Characterizing and Adapting the Consistency-Latency Tradeoff in Distributed Key-Value Stores

MUNTASIR RAIHAN RAHMAN, LEWIS TSENG, SON NGUYEN, INDRANIL GUPTA, and NITIN VAIDYA, University of Illinois at Urbana-Champaign

The CAP theorem is a fundamental result that applies to distributed storage systems. In this article, we first present and prove two CAP-like impossibility theorems. To state these theorems, we present probabilistic models to characterize the three important elements of the CAP theorem: consistency (C), availability or latency (A), and partition tolerance (P). The theorems show the un-achievable envelope, that is, which combinations of the parameters of the three models make them impossible to achieve together. Next, we present the design of a class of systems called Probabilistic CAP (PCAP) that perform close to the envelope described by our theorems. In addition, these systems allow applications running on a single data center to specify either a latency Service Level Agreement (SLA) or a consistency SLA. The PCAP systems automatically adapt, in real time and under changing network conditions, to meet the SLA while optimizing the other C/A metric. We incorporate PCAP into two popular key-value stores: Apache Cassandra and Riak. Our experiments with these two deployments, under realistic workloads, reveal that the PCAP systems satisfactorily meets SLAs and perform close to the achievable envelope. We also extend PCAP from a single data center to multiple geo-distributed data centers.

CCS Concepts: • **Information systems → Distributed storage;**

Additional Key Words and Phrases: Consistency, adaptivity, distributed storage

ACM Reference Format:
Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, and Nitin Vaidya. 2017. Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores. ACM Trans. Auton. Adapt. Syst. 11, 4, Article 20 (January 2017), 36 pages.
DOI: http://dx.doi.org/10.1145/2997654

1. INTRODUCTION

Storage systems form the foundational platform for modern Internet services such as Web searches, analytics, and social networking. Ever-increasing user bases and massive datasets have forced users and applications to forgo conventional relational databases and move towards a new class of scalable storage systems known as NoSQL (Not only SQL) key-value stores. Many of these distributed key-value stores (e.g., Cassandra [Lakshman and Malik 2008], Riak [Gross 2009], Dynamo [DeCandia et al. 2007], and Voldemort [LinkedIn 2009]), support a simple Get/Put interface for accessing and updating data items. The data items are replicated at multiple servers for fault

This work was supported in part by the following grants: NSF CNS 1409416, NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, VMware Graduate Fellowship, Yahoo!, and a generous gift from Microsoft.
Authors' addresses: M. R. Rahman, L. Tseng, S. Nguyen, and I. Gupta, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801; emails: {mrahman2, ltseng3, nguyenb1, indy}@illinois.edu; N. Vaidya, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801; email: nhv@illinois.edu.

DOI: http://dx.doi.org/10.1145/2997654

tolerance. In addition, they offer a very weak notion of consistency known as eventual consistency [Vogels 2009; Bailis and Ghodsi 2013], which, roughly speaking, says that if no further updates are sent to a given data item, all replicas will eventually hold the same value.

These key-value stores are preferred by applications for whom eventual consistency suffices but where high availability and low latency (i.e., fast reads and writes [Abadi 2012]) are paramount. Latency is a critical metric for such cloud services because latency is correlated to user satisfaction; for instance, a 500ms increase in latency for operations at Google.com can cause a 20% drop in revenue [Peled 2010]. At Amazon, this translates to a $6M yearly loss per added millisecond of latency [Lang 2009]. This correlation between delay and lost retention is fundamentally human. Humans suffer from a phenomenon called *user cognitive drift*, wherein if more than a second (or so) elapses between clicking on something and receiving a response, the user's mind is already elsewhere.

At the same time, clients in such applications expect freshness, that is, 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 [Cockcroft 2013], and freshness of data translates to accurate tracking and user satisfaction. This implies that clients care about a *time-based* notion of data freshness. Thus, this article focuses on consistency based on the notion of data freshness (as defined later).

The CAP theorem was proposed by Eric Brewer [2000, 2010] and later formally proved by Gilbert and Lynch [2012, 2002]. It essentially states that a system can choose at most two of three desirable properties: consistency (C), availability (A), and partition tolerance (P). Recently, Abadi [2012] 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 the tradeoff between consistency and availability/latency, while we provide a *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, for example, in geo-distributed data centers. However, individual data centers themselves suffer far more frequently from "soft" partitions [Dean 2009], arising from periods of elevated message delays or loss rates (i.e., the "otherwise" part of PACELC) within a data center. Neither the original CAP theorem nor the existing work on consistency in key-value stores [Bailis et al. 2014; DeCandia et al. 2007; Glendenning et al. 2011; Wei et al. 2015; Li et al. 2012; Lloyd et al. 2011, 2013; Shapiro et al. 2011; Terry et al. 2013; Vogels 2009] address such soft partitions for a single data center.

In this article we state and prove two CAP-like impossibility theorems. To state these theorems, we present probabilistic[1] models to characterize the three important elements: soft partition, latency requirements, and consistency requirements. 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 Service Level Agreement (SLA). 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 the un-achievable envelope, that is, which combinations of the parameters in these three models (partition, latency, consistency) make them impossible to achieve together. Note that the focus of the article is neither defining a

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

new consistency model nor comparing different types of consistency models. Instead, we are interested in the un-achievable envelope of the three important elements and measuring how close a system can perform to this envelop.

Next, we describe the design of a class of systems called Probabilistic CAP (PCAP) that perform close to the envelope described by our theorems. In addition, these systems allow applications running inside a single data center 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, while optimizing the consistency achieved. Similarly, given a probabilistic consistency SLA, PCAP meets the consistency requirement 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 [Cockcroft 2013], online advertising [Barr 2013], and shopping cart applications [Terry et al. 2013]; 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 [Terry et al. 2013], which desires search results with bounded staleness but would like to minimize the response time. While the PCAP system can be used with a variety of consistency and latency models (like Probabilistically Bounded Staleness (PBS) [Bailis et al. 2014]), we use our PCAP models for concreteness.

We have integrated our PCAP system into two key-value stores: Apache Cassandra [Lakshman and Malik 2008] and Riak [Gross 2009]. Our experiments with these two deployments, using YCSB (Yahoo! Cloud Serving Benchmark) [Cooper et al. 2010a] benchmarks, reveal that PCAP systems satisfactorily meet a latency SLA (or consistency SLA), optimize the consistency metric (respectively, latency metric), perform reasonably close to the envelope described by our theorems, and scale well.

We also extend PCAP from a single data center to multiple geo-distributed data centers. The key contribution of our second system (which we call GeoPCAP) is a set of rules for composing probabilistic consistency/latency models from across multiple data centers in order to derive the global consistency-latency tradeoff behavior. Realistic simulations demonstrate that GeoPCAP can satisfactorily meet consistency/latency SLAs for applications interacting with multiple data centers, while optimizing the other metric.

2. CONSISTENCY-LATENCY TRADEOFF

We consider a key-value store system which provides a read/write interface over an asynchronous distributed message-passing network. The system consists of clients and servers, in which servers are responsible for replicating the data (or read/write object) and ensuring the specified consistency requirements and clients can invoke a write (or read) operation that stores (or retrieves) some value of the specified key by contacting server(s). We assume each client has a corresponding client proxy at the set of servers, which submits read and write operations on behalf of clients [Lloyd et al. 2011; Shapiro 1986]. Specifically, in the system, data can be propagated from a writer client to multiple servers by a replication mechanism or background mechanism such as read repair [DeCandia et al. 2007], and the data stored at servers can later be read by clients. There may be multiple versions of the data corresponding to the same key, and the exact value to be read by reader clients depends on how the system ensures the consistency requirements. Note that, as addressed earlier, we define consistency based on freshness of the value returned by read operations (defined below). We first present our probabilistic models for soft partition, latency, and consistency. Then we present our impossibility results. These results only hold for a single data center. In Section 4 we deal with the multiple-data-center case.

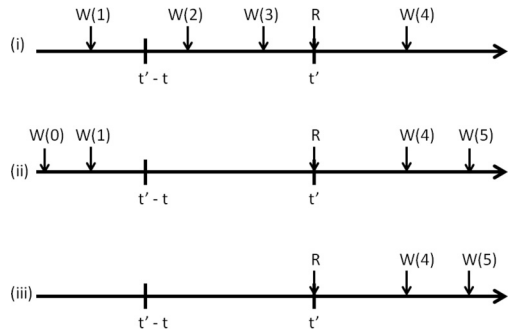

Fig. 1. Examples illustrating Definition 2.1. Only start times of each operation are shown.

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). The write operation ends when the client receives an acknowledgment from the server. The value of a write operation can be reflected on the server side (i.e., visible to other clients) any time after the write starts. For clarity of our presentation, we assume that all write operations end in this article, which is reasonable given client retries. Note that the written value can still propagate to other servers after the write ends by the background mechanism. We assume that at time 0 (initial time), the key has a default value.

Definition 2.1 (*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 $\tau_{fresh}(R, t) = \tau_{start}(R) - t$.
(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 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 Figure 1(i), $\tau_{fresh}(R, t) = \tau_{start}(R) - t = t' - t$; therefore, R is t-fresh if it returns 2, 3, or 4. In Figure 1(ii), $\tau_{fresh}(R, t) = \tau_{start}(W(1))$; therefore, R is t-fresh if it returns 1, 4, or 5. In Figure 1(iii), $\tau_{fresh}(R, t) = 0$; therefore, R is t-fresh if it returns 4, 5, or the default.

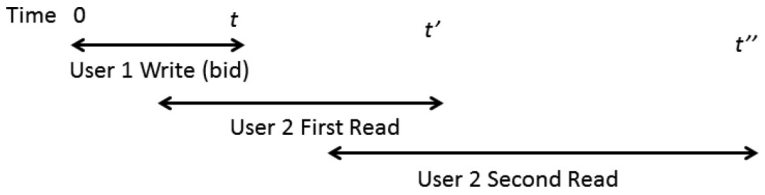

Fig. 2. Example motivating use of Definition 2.2.

Definition 2.2 (*Probabilistic Consistency*). A key-value store satisfies (t_c, p_{ic})-consistency[2] if, in *any execution* 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.

Two similar definitions have been proposed previously: (1) t-visibility from the PBS work [Bailis et al. 2014] and (2) Δ-atomicity [Golab et al. 2011]. These two metrics do not require a read to return the latest write but provide a time bound on the staleness of the data returned by the read. The main difference between t-freshness and these is that we consider the start time of write operations rather than the end time. This allows us to characterize the consistency-latency tradeoff more precisely. While we prefer t-freshness, our PCAP system (Section 3) is modular and could use instead t-visibility or Δ-atomicity for estimating data freshness.

As noted earlier, our focus is neither to compare different consistency models nor to achieve linearizability. We are interested in the un-achievable envelope of soft partition, latency requirements, and consistency requirements. Traditional consistency models like linearizability can be achieved by delaying the effect of a write. On the contrary, the achievability of t-freshness closely ties to the latency of read operations and underlying network behavior, as discussed later. In other words, t-freshness by itself is not a complete definition.

2.1.1. Use Case for t-Freshness. Consider a bidding application (e.g., eBay), where everyone can post a bid, and we want every other participant to see posted bids as fast as possible. Assume that User 1 submits a bid, which is implemented as a write request (Figure 2). User 2 requests to read the bid before the bid write process finishes. The same User 2 then waits a finite amount of time after the bid write completes and submits another read request. Both of these read operations must reflect User 1's bid, whereas t-visibility only reflects the write in User 2's second read (with suitable choice of t). The bid write request duration can include time to send back an acknowledgment to the client, even after the bid has committed (on the servers). A client may not want to wait that long to see a submitted bid. This is especially true when the auction is near the end.

We define our probabilistic notion of latency as follows:

Definition 2.3 (*t-latency*). A read operation R is said to satisfy *t-latency* if and only if it completes within t time units of its start time.

[2]The subscripts c and ic stand for consistency and inconsistency, respectively.

Definition 2.4 (*Probabilistic Latency*). A key-value store satisfies (t_a, p_{ua})-latency[3] if in *any execution* of the system, the fraction of t_a-latency read operations 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 concept of a *soft partition* of the network by defining a probabilistic partition model. In this section, we assume that the partition model for the network does 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 servers using different approaches. For instance, in Apache Cassandra [Lakshman and Malik 2008], a write might go from a writer client to a coordinator server to a replica server or from a replica server to another replica server in the form of read repair [DeCandia et al. 2007]. Our partition model captures the delay of all such propagation approaches. Please note that the partition model applies not to the network directly but to the paths taken by the read or write operation queries themselves. This means that the network as a whole may be good (or bad), but if the paths taken by the queries are bad (or good), only the latter matters.

Definition 2.5 (*Probabilistic Partition*). An execution is said to suffer (t_p, α)-partition if the fraction f of paths from one client to another client, via a server, which have latency higher than t_p, is such that $f \geq \alpha$.

Our delay model loosely describes the message delay caused by any underlying network behavior without relying on the assumptions on the implementation of the key-value store. We do not assume eventual delivery of messages. We neither define propagation delay for each message nor specify the propagation paths (or alternatively, the replication mechanisms). This is because we want to have general lower bounds that apply to all systems that satisfy our models.

2.2. Impossibility Results

We now present two theorems that characterize the consistency-latency tradeoff in terms of our probabilistic models.

First, we consider the case when the client has tight expectations, that is, the client expects all data to be fresh within a time bound, and all reads need to be answered within a time bound.

THEOREM 2.6. *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*, that is, there exists an execution 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

[3]The subscripts a and ua stand for availability and unavailability, respectively.

($\tau_{start}(W) + t_p$). 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. □

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 2.7. *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*, that is, there exists an execution 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 2.6, 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. □

3. PCAP KEY-VALUE STORES

Having formally specified the (un-)achievable envelope of consistency-latency (Theorem 2.7), we now move our attention to designing systems that achieve performance close to this theoretical envelope. We also 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 inside a single data center. We call such systems PCAP systems. So PCAP systems (1) can achieve performance close to the theoretical consistency-latency tradeoff envelope and (2) can adapt to meet probabilistic consistency and latency SLAs inside a single data center. Our PCAP systems can also alternatively be used with SLAs from PBS [Bailis et al. 2014] or Pileus [Terry et al. 2013; Ardekani and Terry 2014].

Assumptions about Underlying Key-Value Store. PCAP systems can be built on top of existing key-value stores. We make a few assumptions about such key-value stores. First, we assume that 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, finds the replica locations for a key (e.g., using consistent hashing [Stoica et al. 2001]), forwards client queries to replicas, and, finally, relays replica responses to clients. Most key-value stores feature such a coordinator [Lakshman and Malik 2008; Gross 2009]. Third, we assume the existence of a background mechanism such as read repair [DeCandia et al. 2007] for reconciling divergent replicas. Finally, we assume that the clocks on each server in the system are synchronized using a protocol like NTP so we can use global timestamps to detect stale data (most key-value stores running within a data center already require this assumption, for example, to decide which updates are fresher). It should be noted that our impossibility results in Section 2 do not depend on the accuracy of the clock synchronization protocol. However, the sensitivity of the protocol affects the ability of PCAP systems to adapt to network delays. For example, if the servers are synchronized to within 1ms using NTP, then the PCAP system cannot react to network delays lower than 1ms.

SLAs. We consider two scenarios, where the SLA specifies either (i) a probabilistic latency requirement or (ii) a probabilistic consistency requirement. In the former case, our adaptive system optimizes the probabilistic consistency while meeting the SLA requirement, whereas in the latter 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 duration.

A latency SLA (i) looks as follows:

Given: Latency $SLA =< p_{ua}^{sla}, t_a^{sla}, t_c^{sla} >$;
Ensure that: The 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: The fraction p_{ic} of reads which do not satisfy t_c^{sla}-freshness.

This SLA is similar to latency SLAs used in the industry today. As an example, consider a shopping cart application [Terry et al. 2013] where the client requires that at most 10% of the operations take longer than 300ms but wishes to minimize staleness. Such an application prefers latency over consistency. In our system, this requirement can be specified as the following PCAP latency SLA:

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

A consistency SLA looks as follows:

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

Note that, as mentioned earlier, consistency is defined based on freshness of the value returned by read operations. As an example, consider a web search application that wants to ensure no more than 10% of search results return data that is over 500ms old but wishes to minimize the fraction of operations taking longer than 100ms [Terry et al. 2013]. Such an application prefers consistency over latency. This requirement can be specified as the following PCAP consistency SLA:

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

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

Fig. 3. Effect of various control knobs.

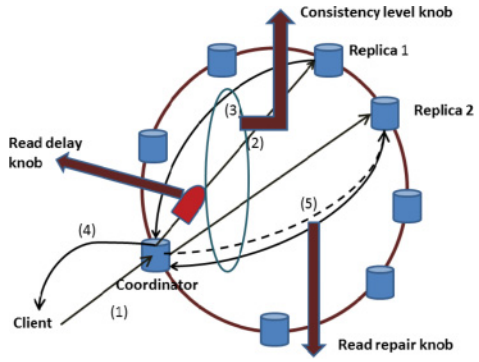

Fig. 4. Cassandra read path and PCAP control knobs.

Our PCAP system can leverage 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 previous literature [Swidler 2009; Bailis et al. 2014; Fan et al. 2015; Golab and Wylie 2014; Zawirski et al. 2015].

3.1. Control Knobs

Figure 3 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 Figure 3 are all directly or indirectly applicable to the read path in the key-value store. As an example, the knobs pertaining to the Cassandra query path are shown in Figure 4, which shows 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 client; and (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 usually performed in the background.

A *read delay* involves the coordinator artificially delaying the read query for a specified duration of time before forwarding it to the replicas. that is, between step (1) and step (2). This gives the system some time to converge after previous writes. Increasing the value of read delay improves consistency (lowers p_{ic}) and degrades latency (increases p_{ua}). Decreasing read delay achieves the reverse. Read delay is an attractive knob because (1) it does not interfere with client specified parameters (e.g., consistency level in Cassandra [DataStax 2016]) and (2) it can take any non-negative continuous value instead of only discrete values allowed by consistency levels. Our PCAP system inserts read delays only when it is needed to satisfy the specified SLA.

However, read delay 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 a distinct step (5) in our outline of Figure 4 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 an accessible 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) consistency. Since the read repair rate does not directly affect the read path (Step (5), described earlier, is performed in the background), it does not affect latency. Figure 3 summarizes this behavior.[4]

The third potential control knob is consistency level. Some 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 Figure 4) before it sends the reply back in step (4). For instance, Cassandra offers consistency levels `ONE`, `TWO`, `QUORUM`, `ALL`. As one increases consistency level from `ONE` to `ALL`, reads are delayed longer (latency decreases) while the possibility of returning the latest write rises (consistency increases).

Our PCAP system relies primarily on read delay and repair rate as the control knobs. Consistency level can be used as a control knob only for applications in which user expectations will not be violated, for example, when reads do not specify a specific discrete consistency level. 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. Techniques like continuous partial quorums (CPQ) [McKenzie et al. 2015] and adaptive hybrid quorums [Davidson et al. 2013] fall into this category and thus interfere with application/client expectations. Further, read delay and repair rate are *non-blocking* control knobs under replica failure, whereas consistency level is *blocking*. 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

As the primary control knob, the PCAP system prefers read delay over read repair rate. This is because the former allows tuning both consistency and latency, while the latter affects only consistency. The only exception occurs when during the PCAP system adaptation process, a state is reached where consistency needs to be degraded (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, the PCAP system switches to using the secondary knob of read repair rate and starts decreasing this instead.

Another reason why read repair rate is not a good choice for the primary knob is that it takes longer to estimate p_{ic} than for read delay. Because read repair rate is a probability, the system needs a larger number of samples (from the operation log) to accurately estimate the actual p_{ic} resulting from a given read repair rate. For example,

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

```
1:  procedure CONTROL(𝒮ℒ𝒜 =< p_ic^sla, t_c^sla, t_a^sla >, ε)
2:      p_ic^sla' := p_ic^sla − ε;
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 ℒ of recent completed reads
10:           and writes (values, start and finish times);
11:         Use 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
```

Fig. 5. Adaptive control loop for consistency SLA.

in our experiments, we observe that the system needs 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

The PCAP control loop adaptively tunes control knobs to always meet the SLA under continuously changing network conditions. The control loop for consistency SLA is depicted in Figure 5. The control loop for a latency SLA is analogous and is not shown.

This control loop runs at a standalone server called the PCAP Coordinator.[5] This server runs an infinite loop. In each iteration, the coordinator (i) injects k operations into the store (line 6), (ii) collects the log $\mathscr{L}$ for the k recent 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 Figure 5 is such that the system will converge to "around" the specified SLA. Because our original latency (consistency) SLAs require p_{ua} (p_{ic}) to stay below the SLA, we introduce 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

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

around $p_{ic}^{sla'} = p_{ic}^{sla} - \varepsilon$ and thus stays below p_{ic}^{sla}. Small values of ε suffice to guarantee convergence (for instance, our experiments use $\varepsilon \leq 0.05$).

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 (Figure 5, 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 active in the system 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} goes just under $p_{ic}^{sla'}$. At this point, the *new_dir* variable changes sign (line 12), so the strategy reverses direction, and the step is reset to unit size (lines 19 and 20). In subsequent iterations, the read delay starts *decreasing* 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 (bounded oscillation), and the metric stays converged under the SLA. Although advanced techniques such as *time dampening* can further reduce oscillations, we decided to avoid them to minimize control loop tuning overheads. In Section 4, we utilized control-theoretic techniques for the control loop in geo-distributed settings to reduce excessive oscillations.

In order to prevent large step sizes, we cap the maximum step size (line 15–17). For our experiments, we do not allow read delay to exceed 10ms, and the unit step size is set to 1ms.

We preferred active measurement (whereby the PCAP Coordinator injects queries rather than passive due to two reasons: (i) the active approach gives the PCAP Coordinator better control on convergence, and 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 5.4.7, we show results using a passive measurement approach. Exploration of hybrid active-passive approaches based on an operation rate threshold could be an interesting direction.

Overall, our PCAP controller satisfies Stability/Accuracy/low Settling time/small Overshoot (SASO) control objectives [Hellerstein et al. 2004].

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

We show that the computation of p_{ua} and p_{ic} (line 10, Figure 5) is efficient. Suppose there are r reads and w writes in the log, thus log size $k = r + w$. Calculating p_{ua} makes a linear pass over the read operations and compares the difference of their finish and start times with 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 key <object value, write key, write timestamp>. In a second pass over the read operations, 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. PCAP FOR GEO-DISTRIBUTED SETTINGS

In this section, we extend our PCAP system from a single data center to multiple geo-distributed data centers. We call this system GeoPCAP.

4.1. System Model

Assume there are n data centers. Each data center stores multiple replicas for each data item. When a client application submits a query, the query is first forwarded to the data center closest to the client. We call this data center the *local* data center for the client. If the local data center stores a replica of the queried data item, that replica might not have the latest value, since write operations at other data centers could have updated the data item. Thus, in our system model, the local data center contacts one or more of other remote data centers to retrieve (possibly) fresher values for the data item.

4.2. Probabilistic Composition Rules

Each data center is running our PCAP-enabled key-value store. Each such PCAP instance defines per-data-center probabilistic latency and consistency models (Section 2). To obtain the global behavior, we need to compose these probabilistic consistency and latency/availability models across different data centers. This is done by our composition rules.

The composition rules for merging independent latency/consistency models from data centers check whether the SLAs are met by the composed system. Since single-data-center PCAP systems define probabilistic latency and consistency models, our composition rules are also probabilistic in nature. However, in reality, our composition rules do not require all data centers to run PCAP-enabled key-value stores systems. As long as we can measure consistency and latency at each data center, we can estimate the probabilistic models of consistency/latency at each data center and use our composition rules to merge them.

We consider two types of composition rules: (1) `QUICKEST` (Q), where at least one data center (e.g., the local or closest remote data center) satisfies client specified latency or freshness (consistency) guarantees, and (2) `ALL` (A), where all the data centers must satisfy latency or freshness guarantees. These two are, respectively, generalizations of Apache Cassandra multi-data-center deployment [Gilmore 2011] consistency levels: `LOCAL_QUORUM` and `EACH_QUORUM`.

Compared to Section 2, which analyzed the fraction of executions that satisfy a predicate (the proportional approach), in this section, we use a simpler probabilistic approach. This is because, although the proportional approach is more accurate, it is more intractable than the probabilistic model in the geo-distributed case.

Our probabilistic composition rules fall into three categories: (1) composing consistency models, (2) composing latency models, and (3) composing a wide-area-network (WAN) partition model with a data-center (consistency or latency) model. The rules are summarized in Figure 6, and we discuss them next.

4.2.1. Composing Latency Models. Assume there are n data centers storing the replica of a key with latency models $(t_a^1, p_{ua}^1), (t_a^2, p_{ua}^2), \ldots, (t_a^n, p_{ua}^n)$. Let $\mathscr{C}^A$ denote the composed system. Let (p_{ua}^c, t_a^c) denote the latency model of the composed system $\mathscr{C}^A$. This indicates that the fraction of reads in $\mathscr{C}^A$ that complete within t_a^c time units is at least $(1 - p_{ua}^c)$. This is the latency SLA expected by clients. Let $p_{ua}^c(t)$ denote the probability of missing deadline by t time units in the composed model. Let X_j denote the random variable measuring read latency in data center j. Let $E_j(t)$ denote the event that $X_j > t$. By definition, we have that $Pr[E_j(t_a^j)] = p_{ua}^j$ and $Pr[\bar{E}_j(t_a^j)] = 1 - p_{ua}^j$. Let $f_j(t)$ denote the

Consistency/Latency/WAN	Composition	$\forall j,\ t_a^j = t$?	Rule
Latency	QUICKEST	Y	$p_{ua}^c(t) = \Pi_j\ p_{ua}^j, \forall j,\ t_a^j = t$
Latency	QUICKEST	N	$p_{ua}^c(min_j\ t_a^j) \geq \Pi_j\ p_{ua}^j \geq p_{ua}^c(max_j\ t_a^j)$, $min_j\ t_a^j \leq t_a^c \leq max_j\ t_a^j$
Latency	ALL	Y	$p_{ua}^c(t) = 1 - \Pi_j\ (1 - p_{ua}^j), \forall j,\ t_a^j = t$
Latency	ALL	N	$p_{ua}^c(min_j\ t_a^j) \geq 1 - \Pi_j\ (1 - p_{ua}^j) \geq p_{ua}^c(max_j\ t_a^j)$, $min_j\ t_a^j \leq t_a^c \leq max_j\ t_a^j$
Consistency	QUICKEST	Y	$p_{ic}^c(t) = \Pi_j\ p_{ic}^j, \forall j,\ t_c^j = t$
Consistency	QUICKEST	N	$p_{ic}^c(min_j\ t_c^j) \geq \Pi_j\ p_{ic}^j \geq p_{ic}^c(max_j\ t_c^j)$, $min_j\ t_c^j \leq t_c^c \leq max_j\ t_c^j$
Consistency	ALL	Y	$p_{ic}^c(t) = 1 - \Pi_j\ (1 - p_{ic}^j), \forall j,\ t_c^j = t$
Consistency	ALL	N	$p_{ic}^c(min_j\ t_c^j) \geq 1 - \Pi_j\ (1 - p_{ic}^j) \geq p_{ic}^c(max_j\ t_c^j)$, $min_j\ t_c^j \leq t_c^c \leq max_j\ t_c^j$
Consistency-WAN	N. A.	N. A.	$Pr[X + Y \geq t_c + t_p^G] \geq p_{ic} \cdot \alpha^G$
Latency-WAN	N. A.	N. A.	$Pr[X + Y \geq t_a + t_p^G] \geq p_{ua} \cdot \alpha^G$

Fig. 6. GeoPCAP composition rules.

cumulative distribution function (CDF) for X_j. So, by definition, $f_j(t_a^j) = Pr[X_j \leq t_a^j] = 1 - Pr[X_j > t_a^j]$. The following theorem articulates the probabilistic latency composition rules:

THEOREM 4.1. *Let n data centers store the replica for a key with latency models $(t_a^1, p_{ua}^1), (t_a^2, p_{ua}^2), \ldots, (t_a^n, p_{ua}^n)$. Let $\mathscr{C}^A$ denote the composed system with latency model (p_{ua}^c, t_a^c). Then for composition rule* QUICKEST *we have:*

$$p_{ua}^c\left(min_j\ t_a^j\right) \geq \Pi_j\ p_{ua}^j \geq p_{ua}^c\left(max_j\ t_a^j\right), \quad (1)$$
$$and\ min_j\ t_a^j \leq t_a^c \leq max_j\ t_a^j,$$
$$where\ j \in \{1, \ldots, n\}.$$

For composition rule ALL*,*

$$p_{ua}^c\left(min_j\ t_a^j\right) \geq 1 - \Pi_j\ \left(1 - p_{ua}^j\right) \geq p_{ua}^c\left(max_j\ t_a^j\right), \quad (2)$$
$$and\ min_j\ t_a^j \leq t_a^c \leq max_j\ t_a^j,$$
$$where\ j \in \{1, \ldots, n\}.$$

PROOF. We outline the proof for composition rule QUICKEST. In QUICKEST, a latency deadline t is violated in the composed model when all data centers miss the t deadline. This happens with probability $p_{ua}^c(t)$ (by definition). We first prove a simpler Case 1 and then the general version in Case 2.

Case 1: Consider the simple case where all t_a^j values are identical, that is, $\forall j, t_a^j = t_a$: $p_{ua}^c(t_a) = Pr[\cap_i E_i(t_a)] = \cap_i Pr[E_i(t_a)] = \Pi_i p_{ua}^i$ (assuming independence across data centers).

Case 2:

Let

$$t_a^i = min_j\ t_a^j, \quad (3)$$

then

$$\forall j,\ t_a^j \geq t_a^i. \quad (4)$$

Then, by definition of CDF function,

$$\forall j,\ f_j(t_a^i) \leq f_j(t_a^j). \quad (5)$$

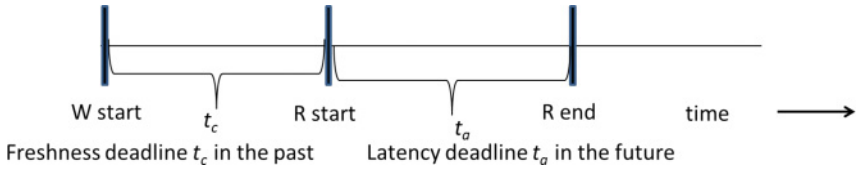

Fig. 7. Symmetry of freshness and latency requirements.

By definition,

$$\forall j,\ \left(Pr\left[X_j \leq t_a^i\right] \leq Pr\left[X_j \leq t_a^j\right]\right), \tag{6}$$

$$\forall j,\ \left(Pr\left[X_j > t_a^i\right] \geq Pr\left[X_j > t_a^j\right]\right). \tag{7}$$

Multiplying all,

$$\Pi_j\ Pr\left[X_j > t_a^i\right] \geq \Pi_j\ Pr\left[X_j > t_a^j\right]. \tag{8}$$

But this means

$$p_{ua}^c(t_a^i) \geq \Pi_j\ p_{ua}^j, \tag{9}$$

$$p_{ua}^c\left(min_j\ t_a^j\right) \geq \Pi_j\ p_{ua}^j. \tag{10}$$

Similarly, let

$$t_a^k = max_j\ t_a^j. \tag{11}$$

Then

$$\forall j,\ t_a^k \geq t_a^j, \tag{12}$$

$$\forall j,\ \left(Pr\left[X_j > t_a^j\right] \geq Pr\left[X_j > t_a^k\right]\right), \tag{13}$$

$$\Pi_j\ Pr\left[X_j > t_a^j\right] \geq \Pi_j\ Pr\left[X_j > t_a^k\right], \tag{14}$$

$$\Pi_j\ p_{ua}^j \geq p_{ua}^c(t_a^k), \tag{15}$$

$$\Pi_j\ p_{ua}^j \geq p_{ua}^c(max_j\ t_a^j). \tag{16}$$

Finally combining Equations (10) and (16), we get Equation (1).

The proof for composition rule ALL follows similarly. In this case, a latency deadline t is satisfied when all data centers satisfy the deadline. So a deadline miss in the composed model means that at least one data center misses the deadline. The derivation of the composition rules are similar and we invite the reader to work them out to arrive at the equations depicted in Figure 6. □

4.2.2. Composing Consistency Models. t-latency (Definition 2.3) and t-freshness (Definitions 2.1) guarantees are time symmetric (Figure 7). While t-lateness can be considered a deadline in the future, t-freshness can be considered a deadline in the past. This means that for a given read, t-freshness constrains how old a read value can be. So the composition rules remain the same for consistency and availability.

Thus the consistency composition rules can be obtained by substituting p_{ua} with p_{ic} and t_a with t_c in the latency composition rules (last four rows in Figure 6).

This leads to the following theorem for consistency composition:

THEOREM 4.2. *Let n data centers store the replica for a key with consistency models* $(t_c^1, p_{ic}^1), (t_c^2, p_{ic}^2), \ldots, (t_c^n, p_{ic}^n)$. *Let* $\mathscr{C}^A$ *denote the composed system with consistency model* (p_{ic}^c, t_c^c). *Then for composition rule* `QUICKEST` *we have:*

$$p_{ic}^c(min_j\ t_c^j) \geq \Pi_j\ p_{ic}^j \geq p_{ic}^c(max_j\ t_c^j), \quad (17)$$
$$and\ min_j\ t_c^j \leq t_c^c \leq max_j\ t_c^j,$$
$$where\ j \in \{1, \ldots, n\}.$$

For composition rule `ALL`,

$$p_{ic}^c(min_j\ t_c^j) \geq 1 - \Pi_j\ (1 - p_{ic}^j) \geq p_{ic}^c(max_j\ t_c^j), \quad (18)$$
$$and\ min_j\ t_c^j \leq t_c^c \leq max_j\ t_c^j,$$
$$where\ j \in \{1, \ldots, n\}.$$

4.2.3. Composing Consistency/Latency Model with a WAN Partition Model. All data centers are connected to each other through a WAN. We assume the WAN follows a partition model (t_p^G, α^G). This indicates that α^G fraction of messages passing through the WAN suffers a delay $> t_p^G$. Note that the WAN partition model is distinct from the per-data-center partition model (Definition 2.5). Let X denote the latency in a remote data center and Y denote the WAN latency of a link connecting the local data center to this remote data center (with latency X). Then the total latency of the path to the remote data center is $X + Y$,[6]

$$Pr[X + Y \geq t_a + t_p^G] \geq \left(Pr[X \geq t_a] \cdot Pr[Y \geq t_p^G]\right) = p_{ua} \cdot \alpha^G. \quad (19)$$

Here we assume that the WAN latency and data-center latency distributions are independent. Note that Equation (19) gives a lower bound of the probability. In practice, we can estimate the probability by sampling both X and Y and estimating the number of times $(X + Y)$ exceeds $(t_a + t_p^G)$.

4.3. Example

The example in Figure 8 shows the composition rules in action. In this example, there is one local data center and two replica data centers. Each data center can hold multiple replicas of a data item. First, we compose each replica data-center latency model with the WAN partition model. Second, we take the WAN-latency-composed models for each data center and compose them using the QUICKEST rule (Figure 6, bottom).

4.4. GeoPCAP Control Knob

We use a similar delay knob to meet the SLAs in a geo-distributed setting. We call this the *geo-delay* knob and denote it as Δ. The time delay Δ is the delay added at the local data center to a read request received from a client before it is forwarded to the replica data centers. Δ affects the consistency-latency tradeoff in a manner similar to the read delay knob in a data center (Section 3.1). Increasing the knob tightens the deadline at each replica data center, thus increasing per-data-center latency (p_{ua}). Similarly to read delay (Figure 3), increasing the geo-delay knob improves consistency, since it gives each data center time to commit latest writes.

[6]We ignore the latency of the local data center in this rule, since the local data-center latency is used in the latency composition rule (Section 4.2.1).

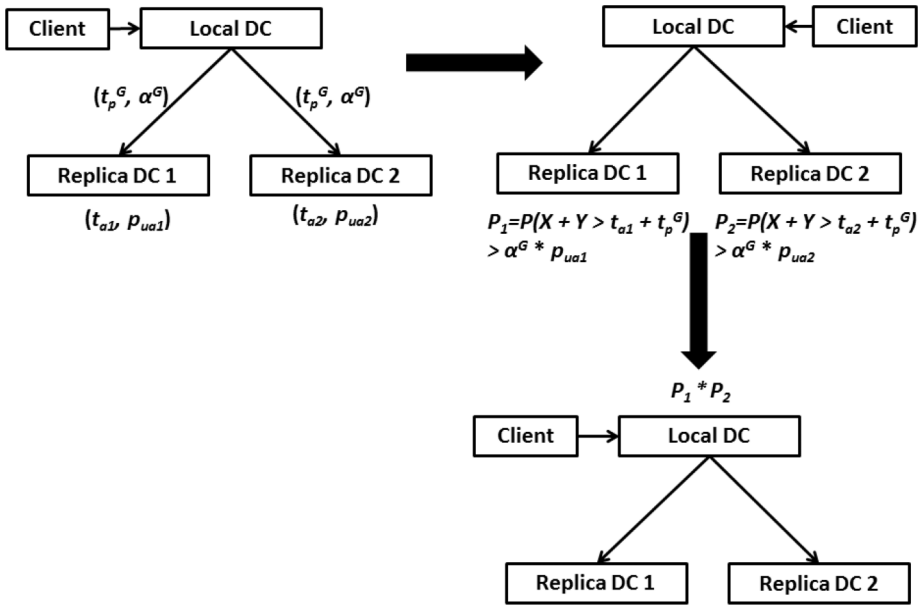

Fig. 8. Example of composition rules in action.

1: **procedure** CONTROL($\mathscr{SLA} =< p_{ic}^{sla}, t_c^{sla}, t_a^{sla} >$)
2: Geo-delay $\Delta := 0$
3: $E := 0$, $Error_{old} := 0$
4: set k_p, k_d, k_i for PID control (tuning)
5: Let (t_p^G, α^G) be the WAN partition model
6: **while** (true) **do**
7: **for** each data-center i **do**
8: Let F_i denote the random freshness interval at i
9: Let L_i denote the random operation latency at i
10: Let W_i denote the WAN latency of the link to i
11: Estimate $p_{ic}^i := Pr[F_i + W_i > t_c^{sla} + t_p^G + \Delta]$ // WAN composition (Section 4.2.3)
12: Estimate $p_{ua}^i := Pr[L_i + W_i > t_a + t_p^G - \Delta = t_a^{sla}]$
13: **end for**
14: $p_{ic}^c := \Pi_i p_{ic}^i$, $p_{ua}^c := \Pi_i p_{ua}^i$ // Consistency/Latency composition (Sections 4.2.1 4.2.2)
15: $Error := p_{ic}^c - p_{ic}^{sla}$
16: $dE := Error - Error_{old}$
17: $E := E + Error$
18: $u := k_p \cdot Error + k_d \cdot dE + k_i \cdot E$
19: $\Delta := \Delta + u$;
20: **end while**
21: **end procedure**

Fig. 9. Adaptive control loop for GeoPCAP consistency SLA (`QUICKEST` Composition).

4.5. GeoPCAP Control Loop

Our GeoPCAP system uses a control loop depicted in Figure 9 for the consistency SLA case using the `QUICKEST` composition rule. The control loops for the other three combinations (Consistency-`QUICKEST`, Latency-`ALL`, and Latency-`QUICKEST`) are similar.

Initially, we opted to use the single-data-center multiplicative control loop (Section 3.3) for GeoPCAP. However, the multiplicative approach led to increased oscillations for the composed consistency (p_{ic}) and latency (p_{ua}) metrics in a geo-distributed setting. The multiplicative approach sufficed for the single-data-center PCAP system, since the oscillations were bounded in a steady state. However, the increased oscillations in a geo-distributed setting prompted us to use a control theoretic approach for GeoCAP.

As a result, we use a proportional-integral-derivative (PID) control theory approach [Astrom and Hagglund 1995] for the GeoPCAP controller. The controller runs an infinite loop, so it can react to network delay changes and meet SLAs. There is a tunable sleep time at the end of each iteration (1s in Section 5.5 simulations). Initially, the geo-delay Δ is set to zero. At each iteration of the loop, we use the composition rules to estimate $p^c_{ic}(t)$, where $t = t^{sla}_c + t^G_p - \Delta$. We also keep track of composed $p^c_{ua}()$ values. We then compute the error, as the difference between current composed p_{ic} and the SLA. Finally, the geo-delay change is computed using the PID control law [Astrom and Hagglund 1995] as follows:

$$u = k_p \cdot Error(t) + k_d \cdot \frac{dError(t)}{dt} + k_i \cdot \int Error(t)dt. \tag{20}$$

Here k_p, k_d, k_i represent the proportional, differential, and integral gain factors for the PID controller, respectively. There is a vast amount of literature on tuning these gain factors for different control systems [Astrom and Hagglund 1995]. Later in our experiments, we discuss how we set these factors to get SLA convergence. Finally, at the end of the iteration, we increase Δ by u. Note that u could be negative if the metric is less than the SLA.

Note that for the single-data-center PCAP system, we used a multiplicative control loop (Section 3.3), which outperformed the unit step size policy. For GeoPCAP, we employ a PID control approach. PID is preferable to the multiplicative approach, since it guarantees fast convergence and can reduce oscillation to arbitrarily small amounts. However, PID's stability depends on proper tuning of the gain factors, which can result in high management overhead. On the other hand, the multiplicative control loop has a single tuning factor (the multiplicative factor), so it is easier to manage. Later, in Section 5.5 we experimentally compare the PID and multiplicative control approaches.

5. EXPERIMENTS

5.1. 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 (in a single data center) via minimal changes.

5.1.1. 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 lines of code), which can be connected to any key-value store.

We integrated PCAP into two popular NoSQL stores: Apache Cassandra [Lakshman and Malik 2008] and Riak [Gross 2009]; each of these required changes to about 50 lines of original store code.

5.1.2. Apache Cassandra. First, we modified the Cassandra v1.2.4 to add read delay and read repair rate as control knobs. We changed the Cassandra Thrift interface so it

would accept 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.

5.1.3. Riak. We modified Riak v1.4.2 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 decided to use YCSB instead of existing Riak clients, since YCSB offers flexible workload choices that model real world key-value store workloads.

We introduced a new systemwide parameter for read delay, which was passed via the Riak http interface to the Riak coordinator, which in turn applies 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.

5.2. Experiment Setup

Our experiments are in three stages: microbenchmarks for a single data center (Section 5.3) and deployment experiments for a single data center (Section 5.4) and a realistic simulation for the geo-distributed setting (Section 5.5). We first discuss the experiments for a single-data-center setting.

Our single-data-center PCAP Cassandra system and our PCAP Riak system were each run with their default settings. We used YCSB v 0.1.4 [Cooper et al. 2010b] to send operations to the store. YCSB generates synthetic workloads for key-value stores and models real-world workload scenarios (e.g., Facebook photo storage workload). It has been used to benchmark many open-source and commercial key-value stores and is the de facto benchmark for key-value stores [Cooper et al. 2010a].

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, 2048B 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 10B for Cassandra, and Riak. Both YCSB-Cassandra and YCSB-Riak connectors were used with the weakest quorum settings and three 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 nine d710 Emulab servers [White et al. 2002], each with four-core Xeon processors, 12GB RAM, and 500GB disks. The default network topology was a local area network (LAN) (star topology), with 100Mbps bandwidth and inter-server round-trip delay of 20ms, dynamically controlled using traffic shaping.

We used NTP to synchronize clocks within 1ms. This is reasonable since we are limited to a single data center. This clock skew can be made tighter by using atomic or GPS (Global Positioning System) clocks [Corbett et al. 2012]. This synchronization is needed by the PCAP coordinator to compute the SLA metrics.

5.3. Microbenchmark Experiments (Single Data Center)

5.3.1. Impact of Control Knobs on Consistency. We study the impact of two control knobs on consistency: read delay and read repair rate.

Figure 10 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 is a flexible 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.

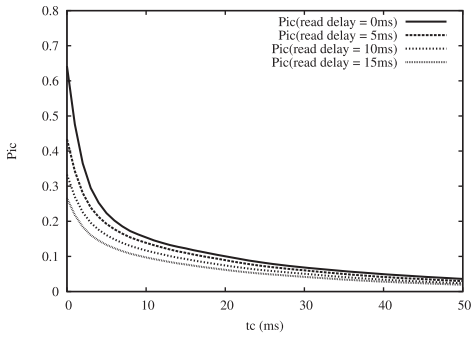

Fig. 10. Effectiveness of read delay knob in PCAP Cassandra. Read repair rate fixed at 0.1.

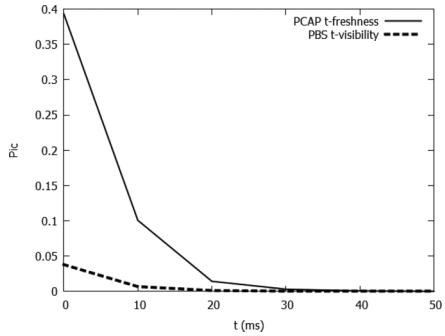

Fig. 11. p_{ic} PCAP vs. PBS consistency metrics. Read repair rate set to 0.1, 50% writes.

On the other hand, read repair rate has a relatively smaller effect. We found that a change in read repair rate from 0.1 to 1 altered p_{ic} by only 15%, whereas Figure 10 showed that a 15ms increase in read delay (at $t_c = 0$ms) lowered inconsistency by over 50%. As mentioned earlier, using read repair rate requires calculating p_{ic} over logs of at least $k = 3,000$ operations, whereas read delay worked well with $k = 100$. Henceforth, by default we use read delay as our sole control knob.

5.3.2. PCAP vs. PBS. To show that our system can work with PBS [Bailis et al. 2014], we integrated t-visibility into PCAP. Figure 11 compares, for a 50%-write workload, the probability of inconsistency against t for both existing work PBS (t-visibility) [Bailis et al. 2014] and PCAP (t-freshness) described in Section 2.1. We observe that PBS's reported inconsistency is lower compared to PCAP. This is because PBS considers a read that returns the value of an in-flight write (overlapping read and write) to be always fresh by default. However, the comparison between PBS and PCAP metrics is not completely fair, since the PBS metric is defined in terms of write operation end times, whereas our PCAP metric is based on write start times. It should be noted that the purpose of this experiment is not to show which metric captures client-centric consistency better. Rather, our goal is to demonstrate that our PCAP system can be made to run by using PBS t-visibility metric instead of PCAP t-freshness.

5.3.3. PCAP Metric Computation Time. Figure 12 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 1.5s, 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 few memory resources. Henceforth, the PCAP Coordinator by default uses a log size of $k = 100$ operations and 16 threads.

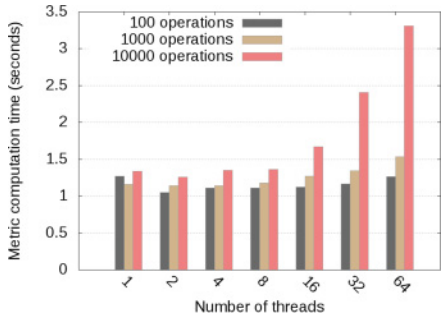

Fig. 12. PCAP Coordinator time taken to both collect logs and compute p_{ic} and p_{ua} in PCAP Cassandra.

System	SLA	Parameters	Delay Model	Plot
Riak	Latency	$p_{ua} = 0.2375, t_a = 150\ ms, t_c = 0\ ms$	Sharp delay jump	Fig. 17
Riak	Consistency	$p_{ic} = 0.17, t_c = 0\ ms, t_a = 150\ ms$	Lognormal	Fig. 21
Cassandra	Latency	$p_{ua} = 0.2375, t_a = 150\ ms, t_c = 0\ ms$	Sharp delay jump	Figs. 14, 15, 16
Cassandra	Consistency	$p_{ic} = 0.15, t_c = 0\ ms, t_a = 150\ ms$	Sharp delay jump	Fig. 18
Cassandra	Consistency	$p_{ic} = 0.135, t_c = 0\ ms, t_a = 200\ ms$	Lognormal	Figs. 19, 20, 25, 26
Cassandra	Consistency	$p_{ic} = 0.2, t_c = 0\ ms, t_a = 200\ ms$	Lognormal	Fig. 27
Cassandra	Consistency	$p_{ic} = 0.125, t_c = 0\ ms, t_a = 25\ ms$	Lognormal	Figs. 22, 23
Cassandra	Consistency	$p_{ic} = 0.12, t_c = 3\ ms, t_a = 200\ ms$	Lognormal	Figs. 28, 29
Cassandra	Consistency	$p_{ic} = 0.12, t_c = 3\ ms, t_a = 200\ ms$	Lognormal	Figs. 30, 31

Fig. 13. Deployment experiments: Summary of settings and parameters.

5.4. Deployment Experiments

We now subject our two PCAP systems to network delay variations 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 $\epsilon \leq 0.05$ (Section 3.3).

Figure 13 summarizes the various of SLA parameters and network conditions used in our experiments.

5.4.1. Latency SLA under Sharp Network Jump. Figure 14 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 one-way server-to-LAN switch delay is 10ms (this is half the machine to machine delay, where a machine can be either a client or a server). After the warm up phase, by $t = 400$s, Figure 14 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). The envelope captures the lowest possible values for consistency (p_{ic}, t_c) and latency (p_{ua}, t_a) allowed by the network partition model (α, t_p) (Theorem 2.7). 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.

First, from Theorem 2.6, 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 the network, we calculate analytically the exact value of α as the fraction of client pairs whose propagation delay exceeds t_p (see Definition 2.5).

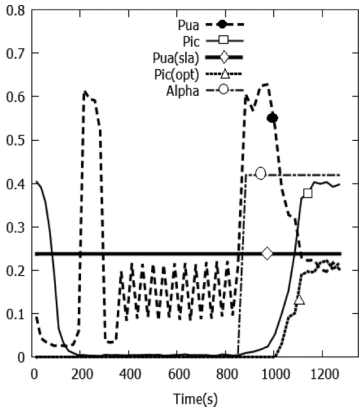

Fig. 14. Latency SLA with PCAP Cassandra under sharp network jump at 800s: Timeline.

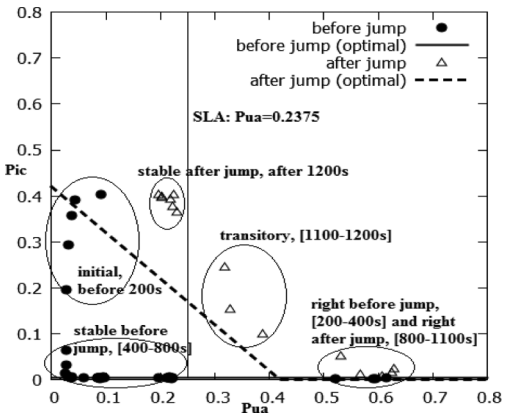

Fig. 15. Latency SLA with PCAP Cassandra under sharp network jump: Consistency-latency scatter plot.

Given this value of α at time t, we can calculate the optimal value of p_{ic} as $p_{ic}(opt) = \max(0, \alpha - p_{ua})$. Figure 14 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 Figure 14, we sharply increase the one-way server-to-LAN delay for five of nine servers from 10ms to 26ms. 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 high 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 Figure 14 by $t = 1200$s.

To visualize how close the PCAP system is to the optimal-achievable envelope, Figure 15 shows 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 Figure 14. 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.

Figure 16 shows the CDF plot for p_{ua} and p_{ic} in the steady-state time interval [400s, 800s] of Figure 14, corresponding to the bottom left cluster from Figure 15. We observe that p_{ua} is always below the SLA.

Figure 17 shows a scatter plot for our PCAP Riak system under a latency SLA ($p_{ua}^{sla} = 0.2375$, $t_a = 150$ms, $t_c = 0$m). The sharp network jump occurs at time $t = 4300$s when we increase the one-way server-to-LAN delay for four of the nine Riak nodes from 10ms to 26ms. It takes about 1200s for p_{ua} to converge to the SLA (at around $t = 1400$s in the warm-up segment and $t = 5500$s in the second segment).

5.4.2. Consistency SLA under Sharp Network Jump. We present consistency SLA results for PCAP Cassandra (PCAP Riak results are similar and are omitted). We use $p_{ic}^{sla} = 0.15$, $t_c = 0$ms, $t_a = 150$ms. The initial one-way server-to-LAN delay is 10ms. At time 750s, we increase the one-way server-to-LAN delay for five of nine nodes to 14ms. This changes α from 0 to 0.42.

Figure 18 shows the scatter plot. First, 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

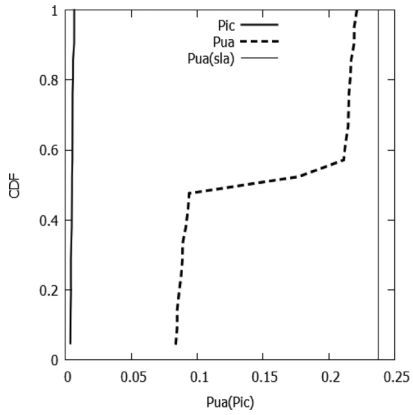

Fig. 16. Latency SLA with PCAP Cassandra under sharp network jump: Steady-state CDF [400s, 800s].

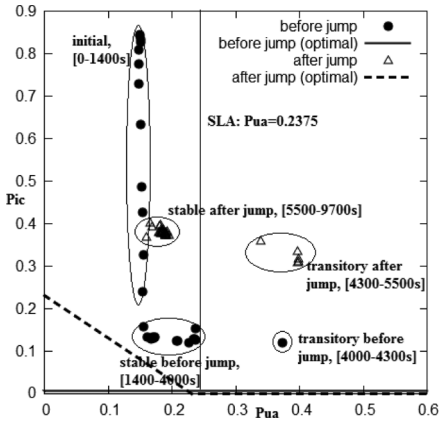

Fig. 17. Latency SLA with PCAP Riak under sharp network jump: Consistency-latency scatter plot.

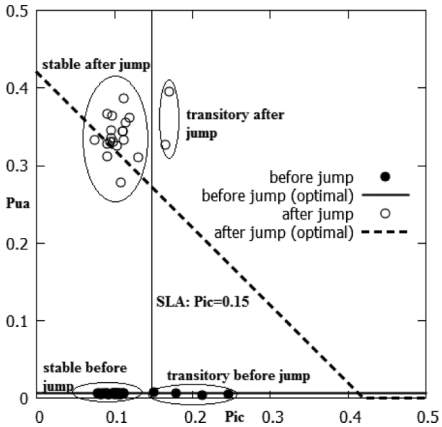

Fig. 18. Consistency SLA with PCAP Cassandra under sharp network jump: Consistency-latency scatter plot.

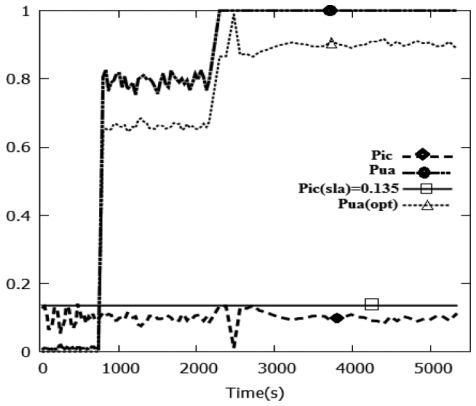

Fig. 19. Consistency SLA with PCAP Cassandra under lognormal delay distribution: Timeline.

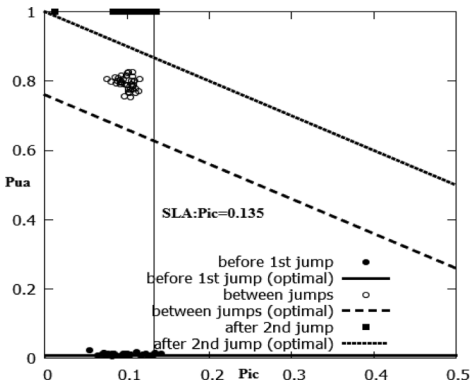

Fig. 20. Consistency SLA with PCAP Cassandra under lognormal delay distribution: Consistency-latency scatter plot.

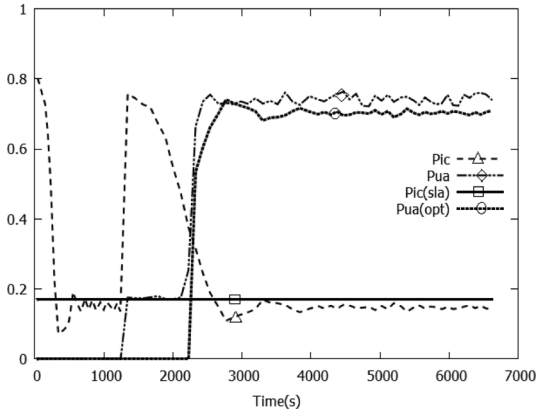

Fig. 21. Consistency SLA with PCAP Riak under lognormal delay distribution: Timeline.

system remains close to the optimal-achievable envelope. The convergence time is about 100s both before and after the jump.

5.4.3. 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).

Based on studies for enterprise data centers [Benson et al. 2010], 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. Figure 19 shows a timeline where initially ($t = 0$ to 800s) 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 4ms and 0.4ms, respectively. Finally, at around 2,100s, μ and σ become 5ms and 0.5ms, respectively. Figure 20 shows the corresponding scatter plot. We observe that, in all three time segments, the inconsistency metric p_{ic} (i) stays below the SLA and (ii) on a sudden network change converges back to the SLA. Additionally, we observe that p_{ua} converges close to its optimal achievable value.

Figure 21 shows the effect of worsening network conditions on PCAP Riak. At around $t = 1300$s we increase μ from 1ms to 4ms, and σ from 0.1ms to 0.5ms. The plot shows

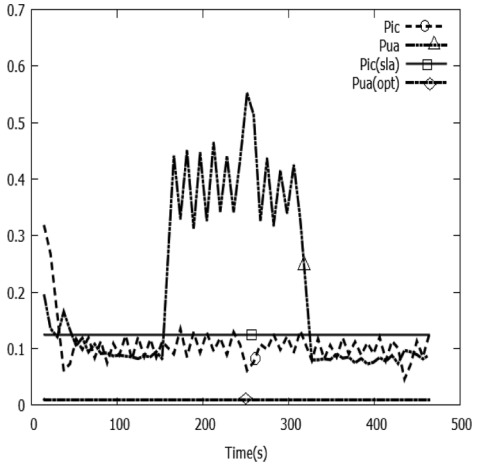

Fig. 22. Consistency SLA ($t_c = 0$ms, $p_{ic} = 0.125$, $t_a = 25$ms) with PCAP Cassandra under lognormal delay (low) distribution: Timeline.

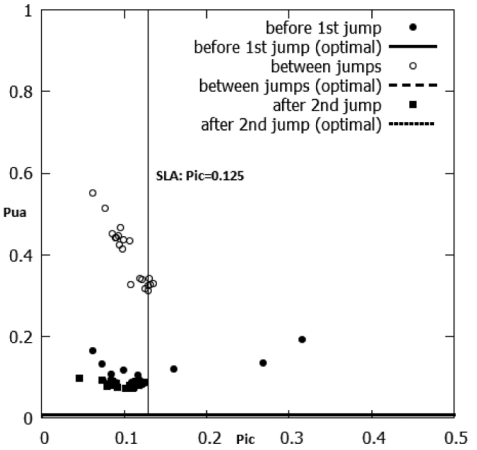

Fig. 23. Consistency SLA ($t_c = 0$ms, $p_{ic} = 0.125$, $t_a = 25$ms) with PCAP Cassandra under lognormal delay (low) distribution: Scatter plot.

that it takes PCAP Riak an additional 1300s to have inconsistency p_{ic} converge to the SLA. Further, the non-SLA metric p_{ua} converges close to the optimal.

So far, all of our experiments used lax timeliness requirements ($t_a = 150$ms, 200ms) and were run on top of relatively high delay networks. Next we perform a stringent consistency SLA experiment ($t_c = 0\ ms$, $p_{ic} = .125$) with a very tight latency timeliness requirement ($t_a = 25$ms). Packet delays are still lognormally distributed but with lower values. Figure 22 shows a timeline where initially the delays are lognormally distributed with $\mu = 1$ms, $\sigma = 0.1$ms. At time $t = 160$s we increase μ and σ to 1.5ms and 0.15ms, respectively. Then, at time $t = 320$s, we decrease μ and σ to return to the initial network conditions. We observe that in all three time segments, p_{ic} stays below the SLA and quickly converges back to the SLA after a network change. Since the network delays are very low throughout the experiment, α is always 0. Thus, the optimal p_{ua} is also 0. We observe that p_{ua} converges very close to optimal before the first jump and after the second jump ($\mu = 1$ms, $\sigma = 0.1$ms). In the middle time segment ($t = 160$s to 320s), p_{ua} degrades in order to meet the consistency SLA under slightly higher packet delays. Figure 23 shows the corresponding scatter plot. We observe that the system is close to the optimal envelope in the first and last time segments, and the SLA is always met. We note that we are far from optimal in the middle time segment, when the network delays are slightly higher. This shows that when the network conditions are relatively good, the PCAP system is close to the optimal envelope, but when situations worsen, we move away. The gap between the system performance and the envelope indicates that the bound (Theorem 2.7) could be improved further. We leave this as an open question.

5.4.4. Effect of Read Repair Rate Knob. All of our deployment experiments use read delay as the only control knob. Figure 24 shows a portion of a run when only read repair rate was used by our PCAP Cassandra system. This was because read 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 (Figure 3). 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 1,000s.

Due to this slow convergence, we conclude that read repair rate is useful only when network delays remain relatively stable. Under continuously changing network

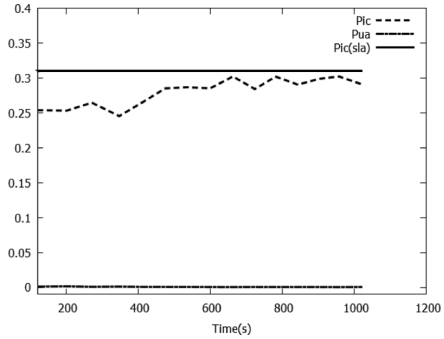

Fig. 24. Effect of read repair rate on PCAP Cassandra. $p_{ic} = 0.31$, $t_c = 0$ms, $t_a = 100$ms.

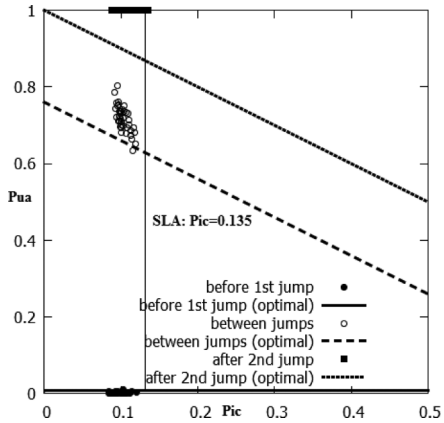

Fig. 25. Scatter plot for same settings as Figure 20 but with 32 servers and 16K ops/s.

conditions (e.g., a lognormal distribution), convergence may be slower and thus read delay should be used as the only control knob.

5.4.5. Scalability. We measure scalability via an increased workload on PCAP Cassandra. Compared to Figure 20, in this new run we increased the number of servers from 9 to 32, and throughput to 16,000ops/s, and ensured that each server stores at least some keys. All other settings are unchanged compared to Figure 20. The result is shown Figure 25. Compared with Figure 20, we observe an improvement with scale—in particular, increasing the number of servers brings the system closer to optimal. As the number of servers in the system increase, the chance of finding a replica server close to a client proxy also increases. This in turn lowers read latency, thus bringing the system closer to the optimal envelope.

5.4.6. Effect of Timeliness Requirement. The timeliness requirements in an SLA directly affect how close the PCAP system is to the optimal-achievable envelope. Figure 26 shows the effect of varying the timeliness parameter t_a in a consistency SLA ($t_c = 0$ms, $p_{ic} = 0.135$) experiment for PCAP Cassandra with 10ms 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 5.4.1).

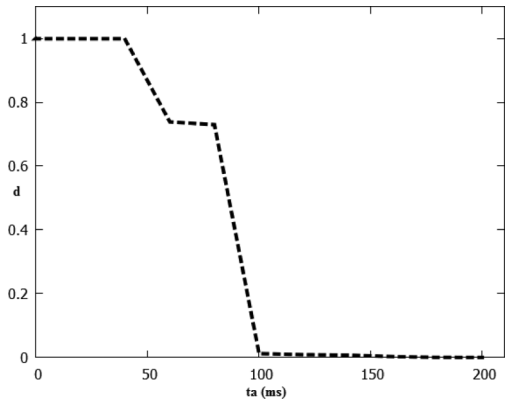

Fig. 26. Effect of timeliness requirement (t_a) on PCAP Cassandra. Consistency SLA with $p_{ic} = 0.135$, $t_c = 0$ms.

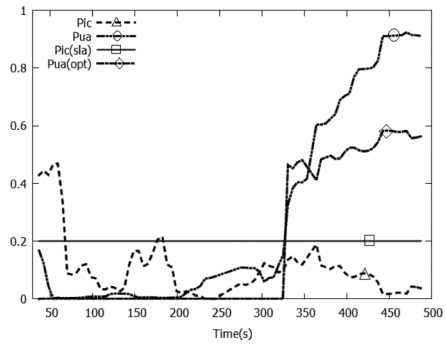

Fig. 27. Consistency SLA with PCAP Cassandra under lognormal delay distribution: Timeline (Passive).

Figure 26 shows that when t_a is too stringent (<100ms), the PCAP system may be far from the optimal envelope even when it satisfies the SLA. In the case of Figure 26, 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.[7] As t_a starts to go beyond this (e.g., $t_a \geq 100$ms), the timeliness requirements are less stringent, and PCAP is essentially optimal (very close to the achievable envelope).

5.4.7. Passive Measurement Approach. So far, all of 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 27, instead of actively injecting operations, we sample ongoing client operations. We estimate p_{ic} and p_{ua} from the 100 latest operations from five 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

[7]Round-trip time for each hop is $2 \times 10 = 20$ms.

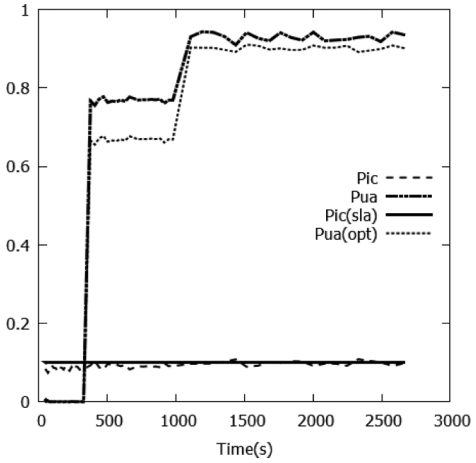

Fig. 28. Consistency SLA ($t_c = 3$ms, $p_{ic} = 0.12$, $t_a = 200$ms) with PCAP Cassandra under log-normal delay (latest) distribution: Timeline.

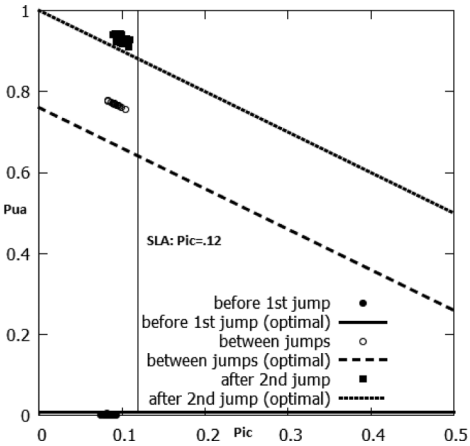

Fig. 29. Consistency SLA ($t_c = 3$ms, $p_{ic} = 0.12$, $t_a = 200$ms) with PCAP Cassandra under log-normal delay (latest) distribution: Scatter plot.

the metrics. Thus we conclude that it is harder to satisfy SLA and optimize the non-SLA metric with the passive approach.

5.4.8. YCSB Benchmark Experiments. So far we have used YCSB client workloads for all our experiments. Specifically, we used a read-heavy (80% reads) workload. However a 80% read-heavy workload is not one of the standard YCSB benchmark workloads [Cooper et al. 2010c]. Thus, to facilitate benchmark comparisons of PCAP with other systems with similar capabilities in the future, we show experimental results for PCAP using two standard YCSB benchmark workloads: (1) Workload A (Update heavy), which has 50/50 reads and writes, and (2) Workload D (Read latest with 95% reads) where most recently inserted records are the most popular.

Read Latest Workload. In this section, we show the results of a PCAP Cassandra consistency SLA experiment ($p_{ic} = 0.12$, $t_c = 3$ms, $t_a = 200$ms) using a read latest workload (YCSB workload d). The timeline plot is shown in Figure 28. Initially ($t = 0$ to 380s) the network delays are lognormally distributed, with the underlying normal distributions of $\mu = 3$ms and $\sigma = 0.3$ms. At $t = 380$s we increase μ and σ to 4ms and 0.4ms, respectively. Finally, at around 1100s, μ and σ become 5ms and 0.5ms, respectively. The timeline plots shows that for read-latest workload (with 95% reads), PCAP Cassandra meets the SLA, and latency p_{ua} converges close to the optimal. This is also evident from the corresponding scatter plot in Figure 29. The trends for read-latest do not exhibit marked differences from our previous experiments with read-heavy workloads (80% reads).

Update Heavy Workload. Next we present results of a PCAP Cassandra consistency SLA experiment ($p_{ic} = 0.12$, $t_c = 3$ms, $t_a = 200$ms) using an update heavy workload (YCSB workload a, 50% writes). For this experiment, we increase the average network delay from 3ms to 4ms at time 540s and from 4ms to 5ms at time 1630s. The timeline plot in Figure 30 indicates poor convergence behavior compared to read-heavy and read-latest distributions. Especially after the second jump, it takes about 1000s to converge back to SLA. This is because our target key-value stores (Cassandra, Riak) rely heavily on read-repair mechanisms to sycnhronize replicas in the background. With an update-heavy workload, read-repair synchronization mechanisms lose their

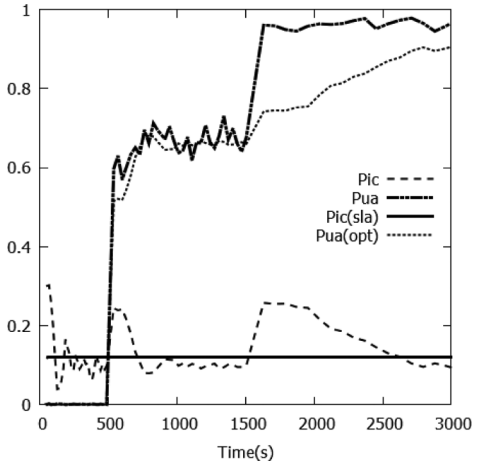

Fig. 30. Consistency SLA (t_c = 3ms, p_{ic} = 0.124, t_a = 200ms) with PCAP Cassandra under lognormal delay (update-heavy) distribution: Timeline.

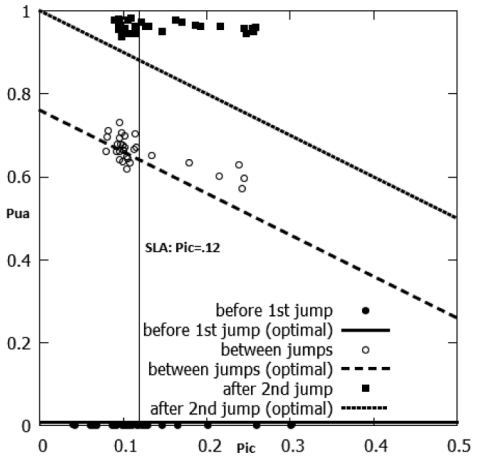

Fig. 31. Consistency SLA (t_c = 3ms, p_{ic} = 0.12, t_a = 200ms) with PCAP Cassandra under lognormal delay (update-heavy) distribution: Scatter plot.

effectiveness, which makes it harder to guarantee consistency SLA convergence. The scatter plot in Figure 31, however, shows that PCAP Cassandra performs close to the optimal envelope, except for some outliers that are transient states before convergence. It should be noted that an update-heavy workload is not a realistic workload for many key-value stores, and we have only presented results with such workloads for future comparison with other systems using standard benchmarks like YCSB.

5.5. GeoPCAP Evaluation

We evaluate GeoPCAP with a Monte Carlo simulation. In our setup, we have four data centers, among which three are remote data centers holding replicas of a key, and the fourth one is the local data center. At each iteration, we estimate t-freshness per data center using a variation of the well-known WARS model [Bailis et al. 2014]. The WARS model is based on Dynamo style quorum systems [DeCandia et al. 2007], where data staleness is due to read and write message reordering. The model has four components. W represents the message delay from coordinator to replica. The acknowledgment from the replica back to the coordinator is modeled by a random variable A. The read message delay from coordinator to replica, and the acknowledgment back are represented by R and S, respectively. A read will be stale if a read is acknowledged before a write reaches the replica, i.e., $R + S < W + A$. In our simulation, we ignore the A component since we do not need to wait for a write to finish before a read starts. We use the LinkedIn solid-state drive (SSD) disk latency distribution [Bailis et al. 2014], Table 3 for read/write operation latency values.

We model the WAN delay using a normal distribution $N(20\text{ms}, \sqrt{2}\text{ms})$ based on results from Baldoni et al. [2006]. Each simulation runs for 300 iterations. At each iteration, we run the PID control loop (Figure 9) to estimate a new value for geo-delay Δ and sleep for 1s. All reads in the following iteration are delayed at the local data center by Δ. At iteration 150, we inject a jump by increasing the mean and standard deviation of each WAN link delay normal distribution to 22ms and $\sqrt{2.2}$ms, respectively. We show only results for consistency and latency SLA for the `ALL` composition. The `QUICKEST` composition results are similar and are omitted.

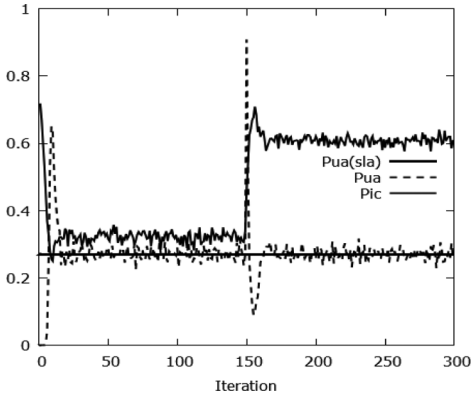

Fig. 32. GeoPCAP SLA timeline for L SLA (ALL).

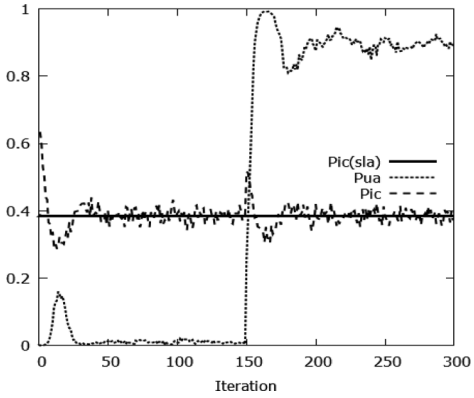

Fig. 33. GeoPCAP SLA timeline for C SLA (ALL).

Figure 32 shows the timeline of SLA convergence for GeoPCAP latency SLA ($p_{ua}^{sla} = 0.27$, $ta^{sla} = 25$ms, $tc^{sla} = 0.1$ms). It should be noted that a latency SLA ($p_{ua}^{sla} = 0.27$, $ta^{sla} = 25$ms, $tc^{sla} = 0.1$ms) for GeoPCAP means that when using the ALL composition rule (all individual data-center deployments must satisfy a latency deadline), each single data-center PCAP needs to provide an SLA specified as ($p_{ua}^{sla} = 0.1$, $ta^{sla} = 25$ms, $tc^{sla} = 0.1$ms) (using the composition rule in the third row of Figure 6). We observe that using the PID controller ($k_p = 1$, $k_d = 0.5$, $k_i = 0.5$), both the SLA and the other metric converge within five iterations initially and also after the jump. Figure 34 shows the corresponding evolution of the geo-delay control knob. Before the jump, the read delay converges to around 5ms. After the jump, the WAN delay increase forces the geo-delay to converge to a lower value (around 3ms) in order to meet the latency SLA.

Figure 33 shows the consistency SLA ($p_{ic}^{sla} = 0.38$, $tc^{sla} = 1$ms, $ta^{sla} = 25$ms) timeline. Here convergence takes 25 iterations, thanks to the PID controller (($k_p = 1$, $k_d = 0.8$, $k_i = 0.5$)). We needed a slightly higher value for the differential gain k_d to deal with increased oscillation for the consistency SLA experiment. Note that the p_{ic}^{sla} value of 0.38 forces a smaller per-data-center p_{ic} convergence. The corresponding geo-delay evolution (Figure 35) initially converges to around 3ms before the jump and converges to around 5ms after the jump to enforce the consistency SLA after the delay increase.

We also repeated the latency SLA experiment with the ALL composition (Figures 32 and 34) using the multiplicative control approach (Section 3.3) instead of the PID control approach. Figure 36 shows the corresponding geo-delay trend compared to Figure 34. Comparing the two figures, we observe that although the multiplicative strategy converges as fast as the PID approach both before and after the delay jump, the read delay value keeps oscillating around the optimal value. Such oscillations cannot be avoided in the multiplicative approach, since at steady state the control loop keeps changing direction with a unit step size. Compared to the multiplicative approach, the PID control approach is smoother and has fewer oscillations.

6. RELATED WORK

6.1. Consistency-Latency Tradeoffs

In addition to the Weak CAP Principle [Fox and Brewer 1999] and PACELC [Abadi 2012] discussed in Section 1, there has been work on theoretically characterizing the tradeoff between latency and strong consistency models. Attiya and Welch [1994] studied the tradeoff between latency and linearizability and sequential consistency. Subsequent work has explored linearizablity under different delay models [Eleftheriou

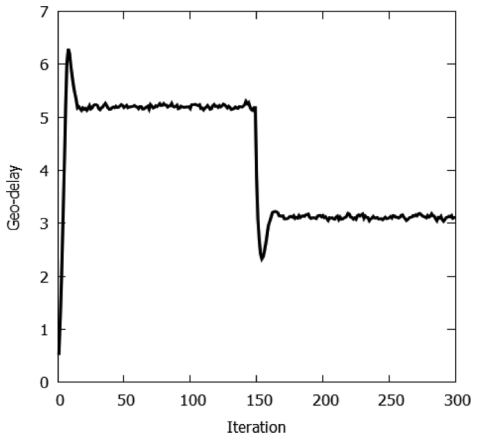

Fig. 34. Geo-delay timeline for L SLA (ALL) (Figure 32).

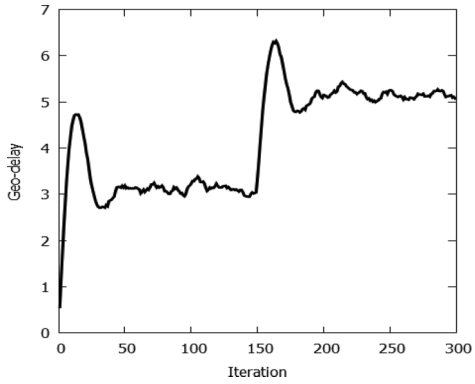

Fig. 35. Geo-delay timeline for C SLA (ALL) (Figure 33).

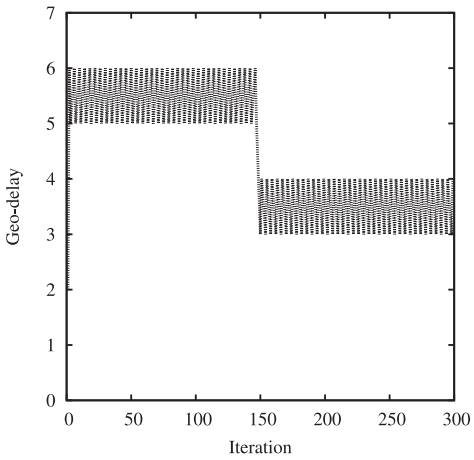

Fig. 36. Geo-delay timeline for A SLA (ALL) with the multiplicative approach.

and Mavronicolas 1999; Mavronicolas and Roth 1999]. All these articles are concerned with strong consistency models, whereas we consider t-freshness, which models data freshness in eventually consistent systems. Moreover, their delay models differ from our partition model. There has been theoretical work on probabilistic quorum systems [Malkhi et al. 1997; Lee and Welch 2001; Abraham and Malkhi 2005]. Their consistency models are different from ours; moreover, they did not consider the tradeoff between consistency and availability.

There are two classes of systems that are closest to our work. The first class of systems are concerned with metrics for measuring data freshness or staleness. We do not compare our work against this class of systems in this article, as it is not our goal to propose yet another consistency model or metric. Bailis et al. [2013, 2014] proposed a probabilistic consistency model (PBS) for quorum-based stores but did not consider latency, soft partitions, or the CAP theorem. Golab et al. [2011] proposed a time-based staleness metric called Δ-atomicity. Δ-atomicity is considered the gold standard for measuring atomicity violations (staleness) across multiple read and write operations.

The Γ metric [Golab et al. 2014] is inspired by the Δ metric and improves on it on multiple fronts. For example, the Γ metric makes fewer technical assumptions than the Δ metric and produces less noisy results. It is also more robust against clock skew. All these related data freshness metrics cannot be directly compared to our t-freshness metric. The reason is that, unlike our metric, which considers write start times, these existing metrics consider end time of write operations when calculating data freshness.

The second class of systems deal with adaptive mechanisms for meeting consistency-latency SLAs for key-value stores. The Pileus system [Terry et al. 2013] considers families of consistency/latency SLAs and requires the application to specify a utility value with each SLA. In comparison, PCAP considers probabilistic metrics of p_{ic}, p_{ua}. Tuba [Ardekani and Terry 2014] extended the predefined and static Pileus mechanisms with dynamic replica reconfiguration mechanisms to maximize Pileus style utility functions without impacting client read and write operations. Golab and Wylie [2014] proposed consistency amplification, which is a framework for supporting consistency SLAs by injecting delays at servers or clients. In comparison, in our PCAP system, we only add delays at servers. McKenzie et al. [2015] proposed CPQ, which is a technique to randomly choose between multiple discrete consistency levels for fine-grained consistency-latency tuning, and compared CPQ against consistency amplification. Compared to all these systems where the goal is to meet SLAs, in our work, we also (1) quantitatively characterize the (un-)achievable consistency-latency tradeoff envelope and (2) show how to design systems that perform close to this envelope in addition to (3) meeting SLAs. The PCAP system can be set up to work with any of these SLAs listed above; but we do not do this in the article since our main goal is to measure how close the PCAP system is to the optimal consistency-latency envelope.

Recently, there has been work on declarative ways to specify application consistency and latency requirements—PCAP proposes mechanisms to satisfy such specifications [Sivaramakrishnan et al. 2015].

6.2. Adaptive Systems

There are a few existing systems that controls consistency in storage systems. FRACS [Zhang and Zhang 2003] controls consistency by allowing replicas to buffer updates up to a given staleness. AQuA [Krishnamurthy et al. 2003] continuously moves replicas between "strong" and "weak" consistency groups to implement different consistency levels. Fox and Brewer [1999] showed how to trade consistency (harvest) for availability (yield) in the context of the Inktomi search engine. While harvest and yield capture continuously changing consistency and availability conditions, we characterize the consistency-availability (latency) tradeoff in a quantitative manner. TACT (Tunable Availability and Consistency Tradeoffs) [Yu and Vahdat 2002] 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 Davidson et al. [2013]; 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) that are correlated with garbage collection in a specific key-value store (Apache Cassandra) [Fan et al. 2015]. Similar techniques have been used to guarantee causal consistency for client-side applications [Zawirski et al. 2015]. Simba [Perkins et al. 2015] proposes new consistency abstractions for mobile application data synchronization services and allows applications to choose among various consistency models.

For stream processing, Gedik et al. [2014] proposed a control algorithm to compute the optimal resource requirements to meet throughput requirements. There has been work on adaptive elasticity control for storage [Lim et al. 2010] and adaptively tuning Hadoop clusters to meet SLAs [Herodotou et al. 2011]. Compared to the controllers present in these systems, our PCAP controller achieves control objectives [Hellerstein et al. 2004] using a different set of techniques to meet SLAs for key-value stores.

6.3. Composition

Composing local policies to form global policies is well studied in other domains, for example, quality of service (QoS) in multimedia networks [Liang and Nahrstedt 2006], software-defined network composition [Monsanto et al. 2013], and web-service orchestration [Dustdar and Schreiner 2005]. Our composition techniques are aimed at consistency and latency guarantees for geo-distributed systems.

7. SUMMARY

In this article, we first formulated and proved a probabilistic variation of the CAP theorem which took into account probabilistic models for consistency, latency, and soft partitions within a data center. Our theorems show the un-achievable envelope, that is, which combinations of these three models make them impossible to achieve together. We then showed how to design systems (called PCAP) that (1) perform close to this optimal envelope and (2) can meet consistency and latency SLAs derived from the corresponding models. We then incorporated these SLAs into Apache Cassandra and Riak running in a single data center. We also extended our PCAP system from a single data center to multiple geo-distributed data centers. 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. Simulations of our GeoPCAP system also showed SLA satisfaction for applications spanning multiple data centers.

REFERENCES

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

Ittai Abraham and Dahlia Malkhi. 2005. Probabilistic quorums for dynamic systems. *Distrib. Comput.* 18, 2 (2005), 113–124.

Masoud Saeida Ardekani and Douglas B. Terry. 2014. A self-configurable geo-replicated cloud storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, USA, 367–381.

K. J. Astrom and T. Hagglund. 1995. *PID Controllers: Theory, Design, and Tuning, 2nd Ed.* The Instrument, Systems, and Automation Society, Research Triangle Park, NC.

Hagit Attiya and Jennifer L. Welch. 1994. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.* 12, 2 (1994), 91–122.

Peter Bailis and Ali Ghodsi. 2013. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue* 11, 3 (2013), 20:20–20:32.

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2014. Quantifying eventual consistency with PBS. *VLDB J.* 23, 2 (2014), 279–302.

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2013. PBS at work: Advancing data management with consistency metrics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, New York, NY. 1113–1116.

R. Baldoni, C. Marchetti, and A. Virgillito. 2006. Impact of WAN channel behavior on end-to-end latency of replication protocols. In *Proceedings of European Dependable Computing Conference (EDCC)*. 109–118.

Jeff Barr. 2013. Real-time ad impression bids using DynamoDB. Retrieved from http://goo.gl/C7gdpc.

Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of datacenters in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC)*. 267–280.

Eric Brewer. 2000. Towards robust distributed systems (abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*.

Eric Brewer. 2010. A certain freedom: Thoughts on the CAP theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 335–335.

Adrian Cockcroft. 2013. Dystopia as a service (invited talk). In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010a. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. 143–154.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010b. Yahoo! Cloud serving benchmark (YCSB). Retrieved from http://goo.gl/GiA5c.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010c. Yahoo! Cloud serving benchmark (YCSB) workloads. Retrieved from https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 251–264.

DataStax. 2016. Configuring data consistency. Retrieved from https://goo.gl/PKNUXV.

Aaron Davidson, Aviad Rubinstein, Anirudh Todi, Peter Bailis, and Shivaram Venkataraman. 2013. Adaptive hybrid quorums in practical settings. Retrieved from http://goo.gl/LbRSW3.

Jeff Dean. 2009. Design, Lessons and Advice from Building Large Distributed Systems. Retrieved from http://goo.gl/HGJqUh.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, DC, 205–220.

Schahram Dustdar and Wolfgang Schreiner. 2005. A survey on web services composition. *Int. J. Web Grid Serv.* 1, 1 (2005), 1–30.

Maria Eleftheriou and Marios Mavronicolas. 1999. Linearizability in the presence of drifting clocks and under different delay assumptions. In *Distributed Computing*. Vol. 1693. 327–341.

Hua Fan, Aditya Ramaraju, Marlon McKenzie, Wojciech Golab, and Bernard Wong. 2015. Understanding the causes of consistency anomalies in apache Cassandra. *Proc. VLDB Endow.* 8, 7 (2015), 810–813.

Armando Fox and Eric A. Brewer. 1999. Harvest, yield, and scalable tolerant systems. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems (HotOS)*. 174–178.

Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1447–1463.

Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2 (2002), 51–59.

Seth Gilbert and Nancy Lynch. 2012. Perspectives on the CAP theorem. *IEEE Comput.* 45, 2 (2012), 30–36.

Eric Gilmore. 2011. Cassandra multi data-center deployment. Retrieved from http://goo.gl/aA8YIS.

Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. 2011. Scalable consistency in scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 15–28.

Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. 2011. Analyzing consistency properties for fun and profit. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, 197–206.

Wojciech Golab, Muntasir Raihan Rahman, Alvin AuYoung, Kimberly Keeton, and Indranil Gupta. 2014. Client-centric benchmarking of eventual consistency for cloud storage systems. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*. Madrid, Spain.

Wojciech Golab and John Johnson Wylie. 2014. Providing A Measure Representing an Instantaneous Data Consistency Level. (Jan. 2014). US Patent Application 20,140,032,504.

Andy Gross. 2009. Basho riak. Retrieved from http://basho.com/riak/.

Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. 2004. *Feedback Control of Computing Systems*. John Wiley & Sons.

Herodotos Herodotou, Fei Dong, and Shivnath Babu. 2011. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 18:1–18:14.

Sudha Krishnamurthy, William H. Sanders, and Michel Cukier. 2003. An adaptive quality of service aware middleware for replicated services. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 14 (2003), 1112–1125.

Avinash Lakshman and Prashant Malik. 2008. Apache Cassandra. Retrieved from http://cassandra.apache.org/.

Keith Lang. 2009. Amazon: Milliseconds means money. Retrieved from http://goo.gl/fs9pZb.

Hyunyoung Lee and Jennifer L. Welch. 2001. Applications of probabilistic quorums to iterative algorithms. In *Proceedings of the The 21st International Conference on Distributed Computing Systems (ICDCS)*. 21.

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 265–278.

Jin Liang and Klara Nahrstedt. 2006. Service composition for generic service graphs. *Multimedia Syst.* 11, 6 (2006), 568–581.

Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. 2010. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC)*. 1–10.

LinkedIn. 2009. Project Voldemort. Retrieved from http://goo.gl/9uhLoU.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. 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 (SOSP)*. 401–416.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 313–328.

Dahlia Malkhi, Michael Reiter, and Rebecca Wright. 1997. Probabilistic quorum systems. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 267–273.

Marios Mavronicolas and Dan Roth. 1999. Linearizable read/write objects. *Theoret. Comput. Sci.* 220, 1 (1999), 267–319.

Marlon McKenzie, Hua Fan, and Wojciech Golab. 2015. Fine-tuning the consistency-latency trade-off in quorum-replicated distributed storage systems. In *Proceedings of the 2015 IEEE International Conference on Big Data*. 1708–1717.

Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 1–14.

Ron Peled. 2010. Activo: "Why low latency matters?" Retrieved from http://goo.gl/2XQ8Ul.

Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V. Madhyastha, and Cristian Ungureanu. 2015. Simba: Tunable end-to-end data consistency for mobile apps. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. 7:1–7:16.

Marc Shapiro. 1986. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. 198–204.

Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS)*. 386–400.

K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 413–424.

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 149–160.

Shlomo Swidler. 2009. Consistency in Amazon S3. Retrieved from http://goo.gl/yhAoJy.

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 309–324.

Werner Vogels. 2009. Eventually consistent. *Commun. ACM* (2009), 40–44.

Hengfeng Wei, Yu Huang, Jiannong Cao, and Jian Lu. 2015. Almost strong consistency: "Good enough" in distributed storage systems. Retrieved from http://arxiv.org/abs/1507.01663.

Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*. 255–270.

Haifeng Yu and Amin Vahdat. 2002. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.* 20, 3 (2002), 239–282.

Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*. 75–87.

Chi Zhang and Zheng Zhang. 2003. Trading replication consistency for performance and availability: An adaptive approach. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*. 687–695.

Received August 2015; revised May 2016; accepted September 2016