash key is the same as the read's returned value timestamp). We also extract neighboring writes of this matching write in constant time (due to the sorting), and thus calculate t_c-freshness for each read. The first pass takes time $O(r + w + w \log w)$, while the second pass takes $O(r + w)$. The total time complexity to calculate p_{ic} is thus $O(r + w + w \log w) = O(k \log k)$.

4. Implementation Details

In this section, we discuss how support for our consistency and latency SLAs can be easily incorporated into the Cassandra and Riak key-value stores via minimal changes.

4.1 PCAP Coordinator

From Section 3.3, recall that the PCAP Coordinator runs an infinite loop that continuously injects operations, collects logs ($k = 100$ operations by default), calculates metrics, and changes the control knob. We implemented a modular PCAP Coordinator using Python (around 100 LOC), which can be connected to any key-value store.

We integrated PCAP into two popular NoSQL stores: Apache Cassandra [4] and Riak [3] – each of these required changes to about 50 lines of original store code.

4.2 Apache Cassandra

First, we modified the Cassandra v 1.2.4 to add read delay and read repair rate as control knobs. We changed the Cassandra Thrift interface so that it accepts read delay as an additional parameter. Incorporating the read delay into the read path required around 50 lines of Java code.

Read repair rate is specified as a column family configuration parameter, and thus did not require any code changes. We used YCSB's Cassandra connector as the client, modified appropriately to talk with the clients and the PCAP Coordinator.

4.3 Riak

We modified Riak v 1.4.2 to add read delay and read repair as control knobs. Due to the unavailability of a YCSB Riak connector, we wrote a separate YCSB client for Riak from scratch (250 lines of Java code).

We introduced a new system-wide parameter for read delay, which was passed via the Riak http interface to the Riak coordinator which in turn applied it to all queries that it receives from clients. This required about 50 lines of Erlang code in Riak. Like Cassandra, Riak also has built-in support for controlling read repair rate.

5. Experiments

Our experiments are in two stages: microbenchmarks (Section 5.2) and deployment experiments (Section 5.3).

5.1 Experiment Setup

Our PCAP Cassandra system and our PCAP Riak system were each run with their default settings. We used YCSB v 0.1.4 [10] to send operations to the store.

Each YCSB experiment consisted of a load phase, followed by a work phase. Unless otherwise specified, we used the following YCSB parameters: 16 threads per YCSB instance, 2048 B values, and a read-heavy distribution (80% reads). We had as many YCSB instances as the cluster size, one co-located at each server. The default key size was 10 B for Cassandra. Both YCSB-Cassandra and YCSB-Riak connectors were used with the weakest quorum settings and 3 replicas per key. The default throughput was 1000 ops/s. All operations use a consistency level of `ONE`.

Both PCAP systems were run in a cluster of 9 d710 Emulab servers [6], each with 4 core Xeon processors, 12 GB RAM, and 500 GB disks. The default network topology was a LAN (star topology), with 100 Mbps bandwidth and inter-server delay of 20 ms, dynamically controlled using traffic shaping.

We used NTP [7] to synchronize clocks within 1 ms – this is reasonable since we are limited to a single datacenter. This clock skew can be made tighter by using atomic or GPS clocks [21]. This synchronization is needed by the PCAP coordinator to calculate SLA metrics on the logs.

5.2 Microbenchmark Experiments

5.2.1 Impact of Control Knobs on Consistency

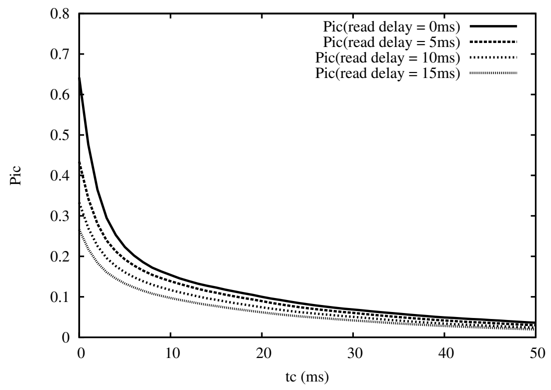

Figure 5: *Effectiveness of Read Delay knob in PCAP Cassandra. Read repair rate fixed at 0.1.*

We study the impact of two control knobs on consistency: read delay and read repair rate.

Fig. 5 shows the inconsistency metric p_{ic} against t_c for different read delays. This shows that when applications desire fresher data (left half of the plot), read delay provides a good knob to control inconsistency p_{ic}. When the freshness requirements are lax (right half of plot), the knob is less useful–however, p_{ic} is already low in this region.

On the other hand, read repair rate had a relatively smaller effect on the system. We found that a change in read repair rate from 0.1 to 1 altered p_{ic} by only 15%, while Fig. 5 showed that a 15 ms increase in read delay (at $t_c = 0$ *ms*) lowered inconsistency by over 50%. As mentioned earlier, using read repair rate required calculating p_{ic} over logs of at least $k = 3000$ operations, while read delay worked well with $k = 100$. Henceforth, by default we use read delay as our sole control knob.

5.2.2 PCAP vs. PBS

Fig. 6 compares, for a 50%-write workload, the probability of inconsistency against t for both existing work PBS (t-visibility) [15] and PCAP (t-freshness) described in Section 2.1. We observe that PBS's reported inconsistency is lower because it does not consider overlapping reads and writes, while our PCAP metric accounts for these and thus is more reflective of reality. Nevertheless, our PCAP system can be made to run by using PBS t-visibility metric instead of our consistency metric.

5.2.3 PCAP Calculation Time

Fig. 7 shows the total time for the PCAP Coordinator to calculate p_{ic} and p_{ua} metrics for values of k from 100 to 10K, and using multiple threads. We observe low computation times of around 1.5 s, except when there are 64 threads and a 10K-sized log: under this situation, the system starts to degrade as too many threads contend for relatively little memory resources. Henceforth, the PCAP Coordinator by default uses a log size of $k = 100$ operations and 16 threads.

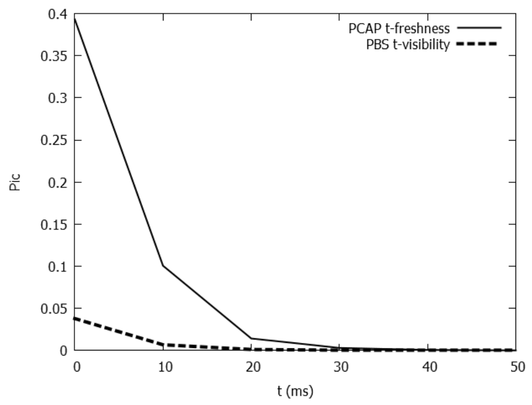

Figure 6: *p_{ic} PCAP vs. PBS consistency metrics. Read repair rate set to 0.1, 50% writes.*

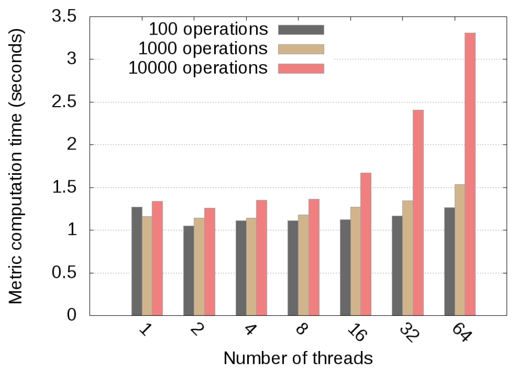

Figure 7: *PCAP Coordinator time taken to both collect logs and compute p_{ic} and p_{ua} in PCAP Cassandra.*

5.2.4 Effect of Read Repair Rate Knob

All of our deployment experiments have solely used read delay as the control knob for the algorithm of Fig. 4. Fig. 8 shows a portion of a run when only read repair rate was used by our PCAP Cassandra system. This was because read

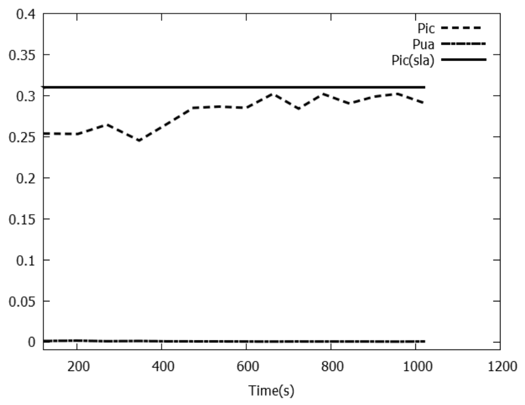

Figure 8: *Effect of Read Repair Rate on PCAP Cassandra. $p_{ic} = 0.31$, $t_c = 0$ ms, $t_a = 100$ ms.*

delay was already zero, and we needed to push p_{ic} up to p_{ic}^{sla}. First we notice that p_{ua} does not change with read repair rate, as expected (Table. 2). Second we notice that the convergence of p_{ic} is very slow – it changes from 0.25 to 0.3 over a long period of 1000 s.

Due to this slow convergence, we conclude that read repair rate is useful only in a system when network delays remain relatively stable. Under continuously changing network conditions (e.g., a lognormal distribution) convergence may be slower and thus one should instead use read delay as the sole control knob.

5.3 Deployment Experiments

We now subject our two PCAP systems to real network workloads and YCSB query workloads. In particular, we present two types of experiments: 1) *sharp network jump* experiments, where the network delay at some of the servers changes suddenly, and 2) *lognormal* experiments, which inject continuously-changing and realistic delays into the network. Our experiments use $\varepsilon <= 0.05$ (Section 3.3).

Fig. 9 summarizes the various of SLA parameters and network conditions used in our experiments.

5.3.1 Latency SLA under Sharp Network Jump

Fig. 10 shows the timeline of a scenario for PCAP Cassandra using the following latency SLA: $p_{ua}^{sla} = 0.2375$, $t_c = 0$ ms, $t_a = 150$ ms.

In the initial segment of this run ($t = 0$ s to $t = 800$ s) the network delays are small; the node to LAN delay is 10 ms. After the warm up phase, by $t = 400$ s, Fig. 10 shows that p_{ua} has converged to the target SLA. Inconsistency p_{ic} stays close to zero.

We wish to measure how close the PCAP system is to the optimal-achievable envelope (Section 2). We do this by first calculating α for our specific network, then calculating the optimal achievable non-SLA metric, and finally seeing how close our non-SLA metric is to this optimal.

System	SLA	Parameters	Delay Model	Plot
Riak	Latency	$p_{ua} = 0.2375, t_a = 150ms, t_c = 0\ ms$	Sharp delay jump	Fig. 13
Riak	Consistency	$p_{ic} = 0.17, t_c = 0\ ms, t_a = 150\ ms$	Lognormal	Fig. 17
Cassandra	Latency	$p_{ua} = 0.2375, t_a = 150\ ms, t_c = 0\ ms$	Sharp delay jump	Figs. 10, 11, 12
Cassandra	Consistency	$p_{ic} = 0.15, t_c = 0\ ms, t_a = 150\ ms$	Sharp delay jump	Fig. 14
Cassandra	Consistency	$p_{ic} = 0.135, t_c = 0\ ms, t_a = 200\ ms$	Lognormal	Figs. 15, 16, 18, 19
Cassandra	Consistency	$p_{ic} = 0.2, t_c = 0\ ms, t_a = 200\ ms$	Lognormal	Fig. 20

Figure 9: *Deployment Experiments: Summary of Settings and Parameters.*

First, from Theorem 1 we know that the achievability region requires $t_c + t_a \geq t_p$ – hence, we set $t_p = t_c + t_a$. Based on this, and the probability distribution of delays in our chosen network, we analytically calculate the exact value of α as the fraction of client pairs whose propagation delay exceeds t_p (see Definition 4).

Given this value of α at time instant t, we can calculate the optimal value of p_{ic} as $p_{ic}(opt) = \max(0, \alpha - p_{ua})$. Fig. 10 shows that in the initial part of the plot (until $t =$ 800 s), the value of α is close to 0, and the p_{ic} achieved by PCAP Cassandra is close to optimal.

At time $t = 800$ s in Fig. 10, we sharply increase the LAN delay for 5 out of 9 servers from 10 ms to 26 ms. This sharp network jump results in a lossier network, as shown by the value of α going up from 0 to 0.42. As a result, the value of p_{ua} initially spikes – however, the PCAP system adapts, and by time $t = 1200$ s the value of p_{ua} has converged back to under the SLA.

However, the elevated value of $\alpha(= 0.42)$ implies that the optimal-achievable $p_{ic}(opt)$ is also higher after $t = 800\ s$. Once again we notice that p_{ic} converges in the second segment of Fig. 10 by $t = 1200$ s.

To visualize how close the PCAP system is to the optimal-achievable envelope Fig. 11 shows, for Fig. 10, the two achievable envelopes as piecewise linear segments (named "before jump" and "after jump") and the (p_{ua}, p_{ic}) data points from our run in Fig. 10. The figure annotates the clusters of data points by their time interval. We observe that in the stable states both before the jump (dark circles) and after the jump (empty triangles) are close to their optimal-achievable envelopes.

Fig. 12 shows the CDF plot for p_{ua} and p_{ic} in the steady state time interval [400 s, 800 s] of Fig. 10, corresponding to the bottom left cluster from Fig. 11. We observe that p_{ua} is always below the SLA.

Fig. 13 shows a scatter plot for our PCAP Riak system under a latency SLA ($p_{ua}^{sla} = 0.2375, t_a = 150\ ms, t_c = 0\ ms$). The sharp network jump occurs at time $t = 4300$ s when we increase the node to LAN delay for 4 out of the 9 Riak nodes from 10 ms to 26 ms. It takes about 1200 s for p_{ua} to converge to the SLA (at around $t = 1400$ s in the warmup segment and $t = 5500$ s in the second segment).

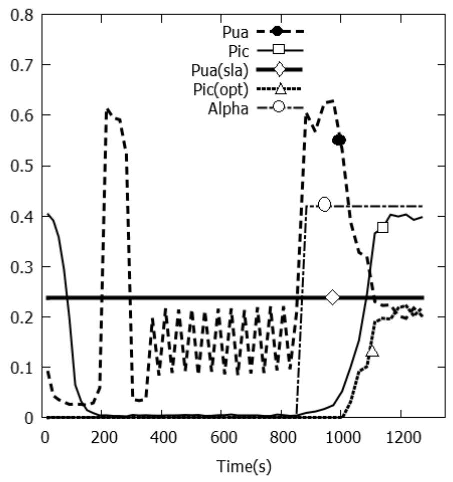

Figure 10: *Latency SLA with PCAP Cassandra under Sharp Network Jump at 800 s: Timeline.*

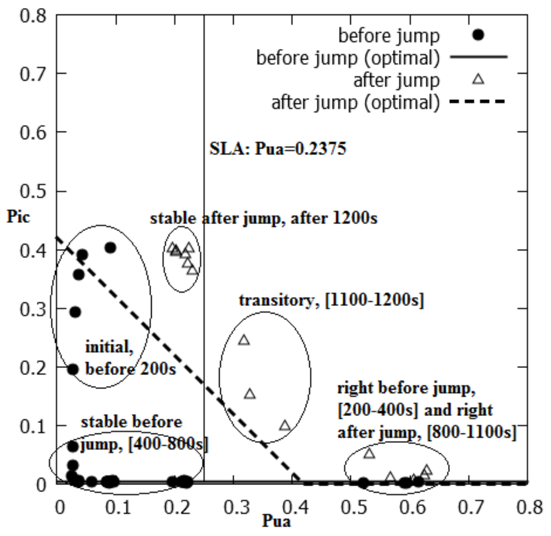

Figure 11: *Latency SLA with PCAP Cassandra under Sharp Network Jump: Consistency-Latency Scatter plot.*

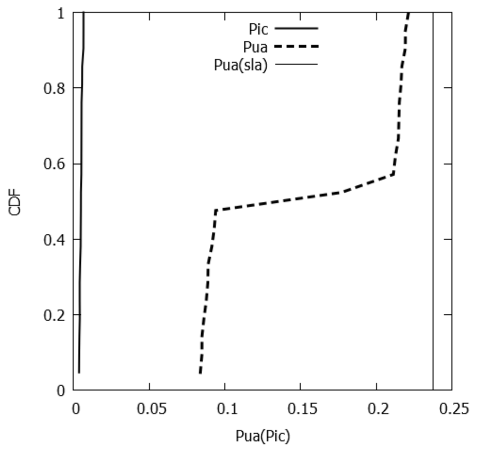

Figure 12: *Latency SLA with PCAP Cassandra under Sharp Network Jump: Steady State CDF [400 s, 800 s].*

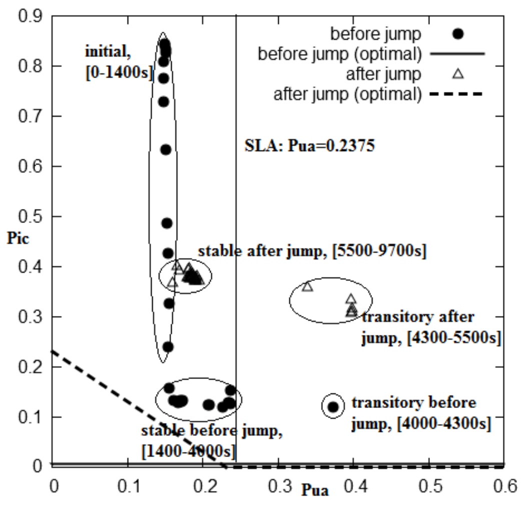

Figure 13: *Latency SLA with PCAP Riak under Sharp Network Jump: Consistency-Latency Scatter plot.*

5.3.2 Consistency SLA under Sharp Network Jump

We present consistency SLA results for PCAP Cassandra (PCAP Riak results were similar and are omitted). We used $p_{ic}^{sla} = 0.15$, $t_c = 0$ *ms*, $t_a = 150$ *ms*. The initial node to LAN delay is 10 ms. At time 750 s, we increase the node to LAN delay for 5 out of 9 nodes to 14 ms. This changes α from 0 to 0.42.

Fig. 14 shows the scatter plot. First we observe that the PCAP system meets the consistency SLA requirements, both before and after the jump. Second, as network conditions worsen, the optimal-achievable envelope moves significantly. Yet the PCAP system remains close to the optimal-achievable envelope. The convergence time is about 100 s, both before and after the jump.

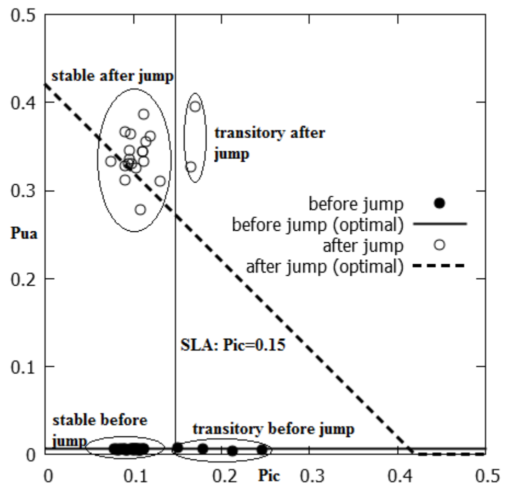

Figure 14: *Consistency SLA with PCAP Cassandra under Sharp Network Jump: Consistency-Latency Scatter plot.*

5.3.3 Experiments with Realistic Delay Distributions

This section evaluates the behavior of PCAP Cassandra and PCAP Riak under continuously-changing network conditions and a consistency SLA (latency SLA experiments yielded similar results and are omitted).

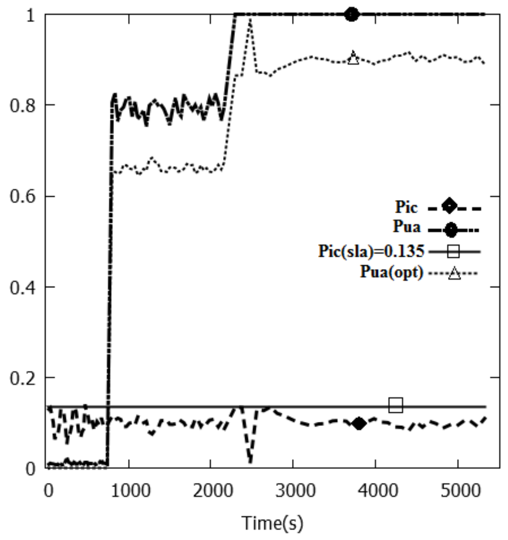

Figure 15: *Consistency SLA with PCAP Cassandra under Lognormal delay distribution: Timeline.*

Based on studies for enterprise datacenters [16] we use a lognormal distribution for injecting packet delays into the network. We modified the Linux traffic shaper to add lognormally distributed delays to each packet. Fig. 15 shows a timeline where initially ($t = 0$ to 800 s) the delays are lognormally distributed, with the underlying normal distributions of $\mu = 3$ ms and $\sigma = 0.3$ ms. At $t = 800$ s we increase μ and σ to 4 ms and 0.4 ms respectively. Finally at around 2100 s, μ and σ become 5 ms and 0.5 ms respectively. Fig. 16 shows

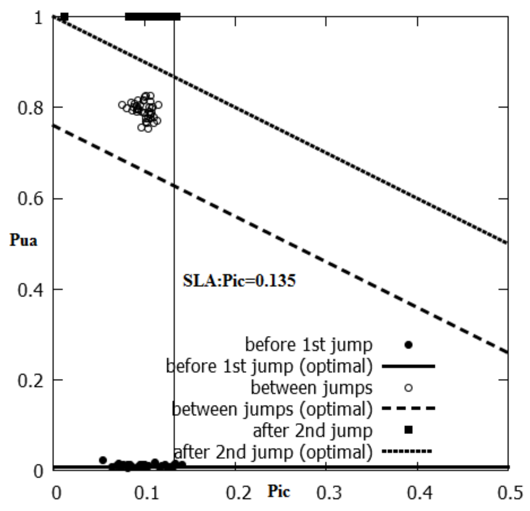

Figure 16: *Consistency SLA with PCAP Cassandra under Lognormal delay distribution: Consistency-Latency Scatter plot.*

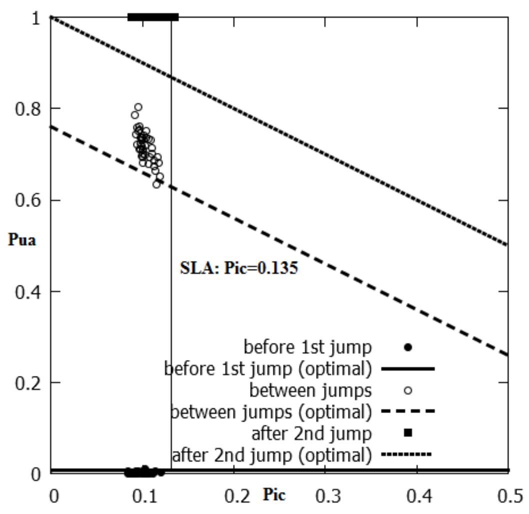

Figure 18: *Scatter plot for same settings as Fig. 16, but with 32 servers and 16K ops/s.*

the corresponding scatter plot. We observe that in all three time segments, the inconsistency metric p_{ic}: i) stays below the SLA, and ii) upon a sudden network change converges back to the SLA. Additionally, we observe that p_{ua} converges close to its optimal achievable value.

Fig. 17 shows the effect of worsening network conditions on PCAP Riak. At around $t = 1300$ s we increase μ from 1 ms to 4 ms, and σ from 0.1 ms to 0.5 ms. The plot shows that it takes PCAP Riak an additional 1300 s to have inconsistency p_{ic} converge to the SLA. Further the non-SLA metric p_{ua} converges close to the optimal.

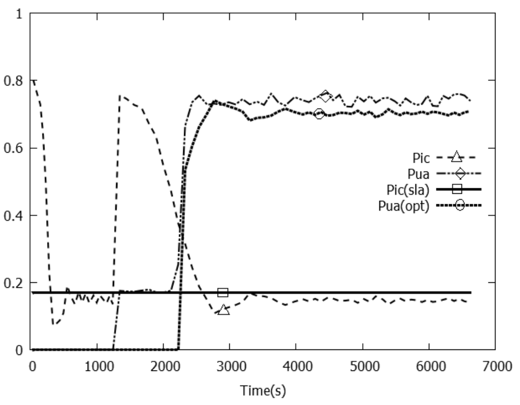

Figure 17: *Consistency SLA with PCAP Riak under Lognormal delay distribution: Timeline.*

5.3.4 Scalability

We measure scalability via an increased workload on PCAP Cassandra. Compared to Fig. 16, in this new run we increased the number of servers from 9 to 32, and throughput to 16000 ops/s, and ensured that each server stores at least some keys. All other settings are unchanged compared to Fig. 16. The result is shown Fig. 18. Compared with Fig. 16, we observe an improvement with scale – in particular, after the first delay increase ("between jumps" envelope), a larger scale brings the system closer to optimal.

5.3.5 Effect of Timeliness Requirement

The timeliness requirements in an SLA directly affect how close the PCAP system is to the optimal-achievable envelope. Fig. 19 shows the effect of varying timeliness parameter t_a on a consistency SLA ($t_c = 0$ *ms*, $p_{ic} = 0.135$) for PCAP Cassandra with 10 ms node to LAN delays. For each t_a, we consider the cluster of the (p_{ua}, p_{ic}) points achieved by the PCAP system in its stable state, calculate its centroid, and measure (and plot on vertical axis) the distance d from this centroid to the optimal-achievable consistency-latency envelope. Note that the optimal envelope calculation also involves t_a, since α depends on it (Section 5.3.1).

Fig. 19 shows that when t_a is too stringent ($<$ 100 ms), the PCAP system may be far from the optimal envelope even when it satisfies the SLA. In the case of Fig. 19, this is because in our network, the average time to cross four hops (client to coordinator to replica, and the reverse) is $20 \times 4 = 80$ ms. As t_a starts to go beyond this (e.g., $t_a \geq$ 100 ms), PCAP is essentially optimal.

5.3.6 Passive Measurement Approach

So far all our experiments have used the active measurement approach. In this section, we repeat a PCAP Cassandra consistency SLA experiment ($p_{ic} = 0.2$, $t_c = 0$ *ms*) using a passive measurement approach.

In Figure 20, instead of actively injecting operations, we sample ongoing client operations. We estimate p_{ic} and

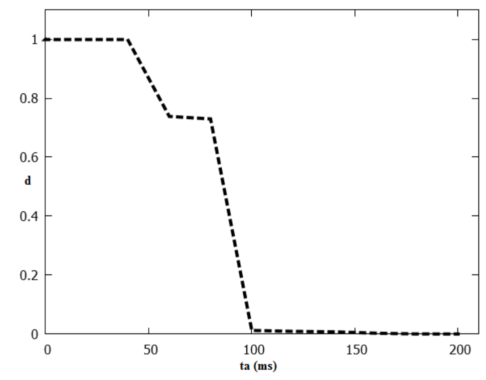

Figure 19: *Effect of Timeliness Requirement (t_a) on PCAP Cassandra. Consistency SLA with $p_{ic} = 0.135$, $t_c = 0$ ms.*

p_{ua} from the 100 latest operations from 5 servers selected randomly.

At the beginning, the delay is lognormally distributed with $\mu = 1$ *ms*, $\sigma = 0.1$ *ms*. The passive approach initially converges to the SLA. We change the delay ($\mu = 2$ *ms*, $\sigma = 0.2$ *ms*) at $t = 325$ *s*. We observe that, compared to the active approach, 1) consistency (SLA metric) oscillates more, and 2) the availability (non-SLA metric) is farther from optimal and takes longer to converge. For the passive approach, SLA convergence and non-SLA optimization depends heavily on the sampling of operations used to estimate the metrics. Thus we conclude that it is harder to satisfy SLA and optimize the non-SLA metric with the passive approach.

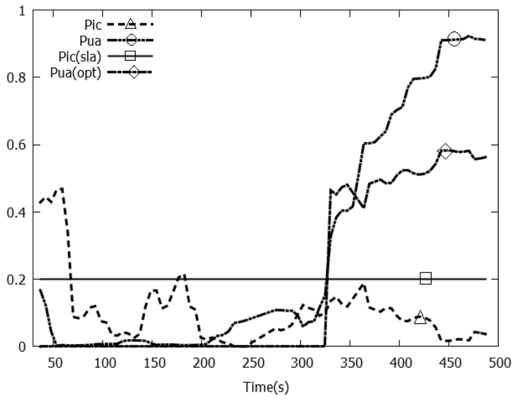

Figure 20: *Consistency SLA with PCAP Cassandra under Lognormal delay distribution: Timeline (Passive).*

6. Related Work

Consistency-Latency Tradeoffs: In addition to the Weak CAP Principle [27] and PACELC [11] discussed in Section 1, there has been work on theoretically characterizing the tradeoff between latency and strong consistency models. Attiya and Welch studied the tradeoff between latency and linearizability and sequential consistency [13]. Subsequent work has explored linearizablity under different delay models [25, 41]. These papers are concerned with strong consistency models whereas we consider *t*-freshness, which is a form of eventual consistency. Moreover, their delay models are different from our partition model.

There are two systems that are closest to our work. First, PBS [15] proposed a probabilistic consistency model for quorum-based stores, but did not focus on latency, soft partitions or the CAP theorem. Second, the Pileus system [44] considered families of consistency/latency SLAs, but for geo-distributed settings. Compared to that, our PCAP SLAs are applicable in a single data-center setting. Pileus requires the application to specify a utility value with each SLA – in comparison, PCAP considers probabilistic metrics of p_{ic}, p_{ua} (which in turn may be derivable from Pileus' utilities).

Adaptive Systems: There are a few existing systems that controls consistency in storage systems. FRACS [47] controls consistency by allowing replicas to buffer updates up to a given staleness. AQuA [35] continuously moves replicas between "strong" and "weak" consistency groups to implement different consistency levels. TACT [46] controls staleness by limiting the number of outstanding writes at replicas (order error) and bounding write propagation delay (staleness). All the mentioned systems provide *best-effort* behavior for consistency, within the latency bounds. In comparison, the PCAP system explicitly allows applications to specify SLAs. Consistency levels have been adaptively changed to deal with node failures and network changes in [22], however this may be intrusive for applications that explicitly set consistency levels for operations. Artificially delaying read operations at servers (similar to our read delay knob) has been used to eliminate staleness spikes (improve consistency) which are correlated with garbage collection in a specific key-value store (Apache Cassandra) [26]. We show that read delay can be used to meet SLAs for arbitrary key-value stores.

For stream processing, [28] proposes a control algorithm to find optimal scaling to meet throughput SLA. There has been work on adaptive elasticity control for storage [37], and adaptively tuning Hadoop clusters to meet SLAs [34]. Compared to the controllers present in these systems, our PCAP controller acheives control objectives [33] using a different set of techniques to meet SLAs for key-value stores.

7. Summary

In this paper, we have first extended the CAP theorem into a probabilistic variant which took into account probabilistic models for consistency, latency, and soft partitions within a datacenter. We then incorporated these consistency and latency SLAs into Cassandra and Riak. Our experiments with YCSB workloads and realistic traffic demonstrated that our PCAP system meets the SLAs, that its performance is close to the optimal-achievable consistency-availability envelope, and that it scales well.

Acknowledgements We are grateful to Doug Terry for his insightful comments on earlier drafts of this manuscript.

References

[1] Activo: “Why Low Latency Matters?”. http://goo.gl/2XQ8Ul.

[2] Amazon: milliseconds means money. http://goo.gl/fs9pZb.

[3] Basho Riak. http://basho.com/riak/.

[4] Cassandra. http://cassandra.apache.org/.

[5] Consistency in Amazon S3. http://goo.gl/yhAoJy.

[6] Emulab. https://www.emulab.net/.

[7] Emulab NTP. http://goo.gl/SMk2uZ.

[8] Project Voldemort. http://goo.gl/9uhLoU.

[9] Real-time ad impression bids using DynamoDB. http://goo.gl/C7gdpc.

[10] Yahoo! Cloud Serving Benchmark (YCSB). http://goo.gl/GiA5c.

[11] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[12] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[13] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[14] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, 2013.

[15] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endowment*, 5(8):776–787, 2012.

[16] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of datacenters in the wild. In *Proc. ACM SIGCOMM IMC*, pages 267–280, 2010.

[17] E. Brewer. A certain freedom: Thoughts on the CAP theorem. In *Proc. ACM PODC*, pages 335–335, 2010.

[18] E. A. Brewer. Towards robust distributed systems (Invited Talk). In *Proc. ACM PODC*, 2000.

[19] A. Cockcroft. Dystopia as a service (Invited Talk). In *Proc. ACM SoCC*, 2013.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. ACM SoCC*, pages 143–154, 2010.

[21] J. C. Corbett et al. Spanner: Google’s globally-distributed database. In *Proc. USENIX OSDI*, pages 251–264, 2012.

[22] A. Davidson, A. Rubinstein, A. Todi, P. Bailis, and S. Venkataraman. Adaptive hybrid quorums in practical settings, 2013. http://goo.gl/LbRSW3.

[23] J. Dean. Design, Lessons and Advice from Building Large Distributed Systems, 2009.

[24] 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 SOSP*, pages 205–220, 2007.

[25] M. Eleftheriou and M. Mavronicolas. Linearizability in the presence of drifting clocks and under different delay assumptions. In *Distributed Computing*, volume 1693, pages 327–341. 1999.

[26] H. Fan, A. Ramaraju, M. McKenzie, W. Golab, and B. Wong. Understanding the causes of consistency anomalies in apache cassandra. *Proc. VLDB Endowment*, 8(7):810–813, 2015.

[27] A. Fox and E. A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proc. HotOS*, pages 174–178, 1999.

[28] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2014.

[29] S. Gilbert and N. A. Lynch. Perspectives on the CAP theorem. *IEEE Computer*, 45(2):30–36, 2012.

[30] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *Proc. ACM SOSP*, pages 15–28, 2011.

[31] W. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and profit. In *Proc. ACM PODC*, pages 197–206, 2011.

[32] W. Golab and J. J. Wylie. Providing a measure representing an instantaneous data consistency level, Jan. 2014. US Patent Application 20,140,032,504.

[33] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[34] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proc. ACM SoCC*, pages 18:1–18:14, 2011.

[35] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An adaptive quality of service aware middleware for replicated services. *IEEE Transactions on Parallel and Distributed Systems*, 14:1112–1125, 2003.

[36] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. USENIX OSDI*, pages 265–278, 2012.

[37] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *Proc. IEEE ICAC*, pages 1–10, 2010.

[38] 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 *Proc. ACM SOSP*, pages 401–416, 2011.

[39] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. USENIX NSDI*, pages 313–328, 2013.

[40] N. Lynch and S. Gilbert. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[41] M. Mavronicolas and D. Roth. Linearizable read/write objects. *Theoretical Computer Science*, 220(1):267 – 319, 1999.

[42] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proc. SSS*, pages 386–400, 2011.

[43] D. Terry. Replicated data consistency explained through baseball. *CACM*, 56(12):82–89, 2013.

[44] 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 SOSP*, pages 309–324, 2013.

[45] W. Vogels. Eventually consistent. *ACM CACM*, pages 40–44, 2009.

[46] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM TOCS*, pages 239–282, 2002.

[47] C. Zhang and Z. Zhang. Trading replication consistency for performance and availability: an adaptive approach. In *Proc. ICDCS*, pages 687–695, 2003.