Wyatt Lloyd
Computer Science Department
35 Olden St.
Princeton, NJ 08540
(814) 880-1392
wlloyd@cs.princeton.edu

December 14, 2012

To Whom It May Concern:

I am seeking a tenure-track position as an assistant professor in your department. I am a Ph.D. candidate in the Computer Science Department at Princeton University, and I expect to graduate this year. My research interests include the distributed systems and networking problems that underlie the architecture of large-scale websites, cloud computing, and big data.

I have enclosed my curriculum vitae with a list of references, research statement, teaching statement, and four representative papers. The most up-to-date version of these materials is available online at http://www.cs.princeton.edu/~wlloyd/application/.

I look forward to discussing my application with you.

Sincerely,

Wyatt Lloyd

encl:
- Curriculum Vitae (including name of references)
- Research Statement
- Teaching Statement
- "Stronger Semantics for Low-Latency Geo-Replicated Storage"
  Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, David G. Andersen
  to appear in NSDI 2013, 14 pages (preprint)
- "Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS"
  Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, David G. Andersen
  from SOSP 2011, 16 pages
- "Coercing Clients into Facilitating Failover for Object Delivery"
  Wyatt Lloyd, Michael J. Freedman
  from DSN 2011, 12 pages
- "Prophecy: Using History for High-Throughput Fault Tolerance"
  Siddhartha Sen, **Wyatt Lloyd**, Michael J. Freedman
  from NSDI 2010, 16 pages

Wyatt Lloyd

http://www.cs.princeton.edu/~wlloyd

205 Hudson St., Apt 601 wlloyd@cs.princeton.edu
Hoboken, N.J. 07030 (814) 880-1392

Education

Princeton University . . . . Princeton, NJ

Ph.D. in Computer Science . . . . Expected June 2013
M.A. in Computer Science 2009
Advisor: Michael J. Freedman

Pennsylvania State University, University Park . . . . State College, PA

B.S. in Computer Science with distinction 2007
Schreyer Honors College
Advisor: Thomas F. La Porta

Research Interests

The distributed systems and networking problems that underlie the architecture of large-scale websites, cloud computing, and big data.

Dissertation

2010– **Stronger Consistency and Semantics for Geo-Replicated Storage.** Geo-replicated storage systems provide the backend for massive-scale websites such as Twitter and Facebook, storing data that includes your profile, friends lists, and status updates. These storage systems seek to provide an "always-on" experience where operations always complete quickly, because of a widely demonstrated link between page load times, user engagement, and revenue. We term systems that can handle data of this scale and provide the always-on experience *ALPS systems*, because they provide four key properties—availability, low latency, partition tolerance, and scalability.

Our COPS [2] system is the first distributed data store to guarantee the ALPS properties and achieve consistency stronger than eventual. Eventual consistency specifies only that writes in one datacenter eventually show up in the others. Causal consistency, which is what COPS provides, maintains the partial order over operations establish by potential causality. Under causal consistency, all of a user's operations appear in the order they are issued and interactions between users, e.g., conversations in comments, appear in their correct order as well. This improvement in consistency gives users a better experience and makes the data store easier for programmers to reason about. A key technical contribution of the COPS work is its fully distributed and scalable architecture that uses explicit metadata and off-path dependency checks to enforce ordering instead of relying on any single point of coordination.

Our Eiger [1] system further pushes on the semantics an ALPS data store can provide. Eiger provides high-performance, guaranteed low-latency read-only and

write-only transactions across the thousands of machines in a cluster. Read-only transactions allow a client to observe a consistent snapshot of an entire cluster. Write-only transactions allow clients to atomically write many values spread across many servers at a single point in time. One important use case for write-only transaction is maintaining symmetrical relationships, e.g., Alice "isAFriendOf" Bob and Bob "isAFriendOf" Alice should both appear or disappear at the same time. Eiger also improves the semantics of ALPS data stores by providing the column-family data model—which is used in BigTable and Cassandra, and can be used to built real applications like Facebook—instead of the key-value data model provided by COPS—which is useful mainly as an opaque cache.

My dissertation research shows that ALPS systems do not need to settle for eventual consistency and weak semantics. Taken together, Eiger and COPS show that causal consistency and stronger semantics are possible for low-latency geo-replicated storage.

Other Research Experience

2009–2011 **Low-Overhead Transparent Recovery for Static Content.** Client connections to web services break when the particular server they are connected to fails or is taken down for maintenance. We designed and built TRODS [3], a system that transparently recovers connections to web services that delivers static content, e.g., photos or videos. TRODS is implemented as a server-side kernel module for immediate deployability, it works with unmodified services and clients. The key insight in TRODS is its use of cross-layer visibility and control: It derives reliable storage for application-level state from the mechanics of the transport layer. In contrast with more general recovery techniques, the overhead of TRODS is minimal. It provides throughput-per-server competitive with unmodified HTTP services, enabling recovery without additional capital expenditures.

2007–2010 **Using History for High-Throughput Fault Tolerance.** Byzantine fault-tolerant (BFT) replication provides protection against arbitrary and malicious faults, but its performance does not scale with cluster size. We designed and built Prophecy [4], a system that interposes itself between clients and any replicated service to scale throughput for read-mostly workloads. Prophecy relaxes consistency to delay-once linearizability so it can perform fast, load-balanced reads when results are historically consistent, and slow, replicated reads otherwise. This dramatically increases the throughput of replicated services, e.g., the throughput of a 4 node Prophecy web service is ~4X the throughput of a 4 node PBFT web service.

2007 **IP Address Passing for VANETs.** In Vehicular Ad-hoc Networks (VANETs), vehicles have short connection times when moving past wireless access points. The time required for acquiring IP addresses via DHCP consumes a significant portion of each connection. We reduce the connection time to under a tenth of a second by passing IP addresses between vehicles. Our implementation improves efficiency, reduces latency, and increases vehicle connectivity without modifying either DHCP or AP software [5].

2006–2007 **Multi-Class Overload Controls for SIP Servers.** When SIP servers that are used for signaling in VoIP network are overloaded, call-setup latency increases significantly and critical calls—e.g., 911 calls—can be denied. My undergraduate thesis on multi-class overload controls [6] reduces call latency by suppressing retransmissions and prioritizes critical calls so they always connect.

Professional Experience

9/07– **Research Assistant.** Princeton University, Princeton, NJ
Major projects include providing stronger consistency for scalable storage systems (COPS), providing stronger semantics for scalable storage systems (Eiger), enabling transparent connection recovery for web services (TRODS), and using history for high-throughput fault tolerance (Prophecy).

5/12–8/12 **Ph.D. Intern.** Facebook, New York, NY
Worked on a distributed-storage-systems team on a project to improve caching for static content. Was the first intern at the new New York office.

6/10–9/10 **Summer Research Fellow.** Intel Labs Pittsburgh / CMU, Pittsburgh, PA
Began leading the COPS project, a collaborative effort between Princeton University, Intel Labs, and Carnegie Mellon University.

6/07–9/07 **Intern-Student Engineer.** The Boeing Company, Anaheim, CA
Worked on an internal research and development project on routing in multi-tier wireless networks as part of the Network Systems group.

5/06–9/06 **Intern-Student Engineer.** The Boeing Company, Anaheim, CA
Worked on an internal research and development project that utilized DHCP for intra-domain mobility management as part of the Network Systems group.

Teaching Experience

11/29/12 **Guest Lecturer.** Distributed Systems, (CMU) 15-440
Lectured on COPS to introduce cutting-edge research to undergraduates.

10/4/12 **Guest Lecturer.** Advanced Computer Networks, COS-561
Lectured on inter-domain routing with BGP and led a discussion of research papers on network isolation and software-defined networking.

2/09–6/09 **Teaching Assistant.** Computer Networks, COS-461
Graded, held office hours, helped design exams, and taught exam-review sessions.

9/08–1/09 **Teaching Assistant.** General Computer Science, COS-126
Graded, held office hours, helped design exams, and taught twice-weekly recitations.

Service

11/9/12 **Panelist.** Princeton Women in Computer Science Graduate School Panel
Shared experiences and advice about graduate school.

10/11–	**Regional Lead.** Organized events for Princeton region and served on the advisory board.	Siebel Scholars Foundation
6/11–8/11 6/09–8/09	**Student Advisor.** Advised novice undergraduate programmers on 6-week-long projects.	Princeton Summer Programming Experience
2/06–5/07	**Student Representative.** Helped shape undergraduate Computer Science curriculum.	Penn State CSE Curriculum Committee

Honors

2012	Wu Prize for Excellence (Princeton)
2012	Facebook Fellowship Finalist
2012	Siebel Scholar
2007	Princeton University Graduate Fellowship
2003-2007	Dean's List (Penn State)
2003-2007	Schreyer Honors College Scholar (Penn State)
2006	College of Engineering General Scholarship (Penn State)
2003	Maryland State Distinguished Scholar
2002	National Merit Scholarship Honorable Mention
2000	Eagle Scout

Refereed Conference Publications

[1] **Wyatt Lloyd**, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. To appear in *Proc. 10th Symposium on Networked Systems Design and Implementation (NSDI 13)*, April 2013. 14 pages.

[2] **Wyatt Lloyd**, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP 11)*, October 2011. 16 pages.

[3] **Wyatt Lloyd** and Michael J. Freedman. Coercing Clients into Facilitating Failover for Object Delivery. In *Proc. 41st IEEE/IFIP International Conference on Dependable Systems and Networks, Dependable Computing and Communication Symposium (DCCS) track (DSN 11)*, June 2011. 12 pages.

[4] Siddhartha Sen, **Wyatt Lloyd**, and Michael J. Freedman. Prophecy: Using History for High-Throughput Fault Tolerance. In *Proc. 7th Symposium on Networked Systems Design and Implementation (NSDI 10)*, April 2010. 16 pages.

[5] Todd Arnold, **Wyatt Lloyd**, Jing Zhao, and Guohong Cao. IP Address Passing for VANETs. In *Proc. 6th IEEE International Conference on Pervasive Computing and Communications (PERCOM 08)*, March 2008. 10 pages.

Theses

[6] **Wyatt Lloyd**. Multi Class Overload Controls for SIP Servers. *Honors Thesis*, The Pennsylvania State University, May 2007.

Refereed Conference Presentations

[7] Stronger Semantics for Low-Latency Geo-Replicated Storage. To appear at *10th Symposium on Networked Systems Design and Implementation (NSDI 13)*, April 2013.

[8] Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *23rd ACM Symposium on Operating Systems Principles (SOSP 11)*, October 2011.

[9] Coercing Clients into Facilitating Failover for Object Delivery. In *41st IEEE/IFIP International Conference on Dependable Systems and Networks, Dependable Computing and Communication Symposium (DCCS) track (DSN 11)*, June 2011.

Other Presentations

[10] Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. Facebook Ph.D. Intern and Distributed Systems Reading Group Talk, August 2012.

[11] Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. Berkeley, Cloud Seminar, April 2012.

[12] Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. Intel Science and Technology Center on Cloud Computing Retreat, Research Talk, December 2011.

[13] Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. University of Maryland, SysChat Group Talk, October 2011.

[14] Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. Johns Hopkins University, Computer Science Seminar, October 2011.

References

Prof. Michael J. Freedman
Assistant Professor
Computer Science Department
Princeton University
`mfreed@cs.princeton.edu`

Prof. David G. Andersen
Associate Professor
Computer Science Department
Carnegie Mellon University
`dga@cs.cmu.edu`

Dr. Michael Kaminsky
Senior Research Scientist
ISTC for Cloud Computing
Intel Labs
`michael.e.kaminksy@intel.com`

Prof. Mike Dahlin
Professor
Computer Science Department
The University of Texas at Austin
`dahlin@cs.utexas.edu`

Research Statement

Wyatt Lloyd

Vision

My research addresses emerging problems in the massive-scale distributed systems that support big data. The recent and dramatic growth in the demands on these systems—on their ability to store, process, and manage large data volumes—makes this area ripe for new research. This new scale, e.g., all of Twitter's tweets, requires massively distributed systems with hundreds or thousands or more machines cooperating to provide the necessary capacity and throughput.

To handle the magnitude of this data, recent systems from industry and academia have stressed performance and scalability at the cost of strong semantic properties, e.g., linearizability and transactions, yet these same properties make systems easier to use and understand. My research questions whether such sacrifices are necessary and seeks to identify properties that make these systems easier to program to and reason about that are compatible with massive scale. I pursue properties that are rigorously defined for the clarity they bring to programmers using the systems, e.g., causal consistency instead of eventual, or guaranteeing low latency for all operations.

Looking at my research from another direction, it reexamines conventional design decisions and approaches given the new reality of big data. Centralized approaches that were simple, straightforward, and effective in the small systems of the past quickly hit bottlenecks that prevent scaling to the large systems of today. In contrast, I design systems that provide strong properties while keeping all operations distributed, so they can scale to the even larger systems of tomorrow.

My focus on scalability is maintained throughout the research process; I include it as a primary design requirement and then build prototypes that are tested with real data at scale. While building and experimentally verifying that a design is scalable is useful and necessary in itself, I have also found that doing so often exposes unanticipated bottlenecks and can reveal important new research topics, as discussed in my future directions. This focus on designing and building massive-scale systems that provide strong, rigorous properties defines my niche as a researcher.

Dissertation Research

Geo-replicated storage systems provide the backend for massive-scale websites like Facebook, storing data that includes your profile, friends list, and status updates. These storage systems seek to provide an "always-on" experience where operations always complete quickly, because of a widely demonstrated link between page load times, user engagement, and revenue. We term systems that can handle such data at scale and provide an always-on experience as *ALPS systems*, because they provide four key properties: Availability, Low latency, Partition tolerance, and Scalability.

Previous ALPS systems such as Amazon's Dynamo, LinkedIn's Project Voldemort, and Facebook's Cassandra (in some configurations) all made large usability sacrifices in pursuit of their scale and performance goals. These sacrifices—such as providing only eventual consistency, which gives no ordering guarantees about operations—make it hard for programmers to reason about the system and result in an end-user experience that is far from ideal. My dissertation research shows that these sacrifices are not fundamental; stronger consistency and semantics are achievable for ALPS storage systems.

Yet, known theoretical results show that low latency and the strongest types of consistency are incompatible.[1] Knowing this, the recent spate of low-latency geo-replicated systems settled for the weakest semantic property, eventual consistency. The first part of my dissertation research, COPS, shows that this sacrifice is not necessary. COPS is the first ALPS system to provide causal consistency, a middle ground between the two extremes where operations always appear in an order consistent with potential causality, e.g., all of a user's operations and all conversations between users appear in their original order. This improvement in consistency makes the distributed storage more intuitive for programmers to use and gives end users more of the experience they expect.

One key technical contribution in COPS is a design focused on the scalability of clusters where the keyspace is partitioned across nodes, replication is done in parallel from all nodes in each cluster, and then nodes use dependency metadata associated with the updates to issue distributed checks that ensure operations are always applied in the correct causal order. Naively, dependency metadata grows exponentially and throttles performance. COPS avoids this problem using multiple types of garbage collection and by exploiting the transitive structure inherent in the graph of potential causality.

COPS's use of distributed metadata runs counter to the traditional wisdom for enforcing causal consistency, which was to exchange logs of operations and then replay them at other locations. The log-exchange approach works well and admits a simple implementation when all data can reside on a single machine. With the new realities of massive scale where data is spread across many machines, however, the log-exchange approach breaks down as logging updates to all machines in a cluster into one place becomes a bottleneck. In contrast, because of COPS's distributed design, it is the first scalable and causally consistent system.

The big data in ALPS systems is spread across thousands of nodes, yet previous systems provided only inconsistent batch operations to read and write data across multiple nodes. Eiger, the second part of my dissertation research, shows that much stronger semantics are possible. These stronger semantics include read-only and write-only transactions that consistently read or write data spread across all the nodes in a cluster. Eiger's semantics also include counter columns and the hierarchical column-family data model used in BigTable and Cassandra, which makes building applications atop it much simpler. The limited transactions and rich data model in Eiger do not come at the expense of high scalability or performance, and designing the system to ensure this was one of Eiger's main challenges.

In particular, to guarantee low latency Eiger must eschew locks and blocking, the typical techniques used for transactions. And to enable scaling it must avoid the centralization that is also typical for transactions. Eiger overcomes both these challenges using distributed algorithms that utilize logical-time-validity metadata and temporarily maintaining multiple versions of the data. In addition, its algorithms for read-only and write-only transactions are designed to work together using indirection to ensure that clients obtain a consistent, up-to-date view of the system and can atomically update data spread across many nodes.

My dissertation research shows that ALPS systems do not need to settle for eventual consistency and weak semantics. Taken together, Eiger and COPS show that causal consistency and stronger semantics are possible for low-latency geo-replicated storage.

[1] This incompatibility is an implication of Brewer's famed CAP theorem from 2000, which was formalized shortly after by Gilbert and Lynch. Its first proof, however, was a lesser-known result from 1988 by Lipton and Sandberg.

Future Directions

Fully-Distributed General Transactions

While working on transactional algorithms for the low-latency geo-replicated setting of COPS and Eiger, I began to design algorithms for fully-distributed general transactions. General transactions are widely recognized as easy for programmers to reason about and necessary for certain scenarios, e.g., banking. Due to the fundamental trade-off between strong consistency and low latency, they are hard to reason about in—and perhaps incompatible with—the geo-replicated and low-latency setting of COPS and Eiger. They are, however, compatible with a low-latency single-datacenter setting or with a non-low-latency geo-replicated setting.

Currently, transactions are either scalable or general, but not both. My research and Google's recent work on Spanner, which provides read-once-then-write transactions, are examples of the former. On the other hand, examples of the latter include traditional databases with general transactions that are restricted to a small subset (shard) of the data and/or are scheduled by a master node. Developing and verifying fully-distributed transactions across large numbers of nodes will bridge this divide and provide scalability for general transactions.

Scalable Transport for Massively-Distributed Systems

Current transport-layer protocols, e.g., TCP and UDP, are ill-suited for massive-scale distributed systems. TCP was designed to provide a reliable stream of data between two machines; UDP was designed to provide unreliable datagrams in the same setting. In contrast, the communication patterns of massive-scale distributed systems typically have large numbers of parallel, asynchronous RPCs between many machines.

One example of this mismatch is that while running experiments for Eiger with hundreds of nodes, I found I could not achieve perfect linear scaling of throughput as cluster size increased due to the overhead from the increasing number of TCP connections on each node. As another example, I've learned from industrial colleagues that it is common practice to aggregate connections from multiple processes on the same machine to reduce connection overhead and to increase batching. Yet another example is Facebook's modification to memcached that uses TCP for writes, but uses UDP for reads so they can build their own retransmission protocol atop it that is aware of how they batch reads (multigets).

All of these issues stem from a mismatch between what current transport layers provide and how massive-scale distributed systems communicate. While there has been much recent work on improving TCP for datacenter usage—e.g., DCTCP that improves congestion control, or the entire "Data Centers: Latency" session at SIGCOMM 2012 that improved flow completion times—a more fundamental approach is necessary and possible. Datacenter services provide a rare opportunity to deploy a clean-slate transport layer because they are in a single administrative domain and can be upgraded en masse. Along with networking colleagues, I am interested in exploring what new transport layer properties, abstractions, and mechanisms can better match the performance or programmability requirements of massive-scale distributed systems.

Making Programming Distributed Storage Make Sense

Strong consistency and general transactions, while incompatible with low-latency geo-replication, are well understood by programmers and easier for them to reason about than weaker consistency and limited transactions. My dissertation research starts to bridge this gap, but there is still much to do before programming massive-scale distributed storage truly "makes sense."

This introduces two exciting avenues of research. One is, how can we make it easier for

programmers to reason about and use low-latency primitives? My current approach has been to make the primitives as strong as possible and I believe this direction will bear more fruit. But, we will also need ways to make it easier for programmers to express themselves. Perhaps a query language will help? Or, a compiler that will deconstruct general transactions into limited ones?

The other avenue of research is, how can we present a single interface that is simple for programmers to reason about that provides access to strong-but-slow general transactions and fast-but-weaker limited transactions? Should transactions and their results be typed so we can reason about their use throughout a program? Would a domain-specific language help? Can we allow programmers to write the simplest code initially and then only specialize it if dictated by performance?

Each of these avenues provides an interesting mix of programming languages and distributed systems problems. I look forward to collaborating on them with PL colleagues and I believe solving them will have a large and lasting impact on the way web services are built and the way programmers interact with big data.

In the last 20 years the field of distributed systems has changed dramatically. The field moved from systems on the order of 10s of nodes in one administrative domain, to peer-to-peer systems with 1000s of nodes in many administrative domains, to datacenter services with a small number of datacenters each with 1000s of nodes within it that are back in a single administrative domain. With billions of edge devices that are increasingly capable, I am intrigued to see how they fit into the distributed systems of the future.

Increasingly, distributed systems problems have connections to networking, databases, programming languages, algorithms, and security. I look forward to working with colleagues in these areas in my future research career, as I believe that many of the most productive types of research come from inter-area collaboration.

Teaching Statement

Wyatt Lloyd

Teaching is an important and exciting part of being a faculty member. I look forward to being involved in teaching at all levels, from sparking student interest in Computer Science in introductory courses to advising students pursuing their own research. I am qualified to teach introductory CS classes and particularly qualified to teach classes on systems and networking at the undergraduate level, the advanced/graduate level, and in seminars on more focused topics like distributed storage systems or datacenter networking.

I first gained teaching experience as a TA for Princeton's introductory course (COS 126). My favorite part of the course was teaching a twice-weekly precept to my section of about a dozen students. I focused on being enthusiastic about the material and keeping the class highly interactive. I believe the more you involve students and show them the interesting facets of a subject, the more they will be motivated to learn.

I also served as a TA for Princeton's networking course at the advanced undergraduate level (COS 461). The course had a strong focus on projects where the students built what we were learning about in class, e.g., a lightweight TCP implementation. This focus on building helped students understand the material more deeply, actively engaged them in the subject, and provided them with practical system building experience. The core parts of course were supplemented with discussions of recent research advancements, showing students our field is still evolving and that they can have an impact. This inspired several students to do research with our group, some of whom ultimately went to graduate school in systems and networking. The effectiveness of the class greatly impressed me, and I will incorporate a focus on building and discussion of research results into any advanced systems or networking course I teach.

Graduate courses offer an in-depth look at a topic and are an excellent vehicle for initiating research. I have found the format of lectures on a topic, followed by reading related papers, and then finally working on a research project to be a successful strategy. Lectures ensure a basic level of knowledge so that papers are accessible and framed by current and historical practices. Paper readings and discussions allow much more depth in exploring a topic as well as help students learn how to best focus their work and its exposition for maximum impact. This fall I had the privilege to guest teach the graduate level advanced networking class (COS 561) where I delivered a lecture with my enthusiastic and interactive style, followed by a discussion of two papers the students had read. This experience, combined with my earlier experience as a TA, confirmed that I enjoy and excel at teaching all levels of courses.

Research projects are an essential part of graduate courses because they give students the opportunity to explore a topic in greater depth, enabling them to make a potentially publishable contribution of their own. This is, in fact, how I published my first paper. An aspect of student projects that I will emphasize is building a prototype and experimentally verifying its design. I consider this an essential component of research in systems as it grounds their work in reality, gives them system-building experience, helps them focus on what is novel about their design, and can lead them to new areas of research.

I have advised one undergraduate on a semester-long research project and three other undergraduates on six-week summer projects. In each case, the experience was rewarding and I look forward to longer-term advisement of graduate students. I view advisement as a tremendous opportunity, and responsibility, to guide students down worthwhile paths without stifling their individuality or creativity. I am particularly excited to see how long-term collaboration will merge my ideas with those of my students into new research directions.

Preprint of our paper to appear in NSDI '13
This version is from Dec 14, 2012
Find the most up-to-date version at:
http://www.cs.princeton.edu/~wlloyd/papers/eiger-nsdi13.pdf

Stronger Semantics for Low-Latency Geo-Replicated Storage

Wyatt Lloyd[⋆], Michael J. Freedman[⋆], Michael Kaminsky[†], and David G. Andersen[‡]
[⋆]Princeton University, [†]Intel Labs, [‡]Carnegie Mellon University

Abstract

We present the first scalable, geo-replicated storage system that guarantees low latency, offers a rich data model, and provides "stronger" semantics. Namely, all clients requests are satisfied in the local datacenter in which they arise; the system efficiently supports useful data model abstractions such as column families and counter columns; and clients can access data in a causally-consistent fashion with read-only and write-only transactional support, even for keys spread across many servers.

The primary contributions of this work are enabling scalable causal consistency for the complex column-family data model, as well as novel, non-blocking algorithms for both read-only and write-only transactions. Our evaluation shows that our system, Eiger, achieves low (single-ms) latency and has throughput competitive with eventually-consistent and non-transactional Cassandra, upon which it is built. Despite Eiger's stronger semantics, its throughput is within 15% of Cassandra's for a large variety of workloads and within 7% for one of Facebook's real-world workloads.

1 Introduction

Large-scale data stores are a critical infrastructure component of many Internet services. In this paper, we address the problem of building a geo-replicated data store targeted at applications that demand fast response times. Such applications are now common: Amazon, EBay, and Google all claim that a slight increase in user-perceived latency translates into concrete revenue loss [20, 21, 36, 44].

Providing *low latency* to the end-user requires two properties from the underlying storage system. First, storage nodes must be near the user to avoid long-distance round trip times; thus, data must be replicated geographically to handle users from diverse locations. Second, the storage layer itself must be fast: client reads and writes must be local to that nearby datacenter and not traverse the wide area. Geo-replicated storage also provides the important benefits of availability and fault tolerance.

Beyond low latency, many services benefit from a *rich data model*. Key-value storage—perhaps the simplest data model provided by data stores—is used by a number of services today [3, 24]. The simplicity of this data model, however, makes building a number of interesting services overly arduous, particularly compared to the column-family data models offered by systems like BigTable [15] and Cassandra [32]. These rich data models provide hierarchical sorted column-families and numerical counters. Column-families are well-matched to services such as Facebook, while counter columns are particularly useful for numerical statistics, as used by collaborative filtering (Digg, Reddit), likes (Facebook), or re-tweets (Twitter).

Unfortunately, to our knowledge, no existing geo-replicated data store provides guaranteed low latency, a rich column-family data model, and *stronger consistency semantics*: consistency guarantees stronger than the weakest choice—eventual consistency—and support for atomic updates and transactions. This paper presents Eiger, a system that achieves all three properties.

The consistency model Eiger provides is tempered by impossibility results: the strongest forms of consistency—such as linearizability, sequential, and serializability—are impossible to achieve with low latency [6, 37] (that is, latency less than the network delay between datacenters). Yet, some forms of stronger-than-eventual consistency are still possible and useful, e.g., *causal consistency* [2], and they can benefit system developers and users. In addition, *read-only* and *write-only transactions* that execute a batch of read or write operations at the same logical time can strengthen the semantics provided to a programmer.

Many previous systems satisfy two of our three design goals. Traditional databases, as well as the more recent Walter [46], MDCC [30], Megastore [7], and some Cassandra configurations, provide stronger semantics and a rich data model, but cannot guarantee low latency. Redis [42], CouchDB [18], and other Cassandra configurations provide low latency and a rich data model, but not stronger semantics. Our prior work on COPS [38] supports low latency, causal consistency, and read-only transactions, but not a richer data model or write-only transactions (see §6.8 and §7 for a detailed comparison).

A key challenge of this work is to meet these three goals while *scaling* to a large numbers of nodes in a single datacenter, which acts as a single logical replica. Traditional solutions in this space [8, 10, 31], such as

Bayou [39], assume a single node per replica and rely on techniques such as log exchange to provide consistency. Log exchange, however, requires serialization through a single node, which does not scale to multi-node replicas.

This paper presents Eiger, a scalable geo-replicated data store that achieves our three goals. Like COPS, Eiger tracks dependencies to ensure consistency; instead of COPS' dependencies on versions of keys, however, Eiger tracks dependencies on operations. Yet, its mechanisms do not simply harken back to the transaction logs common to databases. Unlike those logs, Eiger's operations may depend on those executed on other nodes, and an operation may correspond to a transaction that involves keys stored on different nodes.

Eiger's read-only and write-only transaction algorithms each represent an advance in the state-of-the-art. COPS introduced a read-only transaction algorithm that normally completes in one round of local reads, and two rounds in the worst case. Eiger's read-only transaction algorithm has the same properties, but achieves them using logical time instead of explicit dependencies. Not storing explicit dependencies not only improves Eiger's efficiency, it allows Eiger to tolerate long partitions between datacenters, while COPS may suffer a metadata explosion that can degrade availability.

Eiger's write-only transaction algorithm can atomically update multiple columns of multiple keys spread across multiple servers in a datacenter (i.e., they are atomic within a datacenter, but not globally). It was designed to coexist with Eiger's read-only transactions, so that both can guarantee low-latency by (1) remaining in the local datacenter, (2) taking a small and bounded number of local messages to complete, and (3) never blocking on any other operation. In addition, both transaction algorithms are general in that they can be applied to systems with stronger consistency, e.g., linearizability [28].

The contributions of this paper are as follows:

- The design of a low-latency, causally-consistent data store based on a column-family data model, including all the intricacies necessary to offer abstractions such as column families and counter columns.
- A novel non-blocking read-only transaction algorithm that is both performant and partition tolerant.
- A novel write-only transaction algorithm that atomically writes a set of keys, is lock-free (low latency), and does not block concurrent read transactions.
- An evaluation that shows Eiger has performance competitive to eventually-consistent Cassandra.

2 Background

This section reviews background information related to Eiger: web service architectures, the column-family data model, and causal consistency.

	User Data		Associations				
			Friends			Likes	
	ID	Town	Alice	Bob	Carol	NSDI	SOSP
Alice	1337	NYC	-	3/2/11	9/2/12	9/1/12	-
Bob	2664	LA	3/2/11	-	-	-	-
⋮							

Figure 1: An example use of the column-family data model for a social network setting.

2.1 Web Service Architecture

Eiger targets large geo-replicated web services. These services run in multiple datacenters world-wide, where each datacenter stores a full replica of the data. For example, Facebook stores all user profiles, comments, friends lists, and likes at each of its datacenters [22]. Users connect to a nearby datacenter, and applications strive to handle requests entirely within that datacenter.

Inside the datacenter, client requests are served by a front-end web server. Front-ends serve requests by reading and writing data to and from storage tier nodes. Writes are asynchronously replicated to storage tiers in other datacenters to keep the replicas loosely up-to-date.

In order to scale, the storage cluster in each datacenter is typically partitioned across 10s to 1000s of machines. As a primitive example, Machine 1 might store and serve user profiles for people whose names start with 'A', Server 2 for 'B', and so on.

As a storage system, Eiger's *clients* are the front-end web servers that issue read and write operations on behalf of the human users. When we say, "a client writes a value," we mean that an application running on a web or application server writes into the storage system.

2.2 Column-Family Data Model

Eiger uses the column-family data model, which provides a rich structure that allows programmers to naturally express complex data and then efficiently query it. This data model was pioneered by Google's BigTable [15]. It is now available in the open-source Cassandra system [32], which is used by many large web services including EBay, Netflix, and Reddit.

Our implementation of Eiger is built upon Cassandra and so our description adheres to its specific data model where it and BigTable differ. Our description of the data model and API are simplified, when possible, for clarity.

Basic Data Model. The column-family data model is a "map of maps of maps" of named columns. The first-level map associates a key with a set of named column families. The second level of maps associates the column family with a set comprised exclusively of either columns or super columns. If present, the third and final level of

bool	⟵	batch_mutate	({key→mutation})
bool	⟵	atomic_mutate	({key→mutation})
{key→columns}	⟵	multiget_slice	({key, column_parent, slice_predicate})

Table 1: Core API functions in Eiger's column family data model. Eiger introduces `atomic_mutate` and converts `multiget_slice` into a read-only transaction. All calls also have an `actor_id`.

maps associates each super column with a set of columns. This model is illustrated in Figure 1: "Associations" are a column family, "Likes" are a super column, and "NSDI" is a column.

Within a column family, each *location* is represented as a compound key and a single value, i.e., "Alice:Assocs:Friends:Bob" with value "3/2/11". These pairs are stored in a simple ordered key-value store. All data for a single row must reside on the same server.

Clients use the API shown in Table 1. Clients can insert, update, or delete columns for multiple keys with a `batch_mutate` operation; each mutation is either an `insert` or a `delete`. (If a column exists, an `insert` updates the value.) Similarly, clients can read many columns for multiple keys with the `multiget_slice` operation. The client provides a list of tuples, each involving a key, a column family name and optionally a super column name, and a slice predicate. The slice predicate can be a `(start,stop,count)` three-tuple, which matches the first `count` columns with names between `start` and `stop`. Names may be any comparable type, e.g., strings or integers. Alternatively, the predicate can also be a list of column names. In either case, a *slice* is a subset of the stored columns for a given key.

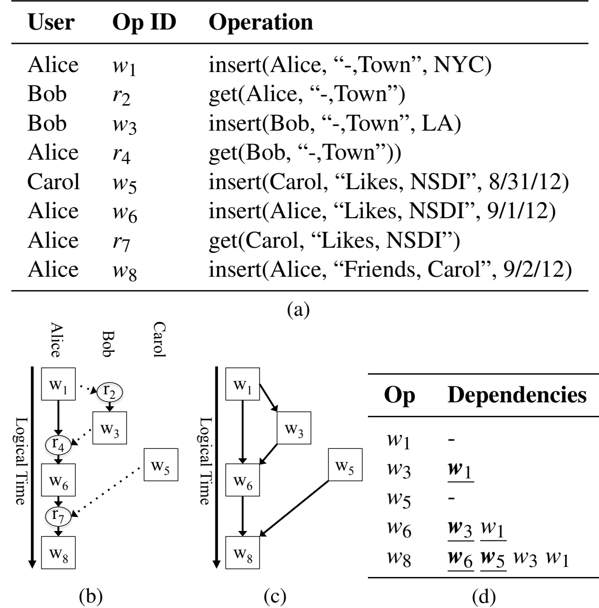
Given the example data model in Figure 1 for a social network, the following function calls show three typical API calls: updating Alice's hometown when she moves, removing Carol from her friends' lists when she quits the social network, and retrieving up to 10 of Alice's friends with names starting with B to Z.

```
batch_mutate ( Alice→insert(UserData:Town=Rome) )

batch_mutate ( Alice→delete(Assocs:Friends:Carol),
               Bob→delete(Assocs:Friends:Carol) )

multiget_slice ({Alice, Assocs:Friends, (B, Z, 10)})
```

Counter Columns. Standard columns are updated by `insert` operations that overwrite the old value. Counter columns, in contrast, can be commutatively updated using an `add` operation. They are useful for maintaining numerical statistics, e.g., a "liked_by_count" for NSDI (not shown in figure), without the need to carefully read-modify-write the object.

2.3 Causal Consistency

A rich data model alone does not provide an intuitive and useful storage system. The storage system's consistency guarantees can restrict the possible ordering and timing



User	Op ID	Operation
Alice	w_1	insert(Alice, "-,Town", NYC)
Bob	r_2	get(Alice, "-,Town")
Bob	w_3	insert(Bob, "-,Town", LA)
Alice	r_4	get(Bob, "-,Town"))
Carol	w_5	insert(Carol, "Likes, NSDI", 8/31/12)
Alice	w_6	insert(Alice, "Likes, NSDI", 9/1/12)
Alice	r_7	get(Carol, "Likes, NSDI")
Alice	w_8	insert(Alice, "Friends, Carol", 9/2/12)

(a)

Op	Dependencies
w_1	-
w_3	$\underline{\mathbf{w_1}}$
w_5	-
w_6	$\underline{\mathbf{w_3}}$ $\underline{w_1}$
w_8	$\underline{\mathbf{w_6}}$ $\underline{\mathbf{w_5}}$ w_3 w_1

(b) (c) (d)

Figure 2: (a) A set of example operations; (b) the graph of causality between them; (c) the corresponding dependency graph; and (d) a table listing nearest (bold), one-hop (underlined), and all dependencies.

of operations throughout the system, helping to simplify the possible behaviors that a programmer must reason about and the anomalies that clients may see.

The strongest forms of consistency (linearizability, serializability, and sequential consistency) are provably incompatible with our low-latency requirement [6, 37], and the weakest (eventual consistency) allows many possible orderings and anomalies. For example, under eventual consistency, after Alice updates her profile, she might not see that update after a refresh. Or, if Alice and Bob are commenting back-and-forth on a blog post, Carol might see a random non-contiguous subset of that conversation.

Fortunately, *causal consistency* can avoid many such inconvenient orderings, while guaranteeing low latency. Causal consistency provides a partial order over operations in the system according to the notion of potential causality [2, 33], which is defined by three rules:

- **Thread-of-Execution.** An operation performed by a thread is causally after all of its previous ones.
- **Reads-From.** An operation that reads a value is causally after the operation that wrote the value.
- **Transitive-Closure.** If operation *a* is causally after *b*, and *b* is causally after *c*, then *a* is causally after *c*.

Figure 2 shows several example operations and illustrates their causal relationships. Arrows indicate the sink is causally after the source.

Write operations have *dependencies* on all other write operations that they are causally after. Eiger uses these dependencies to enforce causal consistency: It does not apply (commit) a write in a cluster until verifying that the operation's dependencies are *satisfied*, meaning those writes have already been applied in the cluster.

While the number of dependencies for a write grows with a client's lifetime, the system does not need to track every dependency. Rather, only a small subset of these, the *nearest dependencies*, are necessary for ensuring causal consistency. These dependencies, which have a longest path of one hop to the current operation, transitively capture all of the ordering constraints on this operation. In particular, because all non-nearest dependencies are depended upon by at least one of the nearest, if this current operation occurs after the nearest dependencies, then it will occur after all non-nearest as well (by transitivity). Eiger actually tracks *one-hop dependencies*, a slightly larger superset of nearest dependencies, which have a shortest path of one hop to the current operation. The motivation behind tracking one-hop dependencies is discussed in Section 3.2. Figure 2(d) illustrates the types of dependencies, e.g., w_6's dependency on w_1 is one-hop but not nearest.

3 Eiger System Design

In our design and discussion of Eiger, we make three simplifying assumptions. First, we assume each datacenter's keyspace is partitioned across the servers [4, 25, 29]. Second, we assume that datacenters may fail, but if a datacenter is available, then so are all "logical" servers within it. Within a datacenter, each key is stored on one "logical" server, implemented using a replicated state machine protocol between a set of servers (e.g., chain replication [50] or Paxos [34]). Finally, we assume that operations within each datacenter are linearizable, which is amenable to sufficiently low latency because it is in the local area.

3.1 Achieving Causal Consistency

Eiger provides causal consistency by explicitly checking that an operation's nearest dependencies have been applied before applying the operation. This approach is similar to the mechanism used by COPS [38], although COPS places dependencies on values, while Eiger uses dependencies on *operations*.

Tracking dependencies on operations significantly improves Eiger's efficiency. In the column family data model, it is not uncommon to simultaneously read or write many columns for a single key. With dependencies on values, a separate dependency and dependency check must be used for each column's value and thus would check $|column|$ dependencies; Eiger could check as few as one. In the worst case, when all columns were written by different operations, the number of required dependency checks degrades to one per value.

Dependencies in Eiger consist of a locator and a unique id. The *locator* is used to ensure that any other operation that depends on this operation knows which node to check with to determine if the operation has been committed. For mutations of individual keys, the locator is simply the key itself. Within an atomic write group, the locator can be any key in the set; all that matters is that each "sub-operation" within an atomic write be labeled with the same locator.

The *unique id* allows dependencies to precisely map to operations. A node in Eiger checks dependencies by sending a `dep_check` operation to the node in its local datacenter that owns the locator. The node that owns the locator checks local data structures to see if has applied the operation identified by its unique id. If it has, it responds immediately. If not, it blocks the `dep_check` until it applies the operation. Thus, once all `dep_check`s return, a server knows all causally previous operations have been applied and it can safely apply this operation.

3.2 Client Library

Clients access their local Eiger datacenter using a client library that: (1) mediates access to nodes in the local datacenter; (2) executes the read and write transaction algorithms, and, most importantly; (3) tracks causality so it can attach dependencies to write operations.[1]

The client library mediates access to the local datacenter by maintaining a view of its live servers and the partitioning of its keyspace. The library uses this information to send operations to the appropriate servers and sometimes to split operations that span multiple servers.

The client library tracks causality by observing a client's operations and their results. The API exposed by the client library matches that shown earlier in Table 1 with the addition of a `actor_id` field. The `actor_id` field allows the library to distinguish between operations issued on behalf of different application-level actors (e.g., operations issued on behalf of Alice are not entangled with operations issued on behalf of Bob).

When a client issues a write, the library attaches dependencies on its previous write and on all the writes that wrote a value this client has observed through reads since then. This *one-hop* set of dependencies is the set of

[1] Our implementation of Eiger, like COPS before it, places the client library with the storage system client—typically a web server. Alternative implementations might store the dependencies on a unique node per client, or even push dependency tracking to a rich javascript application running in the client web browser itself, in order to successfully track web accesses through different servers. Such a design is compatible with Eiger, and we view it as worthwhile future work.

operations that have a path of length one to the current operation in the causality graph. The one-hop dependencies are a superset of the nearest dependencies (which have a longest path of length one) and thus attaching and checking them suffices for providing causal consistency.

We elect to track one-hop dependencies because we can do so without storing any dependency information at the servers. Using one-hop dependencies slightly increases both the amount of memory needed at the client nodes and the data sent to servers on writes.[2]

3.3 Basic Operations

Eiger's basic operations closely resemble Cassandra, upon which it is built. The main differences involve the use of server-supplied logical timestamps instead of client-supplied real-time timestamps and, as described above, the use of dependencies and `dep_checks`.

Logical Time. Clients and servers in Eiger maintain a logical clock [33] and messages include a logical timestamp that updates these clocks. The clocks and timestamps provide a progressing logical time throughout the entire system. The low-order bits in each timestamps are set to the stamping server's unique identifier, so each is globally distinct. Servers use these logical timestamps to uniquely identify and order operations.

Local Write Operations. All three write operations in Eiger—`insert`, `add`, and `delete`—operate by replacing the current (potentially non-existent) column in a location. `insert` overwrites the current value with a new column, e.g., update Alice's home town from NYC to MIA. `add` merges the current column with the update, e.g., increment a liked-by count from 8 to 9. `delete` overwrites the current column with a tombstone, e.g., Carol is no longer friends with Alice. When each new column is written, it is *timestamped* with the current logical time at the server applying the write.

Cassandra atomically applies updates to a single row using snap trees [11], so all updates to a single key in a `batch_mutate` have the same timestamp. Updates to different rows on the same server in a `batch_mutate` will have different timestamps because they are applied at different logical times.

Read Operations. Read operations return the current column for each requested location. Normal columns return binary data. Deleted columns return an empty column with a deleted bit set. The client library strips deleted columns out of the returned results, but records dependencies on them as required for correctness. Counter columns return a 64-bit integer.

[2]In contrast, our alternative design for tracking the (slightly smaller set of) nearest dependencies put the dependency storage burden on the servers, a trade-off we did not believe generally worthwhile.

Replication. Servers replicate write operations to their *equivalent* servers in other datacenters. These are the servers that own the same portions of the keyspace as the local server. Because the keyspace partitioning may vary from datacenter to datacenter, the replicating server must sometimes split `batch_mutate` operations.

When a remote server receives a replicated `add` operation it applies it normally, merging its update with the current value. When a server receives a replicated `insert` or `delete` operation, it compares the timestamps for each included column against the current column for each location. If the replicated column is logically newer, it overwrites the current column in the same way a local write would, except it keeps the original timestamp that uniquely identifies the operation that wrote the value. If the replicated column is older, it is discarded. This simple procedure ensures causal consistency: If one column is causally after the other, it will have a later timestamp and thus overwrite the other.

The overwrite procedure also implicitly handles *conflicting operations* that concurrently update a location. It applies the *last-writer-wins rule* [48] to deterministically allow the later of the updates to overwrite the other. This ensures that all datacenters converge to the same value for each column. Eiger could detect conflicts using previous pointers and then resolve them with application-specific functions similar to COPS, but we did not implement such conflict handling and omit details for brevity.

Counter Columns. The commutative nature of counter columns complicates tracking dependencies. In normal columns with overwrite semantics, each value was written by exactly one operation. In counter columns, each value was affected by many operations. Consider a counter with value 7 from +1, +2, and +4 operations. Each operation contributed to the final value, so a read of the counter incurs dependencies on all three. Eiger stores these dependencies with the counter and returns them to the client, so they can be attached to its next write.

Naively, every update of a counter column would increment the number of dependencies contained by that column *ad infinitum*. To bound the number of contained dependencies, Eiger structures the `add` operations occurring within a datacenter. Recall that all locally originating `add` operations within a datacenter are already ordered because the datacenter is linearizable. Eiger explicitly tracks this ordering in a new `add` by adding an *extra dependency* on the previously accepted `add` operation from the datacenter. This creates a single dependency chain that transitively covers all previous updates from the datacenter. As a result, each counter column contains at most one dependency per datacenter. Due to space constraints, we omit the description of an optimization that further reduces the number of dependencies to the nearest dependencies within that counter column.

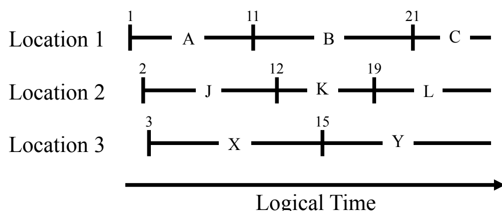


Figure 3: Validity periods for values written to different locations. Crossbars (and the specified numeric times) correspond to the earliest and latest valid time for values, which are represented by letters.

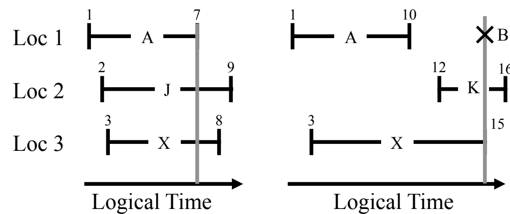


Figure 4: Examples of read-only transactions. The effective time of each transaction is shown with a gray line; this is the time requested for location 1 in the second round in (b).

4 Read-Only Transactions

Read-only transactions—the only read operations in Eiger—enable clients to see a consistent view of multiple keys that may be spread across many servers in the local datacenter. Eiger's algorithm guarantees low latency because it takes at most two rounds of parallel non-blocking reads in the local datacenter, plus at most one additional round of local non-blocking checks during concurrent write transactions, detailed in §5.4. We make the same assumptions about reliability in the local datacenter as before, namely, "logical" servers do not fail due to linearizable state machine replication.

Why read-only transactions? Even though Eiger tracks dependencies to update each datacenter consistently, non-transactional reads can still return an inconsistent set of values. For example, consider a scenario where two items were *written* in a causal order, but read via two separate, parallel reads. The two reads could bridge the write operations (one occurring before either write, the other occurring after both), and thus return values that never actually occurred together, e.g., a "new" object and its "old" access control metadata.

4.1 Read-only Transaction Algorithm

The key insight in the algorithm is that *there exists a consistent result for every query at every logical time*. Figure 3 illustrates this: As operations are applied in a consistent causal order, every data location (key and column) has a consistent value at each logical time.

At a high level, our new read transaction algorithm marks each data location with validity metadata, and uses that metadata to determine if a first round of optimistic reads is consistent. If the first round results are not consistent, the algorithm issues a second round of reads that are guaranteed to return consistent results.

More specifically, each data location is marked with an *earliest valid time* (EVT). The EVT is set to the server's logical time when it locally applies an operation that writes a value. Thus, in an operation's accepting datacenter—the one at which the operation originated—the EVT is the same as its timestamp. In other datacenters, the EVT is later than its timestamp. In both cases, the EVT is the exact logical time when the value became visible in the local datacenter.

A server responds to a read with its currently visible value, the corresponding EVT, and its current logical time, which we call the *latest valid time* (LVT). Because this value is still visible, we know it is valid for at least the interval between the EVT and LVT. Once all first-round reads return, the client library compares their times to check for consistency. In particular, it knows all values were valid at the same logical time (i.e., correspond to a consistent snapshot) iff the maximum EVT $\leq$ the minimum LVT. If so, the client library returns these results; otherwise, it proceeds to a second round. Figure 4(a) shows a scenario that completes in one round.

The *effective time* of the transaction is the minimum LVT $\geq$ the maximum EVT. It corresponds both to a logical time in which all retrieved values are consistent, as well as the current logical time (as of its response) at a server. As such, it ensures freshness—necessary in causal consistency so that clients always see a progressing datacenter that reflects their own updates.

For brevity, we only sketch a proof that read transactions return the set of results that were visible in their local datacenter at the transaction's effective time, EffT. By construction, assume a value is visible at logical time t iff val.EVT $\leq t \leq$ val.LVT. For each returned value, if it is returned from the first round, then val.EVT $\leq$ maxEVT $\leq$ EffT by definition of maxEVT and EffT, and val.LVT $\geq$ EffT because it is not being requested in the second round. Thus, val.EVT $\leq$ EffT $\leq$ val.LVT, and by our assumption, the value was visible at EffT. If a result is from the second round, then it was obtained by a second-round read that explicitly returns the visible value at time EffT, described next.

4.2 Two-Round Read Protocol

A read transaction requires a second round if there does not exist a single logical time for which *all* values read in the first round are valid. This can only occur when there are concurrent updates being applied locally to the requested locations. The example in Figure 4(b) requires a second round because location 2 is updated to value K

```
function read_only_trans(requests):
  # Send first round requests in parallel
  for r in requests
    val[r] = multiget_slice(r)
  # Calculate the maximum EVT
  maxEVT = 0
  for r in requests
    maxEVT = max(maxEVT, val[r].EVT)
  # Calculate effective time
  EffT = ∞
  for r in requests
    if val[r].LVT ≥ maxEVT
      EffT = min(EffT, val[r].LVT)
  # Send second round requests in parallel
  for r in requests
    if val[r].LVT < EffT
       val[r] = multiget_slice_by_time(r, EffT)
  # Return only the requested data
  return extract_keys_to_columns(res)
```

Figure 5: Pseudocode for read-only transactions.

at time 12, which is not before time 10 when location 1's server returns value A.

During the second round, the client library issues `multiget_slice_by_time` requests, specifying a read at the transaction's effective time. These reads are sent only to those locations for which it does not have a valid result, i.e., their LVT is earlier than the effective time.

Servers respond to `multiget_slice_by_time` reads with the value that was valid at the requested logical time. Because that result may be different than the currently visible one, servers sometimes must store old values for each location. Fortunately, the extent of such additional storage can be significantly limited.

4.3 Limiting Old Value Storage

Eiger limits the need to store old values in two ways. First, read transactions have a timeout that specifies its maximum real-time duration. If this timeout fires—which happens only when server queues grow pathologically long due to prolonged overload—the client library restarts a fresh read transaction. Thus, servers only need to store old values that have been overwritten within this timeout's duration.

Second, Eiger only retains old values that could be requested by a `multiget_slice_by_time`. Thus, servers store old values only for keys that had been read within the timeout duration, and only those values that are *newer* than those returned in a first round. For this optimization, Eiger stores the last access time of each value.

4.4 Read Transactions for Linearizability

Linearizability (strong consistency) is attractive to programmers when low latency and availability are not strict requirements. Simply being linearizable, however, does not mean that a system is transactional: There may be no way to extract a mutually consistent set of values from the system, much as in our earlier example for read transactions. Linearizability is only defined on, and used with, operations that read or write a single location (originally, shared memory systems) [28].

Interestingly, our algorithm for read-only transactions works for fully linearizable systems, *without* modification. In Eiger, in fact, if all writes that are concurrent with a read-only transaction originated from the local datacenter, the read-only transaction provides a consistent view of that linearizable system (the local datacenter).

5 Write-Only Transactions

Eiger's write-only transactions allow a client to atomically write many columns spread across many keys in the local datacenter. These values also appear atomically in remote datacenters upon replication. As we will see, the algorithm guarantees low latency because it takes at most 2.5 message RTTs in the *local* datacenter to complete, no operations acquire locks, and all phases wait on only the previous round of messages before continuing.

Write-only transactions have many uses. When a user presses a save button, the system can ensure that all of her five profile updates appear simultaneously. Similarly, they help maintain symmetric relationships in social networks: When Alice accepts Bob's friendship request, both friend associations appear at the same time.

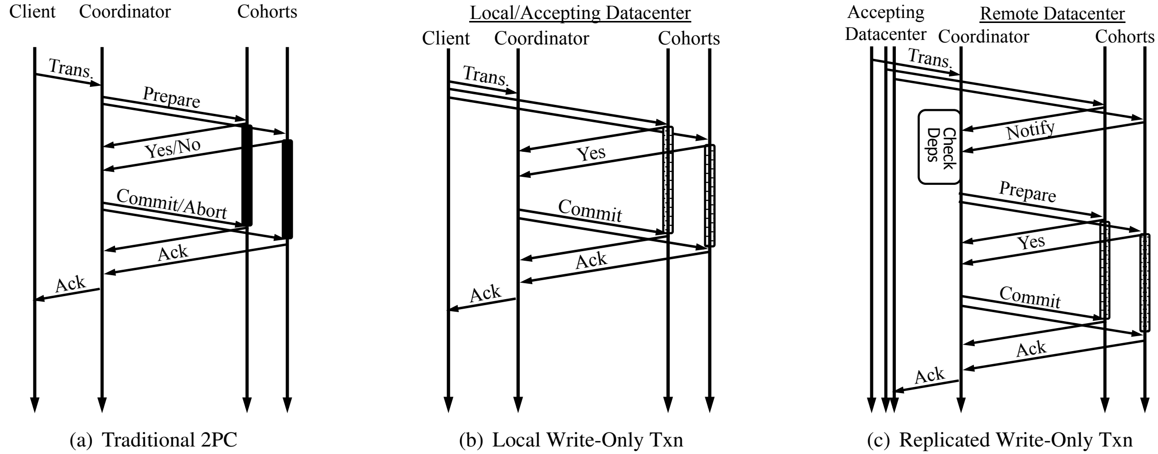
5.1 Write-Only Transaction Algorithm

To execute an `atomic_mutate` request—which has identical arguments to `batch_mutate`—the client library splits the operation into one sub-request per local server across which the transaction is spread. The library randomly chooses one key in the transaction as the *coordinator key*. It then transmits each sub-request to its corresponding server, annotated with the coordinator key.

Our write transaction is a variant of two-phase commit [45], which we call *two-phase commit with positive cohorts and indirection* (2PC-PCI). 2PC-PCI operates differently depending on whether it is executing in the original (or "accepting") datacenter, or being applied in the remote datacenter after replication.

There are three differences between traditional 2PC and 2PC-PCI, as shown in Figure 6. First, 2PC-PCI has only positive cohorts; the coordinator always commits the transaction once it receives a vote from all cohorts.[3] Second, 2PC-PCI has a different pre-vote phase that varies depending on the origin of the write transaction. In

[3]Eiger only has positive cohorts because it avoids all the normal reasons to abort (vote no): It does not have general transactions that can force each other to abort, it does not have users that can cancel operations, and it assumes that its "logical" servers do not fail.



(a) Traditional 2PC (b) Local Write-Only Txn (c) Replicated Write-Only Txn

Figure 6: Message flow diagrams for traditional 2PC and write-only transaction. Solid boxes denote when cohorts block reads. Striped boxes denote when cohorts will indirect a commitment check to the coordinator.

the accepting datacenter (we discuss the remote below), the client library sends each participant its sub-request directly, and this transmission serves as an implicit PREPARE message for each cohort. Third, 2PC-PCI cohorts that cannot answer a query—because they have voted but have not yet received the commit—ask the coordinator if the transaction is committed, effectively *indirecting* the request through the coordinator.

5.2 Local Write-Only Transactions

When a *participant* server, which is either the coordinator or a cohort, receives its transaction sub-request from the client, it prepares for the transaction by writing each included location with a special "pending" value (retaining old versions for second-round reads). It then sends a YESVOTE to the coordinator.

When the coordinator receives a YESVOTE, it updates its count of prepared keys. Once all keys are prepared, the coordinator commits the transaction. The coordinator's current logical time serves as the (global) timestamp and (local) EVT of the transaction and is included in the COMMIT message.

When a cohort receives a COMMIT, it replaces the "pending" columns with the update's real values, and ACKs the committed keys. Upon receiving all ACKs, the coordinator safely cleans up its transaction state.

5.3 Replicated Write-Only Transactions

Each transaction sub-request is replicated to its "equivalent" participant(s) in the remote datacenter, possibly splitting the sub-requests to match the remote key partitioning. When a cohort in a remote datacenter receives a sub-request, it sends a NOTIFY with the key count to the transaction coordinator in its datacenter. This coordinator issues any necessary `dep_check`s upon receiving its own sub-request (which contains the coordinator key). The coordinator's checks cover the entire transaction, so cohorts send no checks. Once the coordinator has received all NOTIFY messages and `dep_check`s responses, it sends each cohort a PREPARE, and then proceeds normally.

For reads received during the *indirection window* in which participants are uncertain about the status of a transaction, cohorts must query the coordinator for its state. To minimize the duration of this window, before preparing, the coordinator waits for (1) *all* participants to NOTIFY and (2) all `dep_check`s to return. This helps prevent a slow replica from causing needless indirection.

Finally, replicated write-only transactions differ in that participants do not always write pending columns. If a location's current value has a newer timestamp than that of the transaction, the validity interval for the transaction's value is empty. Thus, no read will ever return it and it can be safely discarded. The participant continues in the transaction for simplicity, but does not need to indirect reads for this location.

5.4 Reads when Transactions are Pending

If a first-round read accesses a location that could be modified by a pending transaction, the server sends a special empty response that only includes a LVT (i.e., its current time). This alerts the client that it must choose an effective time for the transaction and send the server a second-round `multiget_slice_by_time` request.

When a server with pending transactions receives a `multiget_slice_by_time` request, it first traverses its old versions for each included column. If there exists a version valid at the requested time, the server returns it.

Otherwise, there are pending transactions whose *potential commit window* intersects the requested time and the server must resolve their ordering. It does so by sending a `commit_check` with this requested time to the transactions' coordinator(s). Each coordinator responds

whether the transaction had been committed at that (past) time and, if so, its commit time.

Once a server has collected all `commit_check` responses, it updates the validity intervals of all versions of all relevant locations, up to at least the requested (effective) time. Then, it can respond to the `multiget_slice_by_time` message as normal.

The complementary nature of Eiger's transactional algorithms enable the atomicity of its writes. In particular, the single commit time for a write transaction (EVT) and the single effective time for a read transaction lead each to appear at a single logical time, while its two-phase commit ensures all-or-nothing semantics.

6 Evaluation

This evaluation explores the overhead of Eiger's stronger semantics compared to eventually-consistent Cassandra.

6.1 Implementation

Our Eiger prototype implements everything described in the paper as 5000 lines of Java added to and modifying the existing 75000 LoC in Cassandra 1.1 [13, 32]. All of Eiger's reads are transactional. We use Cassandra configured for wide-area eventual consistency as a baseline for comparison. In each local cluster, both Eiger and Cassandra use consistent hashing to map each key to a single server, and thus trivially provide linearizability.

In unmodified Cassandra, for a single logical request, the client sends all of its sub-requests to a single server. This server splits `batch_mutate` and `multiget_slice` operations from the client that span multiple servers, sends them to the appropriate server, and re-assembles the responses for the client. In Eiger, the client library handles this splitting, routing, and re-assembly directly, allowing Eiger to save a local RTT in latency and potentially many messages between servers. With this change, Eiger outperforms unmodified Cassandra in most settings. Therefore, to make our comparison to Cassandra fair, we implemented an analogous client library that handles the splitting, routing, and re-assembly for Cassandra. The results below use this optimization.

6.2 Eiger Overheads

We first examine the overhead of Eiger's causal consistency, read-only transactions, and write-only transactions. This section explains why each potential source of overhead does not significantly impair throughput, latency, or storage; the next sections confirm empirically.

Causal Consistency Overheads. Write operations carry *dependency metadata*. Its impact on throughput and latency is low because each dependency is 16B; the number of dependencies attached to a write is limited to its small set of one-hop dependencies; and writes are typically less frequent. Dependencies have no storage cost because they are not stored at the server.

Dependency check operations are issued in remote datacenters upon receiving a replicated write. Limiting these checks to the write's one-hop dependencies minimizes throughput degradation. They do not affect client-perceived latency, occuring only during asynchronous replication, nor do they add storage overhead.

Read-only Transaction Overheads. *Validity-interval metadata* is stored on servers and returned to clients with read operations. Its effect is similarly small: Only the 8B EVT is stored, and the 16B of metadata returned to the client is tiny compared to typical key/column/value sets.

If *second-round reads* were always needed, they would roughly double latency and halve throughput. Fortunately, they occur only when there are concurrent writes to the requested columns in the local datacenter, which is rare given the short duration of reads and writes.

Extra-version storage is needed at servers to handle second-round reads. It has no impact on throughput or latency, and its storage footprint is small because we aggressively limit the number of old versions (see §4.3).

Write-only Transaction Overheads. Write transactions *write columns twice*: once to mark them pending and once to write the true value. This accounts for about half of the moderate overhead of write transactions, evaluated in §6.5. When only some writes are transactional and when the writes are a minority of system operations (as found in prior studies [5, 23]), this overhead has a small effect on overall throughput. The second write overwrites the first, consuming no space.

Many *2PC-PCI messages* are needed for the write-only algorithm. These messages add 1.5 local RTTs to latency, but have little effect on throughput: the messages are small and can be handled in parallel with other steps in different write transactions.

Indirected second-round reads add an extra local RTT to latency and reduce read throughput vs. normal second-round reads. They affect throughput minimally, however, because they occur rarely: only when the second-round read arrives when there is a not-yet-committed write-only transaction on an overlapping set of columns that prepared before the read-only transaction's effective time.

6.3 Experimental Setup

Experiments use the shared VICCI testbed [40, 51], which provides users with Linux VServer instances. Each physical machine has 2x6 core Intel Xeon X5650 CPUs, 48GB RAM, 3x1TB HDDs, and 2x1GigE network ports.

All experiments are between multiple VICCI sites. The latency micro-benchmark uses a minimal wide-area setup with a cluster of 2 machines at the Princeton, Stanford, and University of Washington (UW) VICCI sites.

	Latency (ms)			
	50%	90%	95%	99%
Reads				
Cassandra-Eventual	0.38	0.56	0.61	1.13
Eiger 1 Round	0.47	0.67	0.70	1.27
Eiger 2 Round	0.68	0.94	1.04	1.85
Eiger Indirected	0.78	1.11	1.18	2.28
Cassandra-Strong-A	85.21	85.72	85.96	86.77
Cassandra-Strong-B	21.89	22.28	22.39	22.92
Writes				
Cassandra-Eventual Cassandra-Strong-A	0.42	0.63	0.91	1.67
Eiger Normal	0.45	0.67	0.75	1.92
Eiger Normal (2)	0.51	0.79	1.38	4.05
Eiger Transaction (2)	0.73	2.28	2.94	4.39
Cassandra-Strong-B	21.65	21.85	21.93	22.29

Table 2: Latency micro-benchmarks.

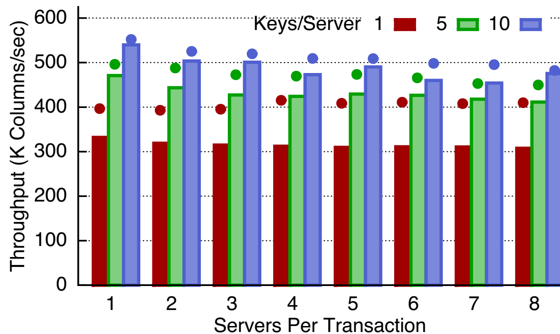


Figure 7: Throughput of an 8-server cluster for write transactions spread across 1 to 8 servers, with 1, 5, or 10 keys written per server. The dot above each bar shows the throughput of a similarly-structured eventually-consistent Cassandra write.

All other experiments use 8-machine clusters in Stanford and UW and an additional 8 machines in Stanford as clients. These clients fully load their local cluster, which replicates its data to the other cluster.

The inter-site latencies were 88ms between Princeton and Stanford, 84ms between Princeton and UW, and 20ms between Stanford and UW.

Every datapoint in the evaluation represents the median of 5+ trials. Latency micro-benchmark trials are 30s, while all other trials are 60s. We elide the first and last quarter of each trial to avoid experimental artifacts.

6.4 Latency Micro-benchmark

Eiger always satisfies client operations within a local datacenter and thus, fundamentally, is low-latency. To demonstrate this, verify our implementation, and compare with strongly-consistent systems, we ran an experiment to compare the latency of read and write operations in Eiger vs. three Cassandra configurations: eventual (R=1, W=1), strong-A (R=3, W=1), and strong-B (R=2, W=2), where R and W indicate the number of datacenters involved in reads and writes.[4]

The experiments were run from UW with a single client thread to isolate latency differences. Table 2 reports the median, 90%, 95%, and 99% latencies from operations on a single 1B column. For comparison, two 1B columns, stored on different servers, were also updated together as part of transactional and non-transactional "Eiger (2)" write operations.

All reads in Eiger—one-round, two-round, and worst-case two-round-and-indirected reads—have median latencies under 1ms and 99% latencies under 2.5ms. `atomic_mutate` operations are slightly slower than `batch_mutate` operations, but still have median latency under 1ms and 99% under 5ms. Cassandra's strongly consistent operations fared much worse. Configuration "A" achieved fast writes, but reads had to access all datacenters (including the ~84ms RTT between UW and Princeton); "B" suffered wide-area latency for both reads and writes (as the second datacenter needed for a quorum involved a ~20ms RTT between UW and Stanford).

[4]Cassandra single-key writes are not atomic across different nodes, so its strong consistency requires read repair (write-back) and $R>N/2$.

6.5 Write Transaction Cost

Figure 7 shows the throughput of write-only transactions, and Cassandra's non-atomic batch mutates, when the keys they touch are spread across 1 to 8 servers. The experiment used the default parameter settings from Table 3 with 100% writes and 100% write transactions.

Eiger's throughput remains competitive with batch mutates as the transaction is spread across more servers. Additional servers only increase 2PC-PCI costs, which account for less than 10% of Eiger's overhead. About half of the overhead of write-only transactions comes from double-writing columns; most of the remainder is due to extra metadata. Both absolute and Cassandra-relative throughput increase with the number of keys written per server, as the coordination overhead remains independent of the number of columns.

6.6 Dynamic Workloads

We created a dynamic workload generator to explore the space of possible workloads. Table 3 shows the range and default value of the generator's parameters. The results from varying each parameter while the others remain at their defaults are shown in Figure 8.

Space constraints permit only a brief review of these results. Overhead decreases with increasing value size, because metadata represents a smaller portion of message size. Overhead is relatively constant with increases in the columns/read, columns/write, keys/read, and keys/write ratios because while the amount of metadata increases,

Parameter	Range	Default	Facebook 50%	Facebook 90%	Facebook 99%
Value Size (B)	1-4K	128	16	32	4K
Cols/Key for Reads	1-32	5	1	2	128
Cols/Key for Writes	1-32	5	1	2	128
Keys/Read	1-32	5	1	16	128
Keys/Write	1-32	5		1	
Write Fraction	0-1.0	.1		.002	
Write Txn Fraction	0-1.0	.5		0 or 1.0	
Read Txn Fraction	1.0	1.0		1.0	

Table 3: Dynamic workload generator parameters. Range is the space covered in the experiments; Facebook describes the distribution for that workload.

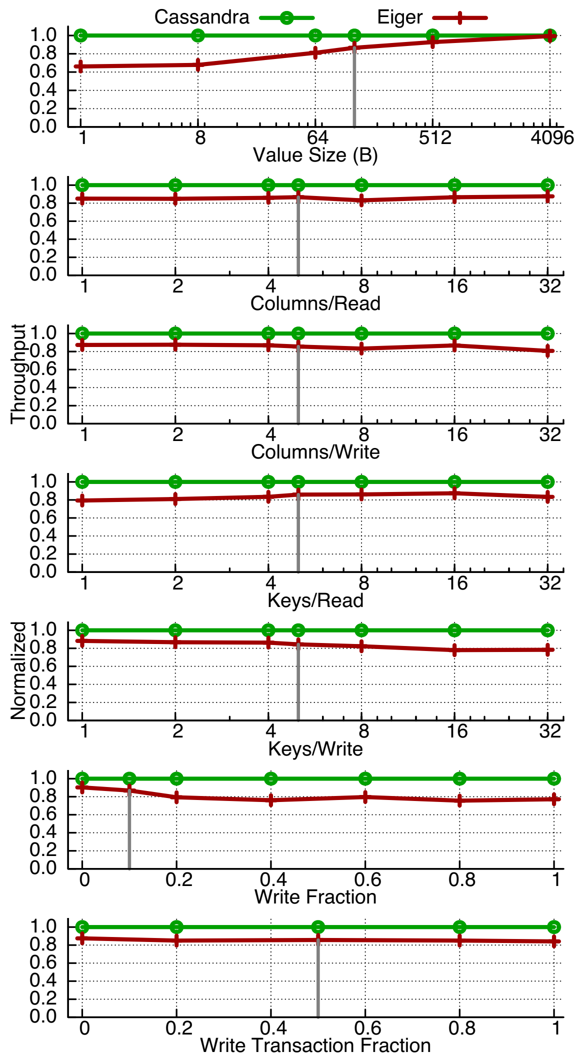


Figure 8: Results from exploring our dynamic-workload generator's parameter space. Each experiment varies one parameter while keeping all others at their default value (indicated by the vertical line). Eiger's throughput is normalized against eventually-consistent Cassandra.

	Ops/sec	Keys/sec	Columns/sec
Cassandra	23,657	94,502	49,8239
Eiger	22,088	88,238	46,6844
Eiger All Txns	22,891	91,439	48,0904
Max Overhead	6.6%	6.6%	6.3%

Table 4: Throughput for the Facebook workload.

it remains in proportion to message size. Higher fractions of write transactions (within an overall 10% write workload) do not increase overhead.

Eiger's throughput is overall competitive with the eventually-consistent Cassandra baseline. With the default parameters, its overhead is 15%. When they are varied, its overhead ranges from 0.5% to 25%.

6.7 Facebook Workload

For one realistic view of Eiger's overhead, we parameterized a synthetic workload based upon Facebook's production TAO system [47]. Parameters for value sizes, columns/key, and keys/operation are chosen from discrete distributions measured by the TAO team. We show results with a 0% write transaction fraction (the actual workload, because TAO lacks transactions), and with 100% write transactions. Table 3 shows the heavy-tailed distributions' 50th, 90th, and 99th percentiles.

Table 4 shows that the throughput for Eiger is within 7% of eventually-consistent Cassandra. The results for 0% and 100% write transactions are effectively identical because writes are such a small part of the workload. For this real-world workload, Eiger's causal consistency and stronger semantics do not impose significant overhead.

6.8 Performance vs. COPS

COPS and Eiger provide different data models and are implemented in different languages, so a direct empirical comparison is not meaningful. We can, however, intuit how Eiger's algorithms perform in the COPS setting.

Both COPS and Eiger achieve low latency around 1ms. Second-round reads would occur in COPS and Eiger equally often, because both are triggered by the same scenario: concurrent writes in the local datacenter to the same keys. Eiger experiences some additional latency when second-round reads are indirected, but this is rare (and the total latency remains low). Write-only transactions in Eiger would have higher latency than their non-atomic counterparts in COPS, but we have also shown their latency to be very low.

Beyond having write transactions, which COPS did not, the most significant difference between Eiger and COPS is the efficiency of read transactions. COPS's read transactions ("COPS-GT") add significant dependency-tracking overhead vs. the COPS baseline under certain conditions. In contrast, by tracking only one-hop dependencies, Eiger avoids the metadata explosion that

	COPS	COPS-GT	Eiger
Data Model	Key Value	Key Value	**Column Fam**
Consistency	**Causal**	**Causal**	**Causal**
Read-Only Txn	No	**Yes**	**Yes**
Write-Only Txn	No	No	**Yes**
Txn Algos Use	-	Deps	**Logic. Time**
Deps On	Values	Values	**Operations**
Transmitted Deps	**One-Hop**	All-GarbageC	**One-Hop**
Checked Deps	One-Hop	**Nearest**	One-Hop
Stored Deps	**None**	All-GarbageC	**None**
GarbageC Deps	**Unneeded**	Yes	**Unneeded**
Versions Stored	**One**	Few	**Fewer**

Table 5: Comparing COPS and Eiger.

COPS' read-only transactions can suffer. We expect that Eiger's read transactions would operate roughly as quickly as COPS' non-transactional reads, and the system as a whole would outperform COPS-GT despite offering both read- and write-only transactions and supporting a much more rich data model.

7 Related Work

A large body of research exists about stronger consistency in the wide area. This includes classical research about two-phase commit protocols [45] and distributed consensus (e.g., Paxos [34]). As noted earlier, protocols and systems that provide the strongest forms of consistency are provably incompatible with low latency [6, 37]. Recent examples includes Megastore [7], Spanner [17], and Scatter [26], which use Paxos in the wide-area; PNUTS [16], which provides sequential consistency on a per-key basis and must execute in a key's specified primary datacenter; and Gemini [35], which provides RedBlue consistency with low latency for its blue operations, but high latency for its globally-serialized red operations. In contrast, Eiger guarantees low latency.

Many previous system designs have recognized the utility of causal consistency, including Bayou [39], lazy replication [31], ISIS [10], causal memory [2], and PRACTI [8]. All of these systems require single-machine replicas (datacenters) and thus are not scalable.

Our previous work, COPS [38], bears the closest similarity to Eiger, as it also uses dependencies to provide causal consistency, and targets low-latency and scalable settings. As we show by comparing these systems in Table 5, however, Eiger represents a large step forward from COPS. In particular, Eiger supports a richer data model, has more powerful transaction support (whose algorithms also work with other consistency models), transmits and stores fewer dependencies, eliminates the need for garbage collection, stores fewer old versions, and is not susceptible to availability problems from metadata explosion when datacenters either fail, are partitioned, or suffer meaningful slow-down for long periods of time.

The database community has long supported consistency across multiple keys through general transactions. In many commercial database systems, a single primary executes transactions across keys, then lazily sends its transaction log to other replicas, potentially over the wide-area. In scale-out designs involving data partitioning (or "sharding"), these transactions are typically limited to keys residing on the same server. Eiger does not have this restriction. More fundamentally, the single primary approach inhibits low-latency, as write operations must be executed in the primary's datacenter.

Several recent systems reduce the inter-datacenter communication needed to provide general transactions. These include Calvin [49], Granola [19], MDCC [30], Orleans [12], and Walter [46]. In their pursuit of general transactions, however, these systems all choose consistency models that cannot guarantee low-latency operations. MDCC and Orleans acknowledge this with options to receive fast-but-potentially-incorrect responses.

The implementers of Sinfonia [1], TxCache [41], HBase [27], and Spanner [17], also recognized the importance of limited transactions. Sinfonia provides "mini" transactions to distributed shared memory and TXCache provides a consistent but potentially stale cache for a relational database, but both only considers operations within a single datacenter. HBase includes read- and write-only transactions within a single "region," which is a subset of the capacity of a single node. Spanner's read-only transactions are similar to the original distributed read-only transactions [14], in that they always take at least two rounds and block until all involved servers can guarantee they have applied all transactions that committed before the read-only transaction started. In comparison, Eiger is designed for geo-replicated storage and its transactions can execute across large cluster of nodes, normally only take one round, and never block.

The widely used MVCC algorithm [9, 43] and Eiger maintain multiple versions of objects so they can provide clients with a consistent view of a system. MVCC provides full snapshot isolation, sometimes rejects writes, has state linear in the number of recent reads and writes, and has a sweeping process that removes old versions. Eiger, in contrast, provides only read-only transactions, never rejects writes, has at worst state linear in the number of recent writes, and avoids storing most old versions while using fast timeouts for cleaning the rest.

8 Conclusion

Eiger is a new step forward in the design of low-latency geo-replicated storage systems. It is the first system to combine a rich data model capable of supporting today's large online services with stronger semantics, including causal consistency with read-only and write-only transactions. Our evaluation demonstrates that despite these new capabilities, Eiger provides competitive performance to its eventually consistent counterparts.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM TOCS*, 27(3), 2009.

[2] M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9 (1), 1995.

[3] Amazon. Simple storage service. http://aws.amazon.com/s3/, 2012.

[4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM TOCS*, 14(1), 1996.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[6] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2), 1994.

[7] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, Jan. 2011.

[8] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, May 2006.

[9] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computer Surveys*, 13(2), June 1981.

[10] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Comp. Soc. Press, 1994.

[11] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPoPP*, Jan. 2010.

[12] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *SOCC*, 2011.

[13] Cassandra. http://cassandra.apache.org/, 2012.

[14] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Trans. Info. Theory*, 11(2), 1985.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2), 2008.

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

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, Oct 2012.

[18] CouchDB. http://couchdb.apache.org/, 2012.

[19] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *USENIX ATC*, Jun 2012.

[20] P. Dixon. Shopzilla site redesign: We get what we measure. Velocity Conference Talk, 2009.

[21] eBay. Personal communication, 2012.

[22] Facebook. Personal communication, 2011.

[23] J. Ferris. The TAO graph database. CMU PDL Talk, April 2012.

[24] B. Fitzpatrick. Memcached: a distributed memory object caching system. http://memcached.org/, 2011.

[25] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[26] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *SOSP*, Oct. 2011.

[27] HBase. http://hbase.apache.org/, 2012.

[28] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[29] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[30] T. Kraska, G. Pang, M. J. Franklin, and S. Madden. MDCC: Multi-data center consistency. *CoRR*, abs/1203.6049, 2012.

[31] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4), 1992.

[32] A. Lakshman and P. Malik. Cassandra – a decentralized structured storage system. In *LADIS*, Oct. 2009.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7), 1978.

[34] L. Lamport. The part-time parliament. *ACM TOCS*,

16(2), 1998.

[35] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, Oct 2012.

[36] G. Linden. Make data useful. Stanford CS345 Talk, 2006.

[37] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[38] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, Oct. 2011.

[39] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, Oct. 1997.

[40] L. Peterson, A. Bavier, and S. Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton Univ., Dept. Comp. Sci., 2011.

[41] D. R. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, Oct. 2010.

[42] Redis. http://redis.io/, 2012.

[43] D. P. Reed. *Naming and Synchronization in a Decentralized Computer Systems*. PhD thesis, Mass. Inst. of Tech., 1978.

[44] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. Velocity Conference Talk, 2009.

[45] D. Skeen. A formal model of crash recovery in a distributed system. *IEEE Trans. Info. Theory*, 9(3), May 1983.

[46] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, Oct. 2011.

[47] TAO. A read-optimized globally distributed store for social graph data. Under Submission, 2012.

[48] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Sys.*, 4(2), 1979.

[49] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, May 2012.

[50] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, Dec. 2004.

[51] VICCI. http://vicci.org/, 2012.

Appeared in ***Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)***

Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS

Wyatt Lloyd*, Michael J. Freedman*, Michael Kaminsky†, and David G. Andersen‡
*Princeton University, †Intel Labs, ‡Carnegie Mellon University

ABSTRACT

Geo-replicated, distributed data stores that support complex online applications, such as social networks, must provide an "always-on" experience where operations always complete with low latency. Today's systems often sacrifice strong consistency to achieve these goals, exposing inconsistencies to their clients and necessitating complex application logic. In this paper, we identify and define a consistency model—causal consistency with convergent conflict handling, or *causal+*—that is the strongest achieved under these constraints.

We present the design and implementation of COPS, a key-value store that delivers this consistency model across the wide-area. A key contribution of COPS is its scalability, which can enforce causal dependencies between keys stored across an entire cluster, rather than a single server like previous systems. The central approach in COPS is tracking and explicitly checking whether causal dependencies between keys are satisfied in the local cluster before exposing writes. Further, in COPS-GT, we introduce get transactions in order to obtain a consistent view of multiple keys without locking or blocking. Our evaluation shows that COPS completes operations in less than a millisecond, provides throughput similar to previous systems when using one server per cluster, and scales well as we increase the number of servers in each cluster. It also shows that COPS-GT provides similar latency, throughput, and scaling to COPS for common workloads.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Distributed Systems

General Terms

Design, Experimentation, Performance

Keywords

Key-value storage, causal+ consistency, scalable wide-area replication, ALPS systems, read transactions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP '11, October 23–26, 2011, Cascais, Portugal.
Copyright 2011 ACM 978-1-4503-0977-6/11/10 . . . $10.00.

1. INTRODUCTION

Distributed data stores are a fundamental building block of modern Internet services. Ideally, these data stores would be strongly consistent, always available for reads and writes, and able to continue operating during network partitions. The CAP Theorem, unfortunately, proves it impossible to create a system that achieves all three [13, 23]. Instead, modern web services have chosen overwhelmingly to embrace availability and partition tolerance at the cost of strong consistency [16, 20, 30]. This is perhaps not surprising, given that this choice also enables these systems to provide low latency for client operations and high scalability. Further, many of the earlier high-scale Internet services, typically focusing on web search, saw little reason for stronger consistency, although this position is changing with the rise of interactive services such as social networking applications [46]. We refer to systems with these four properties—**A**vailability, low **L**atency, **P**artition-tolerance, and high **S**calability—as ALPS systems.

Given that ALPS systems must sacrifice strong consistency (i.e., linearizability), we seek the strongest consistency model that is achievable under these constraints. Stronger consistency is desirable because it makes systems easier for a programmer to reason about. In this paper, we consider *causal consistency with convergent conflict handling*, which we refer to as *causal+ consistency*. Many previous systems believed to implement the weaker causal consistency [10, 41] actually implement the more useful causal+ consistency, though none do so in a scalable manner.

The causal component of causal+ consistency ensures that the data store respects the causal dependencies between operations [31]. Consider a scenario where a user uploads a picture to a web site, the picture is saved, and then a reference to it is added to that user's album. The reference "depends on" the picture being saved. Under causal+ consistency, these dependencies are always satisfied. Programmers never have to deal with the situation where they can get the reference to the picture but not the picture itself, unlike in systems with weaker guarantees, such as eventual consistency.

The convergent conflict handling component of causal+ consistency ensures that replicas never permanently diverge and that conflicting updates to the same key are dealt with identically at all sites. When combined with causal consistency, this property ensures that clients see only progressively newer versions of keys. In comparison, eventually consistent systems may expose versions out of order. By combining causal consistency and convergent conflict handling, causal+ consistency ensures clients see a causally-correct, conflict-free, and always-progressing data store.

Our COPS system (Clusters of Order-Preserving Servers) provides causal+ consistency and is designed to support complex online applications that are hosted from a small number of large-scale datacenters, each of which is composed of front-end servers (clients of

COPS) and back-end key-value data stores. COPS executes all put and get operations in the local datacenter in a linearizable fashion, and it then replicates data across datacenters in a causal+ consistent order in the background.

We detail two versions of our COPS system. The regular version, COPS, provides scalable causal+ consistency between individual items in the data store, even if their causal dependencies are spread across many different machines in the local datacenter. These consistency properties come at low cost: The performance and overhead of COPS is similar to prior systems, such as those based on log exchange [10, 41], even while providing much greater scalability.

We also detail an extended version of the system, COPS-GT, which also provides *get transactions* that give clients a consistent view of multiple keys. Get transactions are needed to obtain a consistent view of multiple keys, even in a fully-linearizable system. Our get transactions require no locks, are non-blocking, and take at most two parallel rounds of intra-datacenter requests. To the best of our knowledge, COPS-GT is the first ALPS system to achieve non-blocking scalable get transactions. These transactions do come at some cost: compared to the regular version of COPS, COPS-GT is less efficient for certain workloads (e.g., write-heavy) and is less robust to long network partitions and datacenter failures.

The scalability requirements for ALPS systems creates the largest distinction between COPS and prior causal+ consistent systems. Previous systems required that all data fit on a single machine [2, 12, 41] or that all data that potentially could be accessed together fit on a single machine [10]. In comparison, data stored in COPS can be spread across an arbitrary-sized datacenter, and dependencies (or get transactions) can stretch across many servers in the datacenter. To the best of our knowledge, COPS is the first scalable system to implement causal+ (and thus causal) consistency.

The contributions in this paper include:

- We explicitly identify four important properties of distributed data stores and use them to define ALPS systems.
- We name and formally define causal+ consistency.
- We present the design and implementation of COPS, a *scalable* system that efficiently realizes the causal+ consistency model.
- We present a non-blocking, lock-free get transaction algorithm in COPS-GT that provides clients with a consistent view of multiple keys in at most two rounds of local operations.
- We show through evaluation that COPS has low latency, high throughput, and scales well for all tested workloads; and that COPS-GT has similar properties for common workloads.

2. ALPS SYSTEMS AND TRADE-OFFS

We are interested in infrastructure that can support many of today's largest Internet services. In contrast with classical distributed storage systems that focused on local-area operation in the small, these services are typically characterized by wide-area deployments across a few to tens of datacenters, as illustrated in Figure 1. Each datacenter includes a set of application-level clients, as well as a back-end data store to which these clients read and write. For many applications—and the setting considered in the paper—data written in one datacenter is replicated to others.

Often, these clients are actually webservers that run code on behalf of remote browsers. Although this paper considers consistency from the perspective of the application client (i.e., the webserver), if the browser accesses a service through a single datacenter, as we expect, it will enjoy similar consistency guarantees.

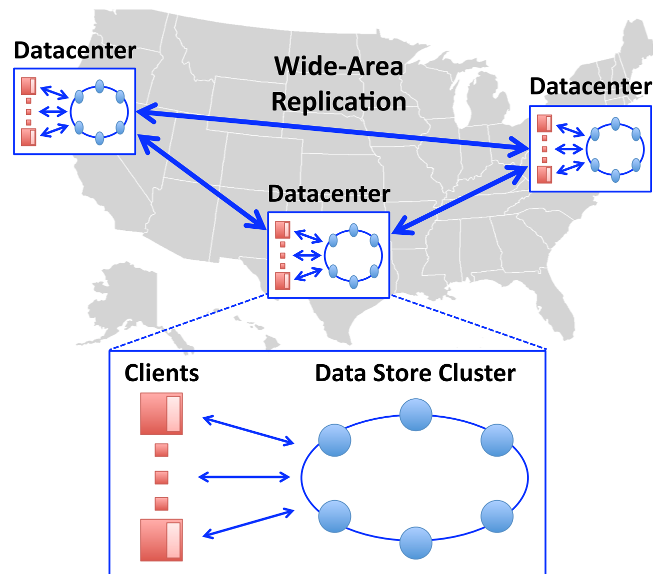


Figure 1: The general architecture of modern web services. Multiple geographically distributed datacenters each have application clients that read and write state from a data store that is distributed across all of the datacenters.

Such a distributed storage system has multiple, sometimes competing, goals: *availability*, *low latency*, and *partition tolerance* to provide an "always on" user experience [16]; *scalability* to adapt to increasing load and storage demands; and a sufficiently strong *consistency* model to simplify programming and provide users with the system behavior that they expect. In slightly more depth, the desirable properties include:

1. Availability. All operations issued to the data store complete successfully. No operation can block indefinitely or return an error signifying that data is unavailable.

2. Low Latency. Client operations complete "quickly." Commercial service-level objectives suggest average performance of a few milliseconds and worse-case performance (i.e., 99.9th percentile) of 10s or 100s of milliseconds [16].

3. Partition Tolerance. The data store continues to operate under network partitions, e.g., one separating datacenters in Asia from the United States.

4. High Scalability. The data store scales out linearly. Adding N resources to the system increases aggregate throughput and storage capacity by $O(N)$.

5. Stronger Consistency. An ideal data store would provide *linearizability*—sometimes informally called *strong consistency*—which dictates that operations appear to take effect across the entire system at a single instance in time between the invocation and completion of the operation [26]. In a data store that provides linearizability, as soon as a client completes a write operation to an object in one datacenter, read operations to the same object in all other datacenters will reflect its newly written state. Linearizability simplifies programming—the distributed system provides a single, consistent image—and users experience the storage behavior they expect. Weaker, eventual consistency models, common in many large distributed systems, are less intuitive: Not only might subsequent reads not reflect the latest value, reads across multiple objects might reflect an incoherent mix of old and new values.

The CAP Theorem proves that a shared-data system that has availability and partition tolerance cannot achieve linearizability [13,

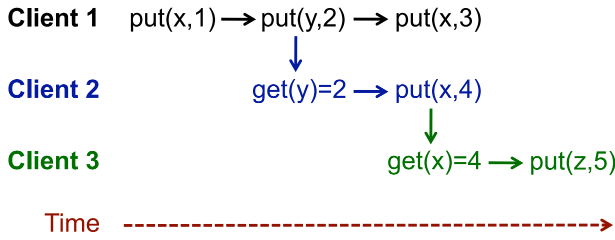


Figure 2: Graph showing the causal relationship between operations at a replica. An edge from *a* to *b* indicates that $a \rightsquigarrow b$, or *b* depends on *a*.

23]. Low latency—defined as latency less than the maximum wide-area delay between replicas—has also been proven incompatible with linearizability [34] and sequential consistency [8]. To balance between the requirements of ALPS systems and programmability, we define an intermediate consistency model in the next section.

3. CAUSAL+ CONSISTENCY

To define *causal consistency with convergent conflict handling* (causal+ consistency), we first describe the abstract model over which it operates. We restrict our consideration to a key-value data store, with two basic operations: `put`(key,val) and `get`(key)=val. These are equivalent to write and read operations in a shared-memory system. Values are stored and retrieved from logical *replicas*, each of which hosts the entire key space. In our COPS system, a single *logical* replica corresponds to an entire local cluster of nodes.

An important concept in our model is the notion of *potential causality* [2, 31] between operations. Three rules define potential causality, denoted $\rightsquigarrow$:

1. **Execution Thread.** If a and b are two operations in a *single thread of execution*, then $a \rightsquigarrow b$ if operation a happens before operation b.
2. **Gets From.** If a is a `put` operation and b is a `get` operation that returns the value written by a, then $a \rightsquigarrow b$.
3. **Transitivity.** For operations a, b, and c, if $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

These rules establish potential causality between operations within the same execution thread and between operations whose execution threads interacted through the data store. Our model, like many, does not allow threads to communicate directly, requiring instead that all communication occur through the data store.

The example execution in Figure 2 demonstrates all three rules. The *execution thread* rule gives `get`(y)=2 $\rightsquigarrow$ `put`(x,4); the *gets from* rule gives `put`(y,2) $\rightsquigarrow$ `get`(y)=2; and the *transitivity* rule gives `put`(y,2) $\rightsquigarrow$ `put`(x,4). Even though some operations follow `put`(x,3) in real time, no other operations depend on it, as none read the value it wrote nor follow it in the same thread of execution.

3.1 Definition

We define causal+ consistency as a combination of two properties: causal consistency and convergent conflict handling. We present intuitive definitions here and the formal definitions in Appendix A.

Causal consistency requires that values returned from `get` operations at a replica are consistent with the order defined by $\rightsquigarrow$ (causality) [2]. In other words, it must appear the operation that writes a value occurs after all operations that causally precede it. For example, in Figure 2, it must appear `put`(y,2) happened before `put`(x,4), which in turn happened before `put`(z,5). If client 2 saw `get`(x)=4 and then `get`(x)=1, causal consistency would be violated.

Causal consistency does not order concurrent operations. If $a \not\rightsquigarrow b$ and $b \not\rightsquigarrow a$, then a and b are concurrent. Normally, this allows increased efficiency in an implementation: Two unrelated put operations can be replicated in any order, avoiding the need for a serialization point between them. If, however, a and b are both puts to the same key, then they are in *conflict*.

Conflicts are undesirable for two reasons. First, because they are unordered by causal consistency, conflicts allow replicas to diverge forever [2]. For instance, if a is `put`(x,1) and b is `put`(x,2), then causal consistency allows one replica to forever return 1 for x and another replica to forever return 2 for x. Second, conflicts may represent an exceptional condition that requires special handling. For example, in a shopping cart application, if two people logged in to the same account concurrently add items to their cart, the desired result is to end up with both items in the cart.

Convergent conflict handling requires that all conflicting puts be handled in the same manner at all replicas, using a handler function h. This handler function h must be associative and commutative, so that replicas can handle conflicting writes in the order they receive them and that the results of these handlings will converge (e.g., one replica's $h(a, h(b, c))$ and another's $h(c, h(b, a))$ agree).

One common way to handle conflicting writes in a convergent fashion is the last-writer-wins rule (also called Thomas's write rule [50]), which declares one of the conflicting writes as having occurred later and has it overwrite the "earlier" write. Another common way to handle conflicting writes is to mark them as conflicting and require their resolution by some other means, e.g., through direct user intervention as in Coda [28], or through a programmed procedure as in Bayou [41] and Dynamo [16].

All potential forms of convergent conflict handling avoid the first issue—conflicting updates may continually diverge—by ensuring that replicas reach the same result after exchanging operations. On the other hand, the second issue with conflicts—applications may want special handling of conflicts—is only avoided by the use of more explicit *conflict resolution* procedures. These explicit procedures provide greater flexibility for applications, but require additional programmer complexity and/or performance overhead. Although COPS can be configured to detect conflicting updates explicitly and apply some application-defined resolution, the default version of COPS uses the last-writer-wins rule.

3.2 Causal+ vs. Other Consistency Models

The distributed systems literature defines several popular consistency models. In decreasing strength, they include: linearizability (or strong consistency) [26], which maintains a global, real-time ordering; sequential consistency [32], which ensures at least a global ordering; causal consistency [2], which ensures partial orderings between dependent operations; FIFO (PRAM) consistency [34], which only preserves the partial ordering of an execution thread, not between threads; per-key sequential consistency [15], which ensures that, for each individual key, all operations have a global order; and eventual consistency, a "catch-all" term used today suggesting eventual convergence to some type of agreement.

The causal+ consistency we introduce falls between sequential and causal consistency, as shown in Figure 3. It is weaker than sequential consistency, but sequential consistency is provably not achievable in an ALPS system. It is stronger than causal consistency

Linearizability > **Sequential** > Causal+ > Causal > FIFO
> Per-Key Sequential > Eventual

Figure 3: A spectrum of consistency models, with stronger models on the left. Bolded models are provably incompatible with ALPS systems.

and per-key sequential consistency, however, and it *is* achievable for ALPS systems. Mahajan et al. [35] have concurrently defined a similar strengthening of causal consistency; see Section 7 for details.

To illustrate the utility of the causal+ model, consider two examples. First, let Alice try to share a photo with Bob. Alice uploads the photo and then adds the photo to her album. Bob then checks Alice's album expecting to see her photo. Under causal and thus causal+ consistency, if the album has a reference to the photo, then Bob must be able to view the photo. Under per-key sequential and eventual consistency, it is possible for the album to have a reference to a photo that has not been written yet.

Second, consider an example where Carol and Dan both update the starting time for an event. The time was originally set for 9pm, Carol changed it to 8pm, and Dan concurrently changed it to 10pm. Regular causal consistency would allow two different replicas to forever return different times, even after receiving both put operations. Causal+ consistency requires that replicas handle this conflict in a convergent manner. If a last-writer-wins policy is used, then either Dan's 10pm or Carol's 8pm would win. If a more explicit conflict resolution policy is used, the key could be marked as in conflict and future gets on it could return both 8pm and 10pm with instructions to resolve the conflict.

If the data store was sequentially consistent or linearizable, it would still be possible for there to be two simultaneous updates to a key. In these stronger models, however, it is possible to implement mutual exclusion algorithms—such as the one suggested by Lamport in the original sequential consistency paper [32]—that can be used to avoid creating a conflict altogether.

3.3 Causal+ in COPS

We use two abstractions in the COPS system, *versions* and *dependencies*, to help us reason about causal+ consistency. We refer to the different values a key has as the *versions* of a key, which we denote $\text{key}_{\text{version}}$. In COPS, versions are assigned in a manner that ensures that if $x_i \rightsquigarrow y_j$ then $i < j$. Once a replica in COPS returns version i of a key, x_i, causal+ consistency ensures it will then only return that version or a causally later version (note that the handling of a conflict is causally later than the conflicting puts it resolves).[1] Thus, each replica in COPS always returns non-decreasing versions of a key. We refer to this as causal+ consistency's *progressing property*.

Causal consistency dictates that all operations that causally precede a given operations must appear to take effect before it. In other words, if $x_i \rightsquigarrow y_j$, then x_i must be written before y_j. We call these preceding values *dependencies*. More formally, we say y_j ***depends on*** x_i if and only if $\texttt{put}(x_i) \rightsquigarrow \texttt{put}(y_j)$. These dependencies are in essence the reverse of the causal ordering of writes. COPS provides causal+ consistency during replication by writing a version only after writing all of its dependencies.

[1] To see this, consider by contradiction the following scenario: assume a replica first returns x_i and then x_k, where $i \neq k$ and $x_i \not\rightsquigarrow x_k$. Causal consistency ensures that if x_k is returned after x_i, then $x_k \not\rightsquigarrow x_i$, and so x_i and x_k conflict. But, if x_i and x_k conflict, then convergent conflict handling ensures that as soon as both are present at a replica, their handling $h(x_i,x_k)$, which is causally after both, will be returned instead of either x_i or x_k, which contradicts our assumption.

3.4 Scalable Causality

To our knowledge, this paper is the first to name and formally define causal+ consistency. Interestingly, several previous systems [10, 41] believed to achieve causal consistency in fact achieved the stronger guarantees of causal+ consistency.

These systems were not designed to and do not provide *scalable* causal (or causal+) consistency, however, as they all use a form of log serialization and exchange. All operations at a logical replica are written to a single log in serialized order, commonly marked with a version vector [40]. Different replicas then exchange these logs, using version vectors to establish potential causality and detect concurrency between operations at different replicas.

Log-exchange-based serialization inhibits replica scalability, as it relies on a single serialization point in each replica to establish ordering. Thus, either causal dependencies between keys are limited to the set of keys that can be stored on one node [10, 15, 30, 41], or a single node (or replicated state machine) must provide a commit ordering and log for all operations across a cluster.

As we show below, COPS achieves scalability by taking a different approach. Nodes in each datacenter are responsible for different partitions of the keyspace, but the system can track and enforce dependencies between keys stored on different nodes. COPS explicitly encodes dependencies in metadata associated with each key's version. When keys are replicated remotely, the receiving datacenter performs dependency checks before committing the incoming version.

4. SYSTEM DESIGN OF COPS

COPS is a distributed storage system that realizes causal+ consistency and possesses the desired ALPS properties. There are two distinct versions of the system. The first, which we refer to simply as COPS, provides a data store that is causal+ consistent. The second, called COPS-GT, provides a superset of this functionality by also introducing support for *get transactions*. With get transactions, clients request a set of keys and the data store replies with a consistent snapshot of corresponding values. Because of the additional metadata needed to enforce the consistency properties of get transactions, a given deployment must run exclusively as COPS or COPS-GT.

4.1 Overview of COPS

COPS is a key-value storage system designed to run across a small number of datacenters, as illustrated in Figure 4. Each datacenter has a local COPS *cluster* with a complete replica of the stored data.[2] A *client* of COPS is an application that uses the COPS *client library* to call directly into the COPS key-value store. Clients communicate only with their local COPS cluster running in the same datacenter.

Each local COPS cluster is set up as a linearizable (strongly consistent) key-value store [5, 48]. Linearizable systems can be implemented scalably by partitioning the keyspace into N linearizable partitions (each of which can reside on a single node or a single chain of nodes) and having clients access each partition independently. The composability of linearizability [26] ensures that the resulting system as a whole remains linearizable. Linearizability is acceptable locally because we expect very low latency and no

[2] The assumption of full replication simplifies our presentation, though one could imagine clusters that replicate only part of the total data store and sacrifice low latency for the rest (according to configuration rules).

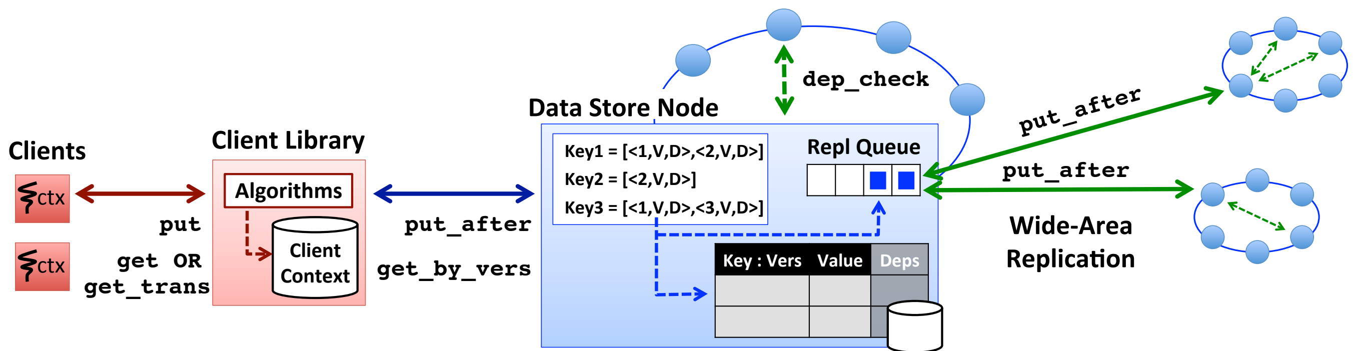


Figure 4: The COPS architecture. A client library exposes a put/get interface to its clients and ensures operations are properly labeled with causal dependencies. A key-value store replicates data between clusters, ensures writes are committed in their local cluster only after their dependencies have been satisfied, and in COPS-GT, stores multiple versions of each key along with dependency metadata.

partitions within a cluster—especially with the trend towards redundant paths in modern datacenter networks [3, 24]—unlike in the wide-area. On the other hand, replication *between* COPS clusters happens asynchronously to ensure low latency for client operations and availability in the face of external partitions.

System Components. COPS is composed of two main software components:

- *Key-value store.* The basic building block in COPS is a standard key-value store that provides linearizable operations on keys. COPS extends the standard key-value store in two ways, and COPS-GT adds a third extension.
  1. Each key-value pair has associated metadata. In COPS, this metadata is a version number. In COPS-GT, it is both a version number and a list of dependencies (other keys and their respective versions).
  2. The key-value store exports three additional operations as part of its key-value interface: `get_by_version`, `put_after`, and `dep_check`, each described below. These operations enable the COPS client library and an asynchronous replication process that supports causal+ consistency and get transactions.
  3. For COPS-GT, the system keeps around old versions of key-value pairs, not just the most recent `put`, to ensure that it can provide get transactions. Maintaining old versions is discussed further in Section 4.3.
- *Client library.* The client library exports two main operations to applications: reads via `get` (in COPS) or `get_trans` (in COPS-GT), and writes via `put`.[3] The client library also maintains state about a client's current dependencies through a *context* parameter in the client library API.

Goals. The COPS design strives to provide causal+ consistency with resource and performance overhead similar to existing eventually consistent systems. COPS and COPS-GT must therefore:

- *Minimize overhead of consistency-preserving replication.* COPS must ensure that values are replicated between clusters in a causal+ consistent manner. A naive implementation, however, would require checks on all of a value's dependencies. We present a mechanism that requires only a small number of such checks by leveraging the graph structure inherent to causal dependencies.

[3]This paper uses different fixed-width fonts for client-facing API calls (e.g., `get`) and data store API calls (e.g., `get_by_version`).

- *(COPS-GT) Minimize space requirements.* COPS-GT stores (potentially) multiple versions of each key, along with their associated dependency metadata. COPS-GT uses aggressive garbage collection to prune old state (see Section 5.1).
- *(COPS-GT) Ensure fast `get_trans` operations.* The get transactions in COPS-GT ensure that the set of returned values are causal+ consistent (all dependencies are satisfied). A naive algorithm could block and/or take an unbounded number of get rounds to complete. Both situations are incompatible with the availability and low latency goals of ALPS systems; we present an algorithm for `get_trans` that completes in at most two rounds of local `get_by_version` operations.

4.2 The COPS Key-Value Store

Unlike traditional ⟨*key*, *val*⟩–tuple stores, COPS must track the versions of written values, as well as their dependencies in the case of COPS-GT. In COPS, the system stores the most recent version number and value for each key. In COPS-GT, the system maps each key to a list of version entries, each consisting of ⟨*version*, *value*, *deps*⟩. The *deps* field is a list of the version's zero or more dependencies; each dependency is a ⟨*key*, *version*⟩ pair.

Each COPS cluster maintains its own copy of the key-value store. For scalability, our implementation partitions the keyspace across a cluster's nodes using consistent hashing [27], through other techniques (e.g., directory-based approaches [6, 21]) are also possible. For fault tolerance, each key is replicated across a small number of nodes using chain replication [5, 48, 51]. Gets and puts are linearizable across the nodes in the cluster. Operations return to the client library as soon as they execute in the *local* cluster; operations between clusters occur asynchronously.

Every key stored in COPS has one *primary* node in each cluster. We term the set of primary nodes for a key across all clusters as the *equivalent nodes* for that key. In practice, COPS's consistent hashing assigns each node responsibility for a few different key ranges. Key ranges may have different sizes and node mappings in different datacenters, but the total number of equivalent nodes with which a given node needs to communicate is proportional to the number of datacenters (i.e., communication is *not* all-to-all between nodes in different datacenters).

After a write completes locally, the primary node places it in a replication queue, from which it is sent asynchronously to remote equivalent nodes. Those nodes, in turn, wait until the value's dependencies are satisfied in their local cluster before locally committing

```
# Alice's Photo Upload
ctx_id = createContext()   // Alice logs in
put(Photo, "Portuguese Coast", ctx_id)
put(Album, "add &Photo", ctx_id)
deleteContext(ctx_id)      // Alice logs out

# Bob's Photo View
ctx_id = createContext()   // Bob logs in
"&Photo" ← get(Album, ctx_id)
"Portuguese Coast" ← get(Photo, ctx_id)
deleteContext(ctx_id)      // Bob logs out
```

Figure 5: Snippets of pseudocode using the COPS programmer interface for the photo upload scenario from Section 3.2. When using COPS-GT, each `get` would instead be a `get_trans` on a single key.

the value. This dependency checking mechanism ensures writes happen in a causally consistent order and reads never block.

4.3 Client Library and Interface

The COPS client library provides a simple and intuitive programming interface. Figure 5 illustrates the use of this interface for the photo upload scenario. The client API consists of four operations:

1. *ctx_id* ← `createContext()`
2. bool ← `deleteContext`(*ctx_id*)
3. bool ← `put` (*key*, *value*, *ctx_id*)
4. *value* ← `get` (*key*, *ctx_id*) [In COPS]
   or
4. ⟨*values*⟩ ← `get_trans` (⟨*keys*⟩, *ctx_id*) [In COPS-GT]

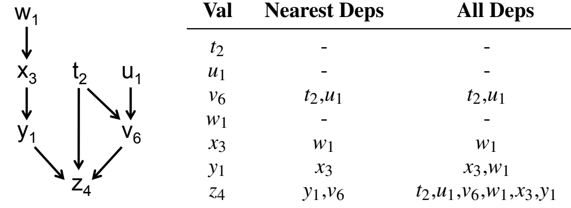
The client API differs from a traditional key-value interface in two ways. First, COPS-GT provides `get_trans`, which returns a consistent view of multiple key-value pairs in a single call. Second, all functions take a context argument, which the library uses internally to track causal dependencies across each client's operations [49]. The context defines the causal+ "thread of execution." A single process may contain many separate threads of execution (e.g., a web server concurrently serving 1000s of independent connections). By separating different threads of execution, COPS avoids false dependencies that would result from intermixing them.

We next describe the state kept by the client library in COPS-GT to enforce consistency in get transactions. We then show how COPS can store significantly less dependency state.

COPS-GT Client Library. The client library in COPS-GT stores the client's context in a table of ⟨*key*, *version*, *deps*⟩ entries. Clients reference their context using a *context ID* (*ctx_id*) in the API.[4] When a client gets a key from the data store, the library adds this key and its causal dependencies to the context. When a client puts a value, the library sets the put's dependencies to the most recent version of each key in the current context. A successful put into the data store returns the version number v assigned to the written value. The client library then adds this new entry, ⟨*key*, v, D⟩, to the context.

The context therefore includes all values previously read or written in the client's session, as well as all of those dependencies' dependencies, as illustrated in Figure 6. This raises two concerns about the potential size of this causality graph: (i) state requirements for storing these dependencies, both in the client library and in the data store, and (ii) the number of potential checks that must occur when replicating writes between clusters, in order to ensure causal consistency. To mitigate the client and data-store state required to track dependencies, COPS-GT provides garbage collection, described in Section 5.1, that removes dependencies once they are committed to all COPS replicas.

[4]Maintaining state in the library and passing in an ID was a design choice; one could also encode the entire context table as an opaque blob and pass it between client and library so that the library is stateless.



Val	Nearest Deps	All Deps
t_2	-	-
u_1	-	-
v_6	t_2,u_1	t_2,u_1
w_1	-	-
x_3	w_1	w_1
y_1	x_3	x_3,w_1
z_4	y_1,v_6	t_2,u_1,v_6,w_1,x_3,y_1

Figure 6: A sample graph of causal dependencies for a client context. Arrows indicate causal relationships (e.g., x_3 depends on w_1). The table lists all dependencies for each value and the "nearest" dependencies used to minimize dependency checks.

To reduce the number of dependency checks during replication, the client library identifies several potential optimizations for servers to use. Consider the graph in Figure 6. y_1 depends on x_3 and, by transitivity, on w_1. If the storage node committing y_1 determines that x_3 has been committed, then it can infer that w_1 has also been committed, and thus, need not check for it explicitly. Similarly, while z_4 depends directly on t_2 and v_6, the committing node needs only check v_6, because v_6 itself depends on t_2.

We term dependencies that *must* be checked the *nearest* dependencies, listed in the table in Figure 6.[5] To enable servers to use these optimizations, the client library first computes the nearest dependencies within the write's dependency list and marks them accordingly when issuing the write.

The nearest dependencies are sufficient for the key-value store to provide causal+ consistency; the full dependency list is only needed to provide `get_trans` operations in COPS-GT.

COPS Client Library. The client library in COPS requires significantly less state and complexity because it only needs to learn the nearest, rather than all, dependencies. Accordingly, it does not store or even retrieve the dependencies of any value it gets: The retrieved value is nearer than any of its dependencies, rendering them unnecessary.

Thus, the COPS client library stores only ⟨*key*, *version*⟩ entries. For a `get` operation, the retrieved ⟨*key*, *version*⟩ is added to the context. For a put operation, the library uses the current context as the nearest dependencies, clears the context, and then repopulates it with only this put. This put depends on all previous key-version pairs and thus is nearer than them.

4.4 Writing Values in COPS and COPS-GT

Building on our description of the client library and key-value store, we now walk through the steps involved in writing a value to COPS. All writes in COPS first go to the client's local cluster and then propagate asynchronously to remote clusters. The key-value store exports a single API call to provide both operations:

⟨bool,*vers*⟩ ← `put_after` (*key*, *val*, [*deps*], *nearest*, *vers*=∅)

[5]In graph-theoretic terms, the nearest dependencies of a value are those in the causality graph with a *longest* path to the value of length one.

Writes to the local cluster. When a client calls put (*key*,*val*,*ctx_id*), the library computes the complete set of dependencies *deps*, and identifies some of those dependency tuples as the value's *nearest* ones. The library then calls put_after without the *version* argument (i.e., it sets *version*=∅). In COPS-GT, the library includes *deps* in the put_after call because dependencies must be stored with the value; in COPS, the library only needs to include *nearest* and does not include *deps*.[6] The key's primary storage node in the local cluster assigns the key a version number and returns it to the client library. We restrict each client to a single outstanding put; this is necessary because later puts must know the version numbers of earlier puts so they may depend on them.

The put_after operation ensures that *val* is committed to each cluster only *after* all of the entries in its dependency list have been written. In the client's local cluster, this property holds automatically, as the local store provides linearizability. (If *y* depends on *x*, then put(*x*) must have been committed before put(*y*) was issued.) This is not true in remote clusters, however, which we discuss below.

The primary storage node uses a Lamport timestamp [31] to assign a unique version number to each update. The node sets the version number's high-order bits to its Lamport clock and the low-order bits to its unique node identifier. Lamport timestamps allow COPS to derive a single global order over all writes for each key. This order implicitly implements the last-writer-wins convergent conflict handling policy. COPS is also capable of explicitly detecting and resolving conflicts, which we discuss in Section 5.3. Note that because Lamport timestamps provide a partial ordering of all distributed events in a way that respects potential causality, this global ordering is compatible with COPS's causal consistency.

Write replication between clusters. After a write commits locally, the primary storage node asynchronously replicates that write to its equivalent nodes in different clusters using a stream of put_after operations; here, however, the primary node includes the key's version number in the put_after call. As with local put_after calls, the *deps* argument is included in COPS-GT, and not included in COPS. This approach scales well and avoids the need for a single serialization point, but requires the remote nodes receiving updates to commit an update only after its dependencies have been committed to the same cluster.

To ensure this property, a node that receives a put_after request from another cluster must determine if the value's *nearest* dependencies have already been satisfied locally. It does so by issuing a check to the local nodes responsible for the those dependencies:

bool ← dep_check (*key*, *version*)

When a node receives a dep_check, it examines its local state to determine if the dependency value has already been written. If so, it immediately responds to the operation. If not, it blocks until the needed version has been written.

If all dep_check operations on the nearest dependencies succeed, the node handling the put_after request commits the written value, making it available to other reads and writes in its local cluster. (If any dep_check operation times out the node handling the put_after reissues it, potentially to a new node if a failure occurred.) The way that *nearest* dependencies are computed ensures that *all* dependencies have been satisfied before the value is committed, which in turn ensures causal consistency.

[6] We use bracket notation ([]) to indicate an argument is optional; the optional arguments are used in COPS-GT, but not in COPS.

4.5 Reading Values in COPS

Like writes, reads are satisfied in the local cluster. Clients call the get library function with the appropriate context; the library in turn issues a read to the node responsible for the key in the local cluster:

⟨*value*, *version*, *deps*⟩ ← get_by_version (*key*, *version*=LATEST)

This read can request either the latest version of the key or a specific older one. Requesting the latest version is equivalent to a regular single-key get; requesting a specific version is necessary to enable get transactions. Accordingly, get_by_version operations in COPS always request the latest version. Upon receiving a response, the client library adds the ⟨*key*,*version*[,*deps*]⟩ tuple to the client context, and returns *value* to the calling code. The *deps* are stored only in COPS-GT, not in COPS.

4.6 Get Transactions in COPS-GT

The COPS-GT client library provides a get_trans interface because reading a set of dependent keys using a single-key get interface cannot ensure causal+ consistency, even though the data store itself is causal+ consistent. We demonstrate this problem by extending the photo album example to include access control, whereby Alice first changes her album ACL to "friends only", and then writes a new description of her travels and adds more photos to the album.

A natural (but incorrect!) implementation of code to read Alice's album might (1) fetch the ACL, (2) check permissions, and (3) fetch the album description and photos. This approach contains a straightforward "time-to-check-to-time-to-use" race condition: when Eve accesses the album, her get(ACL) might return the old ACL, which permitted anyone (including Eve) to read it, but her get(album contents) might return the "friends only" version.

One straw-man solution is to require that clients issue single-key get operations in the reverse order of their causal dependencies: The above problem would not have occurred if the client executed get(album) before get(ACL). This solution, however, is *also* incorrect. Imagine that after updating her album, Alice decided that some photographs were too personal, so she (3) deleted those photos and rewrote the description, and then (4) marked the ACL open again. This straw-man has a different time-of-check-to-time-of-use error, where get(album) retrieves the private album, and the subsequent get(ACL) retrieves the "public" ACL. In short, there is no correct canonical ordering of the ACL and the album entries.

Instead, a better programming interface would allow the client to obtain a causal+ consistent view of multiple keys. The standard way to achieve such a guarantee is to read and write all related keys in a transaction; this, however, requires a single serialization point for all grouped keys, which COPS avoids for greater scalability and simplicity. Instead, COPS allows keys to be written independently (with explicit dependencies in metadata), and provides a get_trans operation for retrieving a consistent view of multiple keys.

Get transactions. To retrieve multiple values in a causal+ consistent manner, a client calls get_trans with the desired set of keys, e.g., get_trans(⟨ACL, album⟩). Depending on when and where it was issued, this get transaction can return different combinations of ACL and album, but never ⟨$\text{ACL}_{\text{public}}$, $\text{Album}_{\text{personal}}$⟩.

The COPS client library implements the get transactions algorithm in two rounds, shown in Figure 7. In the first round, the library issues *n* concurrent get_by_version operations to the local cluster, one for each key the client listed in get_trans. Because

```
# @param  keys    list of keys
# @param  ctx_id  context id
# @return values  list of values

function get_trans(keys, ctx_id):
  # Get keys in parallel (first round)
  for k in keys
    results[k] = get_by_version(k, LATEST)

  # Calculate causally correct versions (ccv)
  for k in keys
    ccv[k] = max(ccv[k], results[k].vers)
    for dep in results[k].deps
      if dep.key in keys
        ccv[dep.key] = max(ccv[dep.key], dep.vers)

  # Get needed ccvs in parallel (second round)
  for k in keys
    if ccv[k] > results[k].vers
      results[k] = get_by_version(k, ccv[k])

  # Update the metadata stored in the context
  update_context(results, ctx_id)

  # Return only the values to the client
  return extract_values(results)
```

Figure 7: Pseudocode for the `get_trans` algorithm.

COPS-GT commits writes locally, the local data store guarantees that each of these explicitly listed keys' dependencies are already satisfied—that is, they have been written locally and reads on them will immediately return. These explicitly listed, independently retrieved values, however, may not be consistent with one another, as shown above. Each `get_by_version` operation returns a ⟨*value*, *version*, *deps*⟩ tuple, where *deps* is a list of keys and versions. The client library then examines every dependency entry ⟨*key*, *version*⟩. The causal dependencies for that result are satisfied if either the client did not request the dependent key, or if it did, the version it retrieved was ≥ the version in the dependency list.

For all keys that are not satisfied, the library issues a second round of concurrent `get_by_version` operations. The version requested will be the newest version seen in any dependency list from the first round. These versions satisfy all causal dependencies from the first round because they are ≥ the needed versions. In addition, because dependencies are transitive and these second-round versions are all depended on by versions retrieved in the first round, they do not introduce any new dependencies that need to be satisfied. This algorithm allows `get_trans` to return a consistent view of the data store as of the time of the latest timestamp retrieved in first round.

The second round happens only when the client must read newer versions than those retrieved in the first round. This case occurs only if keys involved in the get transaction are updated during the first round. Thus, we expect the second round to be rare. In our example, if Eve issues a `get_trans` concurrent with Alice's writes, the algorithms first round of gets might retrieve the public ACL and the private album. The private album, however, depends on the "friends only" ACL, so the second round would fetch this newer version of the ACL, allowing `get_trans` to return a causal+ consistent set of values to the client.

The causal+ consistency of the data store provides two important properties for the get transaction algorithm's second round. First, the `get_by_version` requests will succeed immediately, as the requested version must already exist in the local cluster. Second, the new `get_by_version` requests will not introduce any new dependencies, as those dependencies were already known in the first round due to transitivity. This second property demonstrates why the get transaction algorithm specifies an explicit version in its second round, rather than just getting the latest: Otherwise, in the face of concurrent writes, a newer version could introduce still newer dependencies, which could continue indefinitely.

5. GARBAGE, FAULTS, AND CONFLICTS

This section describes three important aspects of COPS and COPS-GT: their garbage collection subsystems, which reduce the amount of extra state in the system; their fault tolerant design for client, node, and datacenter failures; and their conflict detection mechanisms.

5.1 Garbage Collection Subsystem

COPS and COPS-GT clients store metadata; COPS-GT servers additionally keeps multiple versions of keys and dependencies. Without intervention, the space footprint of the system would grow without bound as keys are updated and inserted. The COPS garbage collection subsystem deletes unneeded state, keeping the total system size in check. Section 6 evaluates the overhead of maintaining and transmitting this additional metadata.

Version Garbage Collection. (COPS-GT only)
What is stored: COPS-GT stores multiple versions of each key to answer `get_by_version` requests from clients.
Why it can be cleaned: The `get_trans` algorithm limits the number of versions needed to complete a get transaction. The algorithm's second round issues `get_by_version` requests only for versions later than those returned in the first round. To enable prompt garbage collection, COPS-GT limits the total running time of `get_trans` through a configurable parameter, *trans_time* (set to 5 seconds in our implementation). (If the timeout fires, the client library will restart the `get_trans` call and satisfy the transaction with newer versions of the keys; we expect this to happen only if multiple nodes in a cluster crash.)
When it can be cleaned: After a new version of a key is written, COPS-GT only needs to keep the old version around for *trans_time* plus a small delta for clock skew. After this time, no `get_by_version` call will subsequently request the old version, and the garbage collector can remove it.
Space Overhead: The space overhead is bounded by the number of old versions that can be created within the *trans_time*. This number is determined by the maximum write throughput that the node can sustain. Our implementation sustains 105MB/s of write traffic per node, requiring (potentially) a non-prohibitive extra 525MB of buffer space to hold old versions. This overhead is per-machine and does not grow with the cluster size or the number of datacenters.

Dependency Garbage Collection. (COPS-GT only)
What is stored: Dependencies are stored to allow get transactions to obtain a consistent view of the data store.
Why it can be cleaned: COPS-GT can garbage collect these dependencies once the versions associated with old dependencies are no longer needed for correctness in get transaction operations.

To illustrate when get transaction operations no longer need dependencies, consider value z_2 that depends on x_2 and y_2. A get transaction of x, y, and z requires that if z_2 is returned, then $x_{\geq 2}$ and $y_{\geq 2}$ must be returned as well. Causal consistency ensures that x_2 and y_2 are written before z_2 is written. Causal+ consistency's progressing property ensures that once x_2 and y_2 are written, then either they or some later version will always be returned by a get

operation. Thus, once z_2 has been written in all datacenters and the *trans_time* has passed, any get transaction returning z_2 will return $x_{\geq 2}$ and $y_{\geq 2}$, and thus z_2's dependencies can be garbage collected.
When it can be cleaned: After *trans_time* seconds after a value has been committed in *all* datacenters, COPS-GT can clean a value's dependencies. (Recall that committed enforces that its dependencies have been satisfied.) Both COPS and COPS-GT can further set the value's *never-depend* flag, discussed below. To clean dependencies each remote datacenter notifies the originating datacenter when the write has committed and the timeout period has elapsed. Once all datacenters confirm, the originating datacenter cleans its own dependencies and informs the others to do likewise. To minimize bandwidth devoted to cleaning dependencies, a replica only notifies the originating datacenter if this version of a key is the newest after *trans_time* seconds; if it is not, there is no need to collect the dependencies because the entire version will be collected.[7]
Space Overhead: Under normal operation, dependencies are garbage collected after *trans_time* plus a round-trip time. Dependencies are only collected on the most recent version of the key; older versions of keys are already garbage collected as described above.

During a partition, dependencies on the most recent versions of keys cannot be collected. This is a limitation of COPS-GT, although we expect long partitions to be rare. To illustrate why this concession is necessary for get transaction correctness, consider value b_2 that depends on value a_2: if b_2's dependence on a_2 is prematurely collected, some later value c_2 that causally depends on b_2—and thus on a_2—could be written without the explicit dependence on a_2. Then, if a_2, b_2, and c_2 are all replicated to a datacenter in short order, the first round of a get transaction could obtain a_1, an earlier version of a, with c_2, and then return these two values to the client, precisely because it did not know c_2 depends on the newer a_2.

Client Metadata Garbage Collection. (COPS + COPS-GT)
What is Stored: The COPS client library tracks all operations during a client session (single thread of execution) using the *ctx_id* passed with all operation. In contrast to the dependency information discussed above which resides in the key-value store itself, the dependencies discussed here are part of the client metadata and are store in the client library. In both systems, each `get` since the last `put` adds another nearest dependency. Additionally in COPS-GT, all new values and their dependencies returned in `get_trans` operations and all `put` operations add normal dependencies. If a client session lasts for a long time, the number of dependencies attached to updates will grow large, increasing the size of the dependency metadata that COPS needs to store.
Why it can be cleaned: As with the dependency tracking above, clients need to track dependencies only until they are guaranteed to be satisfied everywhere.
When it can be cleaned: COPS reduces the size of this client state (the context) in two ways. First, as noted above, once a `put_after` commits successfully to all datacenters, COPS flags that key version as *never-depend*, in order to indicate that clients need not express a dependence upon it. `get_by_version` results include this flag, and the client library will immediately remove a *never-depend* item from the list of dependencies in the client context. Furthermore, this process is transitive: Anything that a *never-depend* key depended on must have been flagged *never-depend*, so it too can be garbage collected from the context.

Second, the COPS storage nodes remove unnecessary dependencies from `put_after` operations. When a node receives a `put_after`, it checks each item in the dependency list and removes items with version numbers older than a *global checkpoint time*. This checkpoint time is the newest Lamport timestamp that is satisfied at *all* nodes across the entire system. The COPS key-value store returns this checkpoint time to the client library (e.g., in response to a `put_after`), allowing the library to clean these dependencies from the context.[8]

To compute the global checkpoint time, each storage node first determines the oldest Lamport timestamp of any *pending* `put_after` in the key range for which it is primary. (In other words, it determines the timestamp of its oldest key that is not guaranteed to be satisfied at all replicas.) It then contacts its equivalent nodes in other datacenters (those nodes that handle the same key range). The nodes pair-wise exchange their minimum Lamport times, remembering the oldest observed Lamport clock of any of the replicas. At the conclusion of this step, all datacenters have the same information: each node knows the globally oldest Lamport timestamp in its key range. The nodes within a datacenter then gossip around the per-range minimums to find the minimum Lamport timestamp observed by any one of them. This periodic procedure is done 10 times a second in our implementation and has no noticeable impact on performance.

5.2 Fault Tolerance

COPS is resilient to client, node, and datacenter failures. For the following discussion, we assume that failures are fail-stop: components halt in response to a failure instead of operating incorrectly or maliciously, and failures are detectable.

Client Failures. COPS's key-value interface means that each client request (through the library) is handled independently and atomically by the data store. From the storage system's perspective, if a client fails, it simply stops issuing new requests; no recovery is necessary. From a client's perspective, COPS's dependency tracking makes it easier to handle failures of other clients, by ensuring properties such as referential integrity. Consider the photo and album example: If a client fails after writing the photo, but before writing a reference to the photo, the data store will still be in a consistent state. There will never be an instance of the reference to the photo without the photo itself already being written.

Key-Value Node Failures. COPS can use any underlying fault-tolerant linearizable key-value store. We built our system on top of independent clusters of FAWN-KV [5] nodes, which use chain replication [51] *within* a cluster to mask node failures. Accordingly, we describe how COPS can use chain replication to provide tolerance to node failures.

Similar to the design of FAWN-KV, each data item is stored in a chain of R consecutive nodes along the consistent hashing ring. `put_after` operations are sent to the head of the appropriate chain, propagate along the chain, and then commit at the tail, which then acknowledges the operation. `get_by_version` operations are sent to the tail, which responds directly.

Server-issued operations are slightly more involved because they are issued from and processed by different chains of nodes. The tail in the local cluster replicates `put_after` operations to the head in each remote datacenter. The remote heads then send `dep_check` operations, which are essentially read operations, to the appropriate

[7] We are currently investigating if collecting dependencies in this manner provides a significant enough benefit over collecting them after the global checkpoint time (discussed below) to justify its messaging cost.

[8] Because of outstanding reads, clients and servers must also wait *trans_time* seconds before they can use a new global checkpoint time.

tails in their local cluster. Once these return (if the operation does not return, a timeout will fire and the `dep_check` will be reissued), the remote head propagates the value down the (remote) chain to the remote tail, which commits the value and acknowledges the operation back to the originating datacenter.

Dependency garbage collection follows a similar pattern of interlocking chains, though we omit details for brevity. Version garbage collection is done locally on each node and can operate as in the single node case. Calculation of the global checkpoint time, for client metadata garbage collection, operates normally with each tail updating its corresponding key range minimums.

Datacenter Failures. The partition-tolerant design of COPS also provides resiliency to entire datacenter failures (or partitions). In the face of such failures, COPS continues to operate as normal, with a few key differences.

First, any `put_after` operations that originated in the failed datacenter, but which were not yet copied out, will be lost. This is an inevitable cost of allowing low-latency local writes that return faster than the propagation delay between datacenters. If the datacenter is only partitioned and has not failed, no writes will be lost. Instead, they will only be delayed until the partition heals.[9]

Second, the storage required for replication queues in the active datacenters will grow. They will be unable to send `put_after` operations to the failed datacenter, and thus COPS will be unable to garbage collect those dependencies. The system administrator has two options: allow the queues to grow if the partition is likely to heal soon, or reconfigure COPS to no longer use the failed datacenter.

Third, in COPS-GT, dependency garbage collection cannot continue in the face of a datacenter failure, until either the partition is healed or the system is reconfigured to exclude the failed datacenter.

5.3 Conflict Detection

Conflicts occur when there are two "simultaneous" (i.e., not in the same context/thread of execution) writes to a given key. The default COPS system avoids conflict detection using a last-writer-wins strategy. The "last" write is determined by comparing version numbers, and allows us to avoid conflict detection for increased simplicity and efficiency. We believe this behavior is useful for many applications. There are other applications, however, that become simpler to reason about and program with a more explicit conflict-detection scheme. For these applications, COPS can be configured to detect conflicting operations and then invoke some application-specific convergent conflict handler.

COPS with conflict detection, which we refer to as COPS-CD, adds three new components to the system. First, all put operations carry with them *previous version* metadata, which indicates the most recent previous version of the key that was visible at the local cluster at the time of the write (this previous version may be null). Second, all put operations now have an implicit dependency on that previous version, which ensures that a new version will only be written after its previous version. This implicit dependency entails an additional `dep_check` operation, though this has low overhead and always executes on the local machine. Third, COPS-CD has an application-specified convergent conflict handler that is invoked when a conflict is detected.

[9] It remains an interesting aspect of future work to support flexibility in the number of datacenters required for committing within the causal+ model.

System	Causal+	Scalable	Get Trans
LOG	Yes	No	No
COPS	Yes	Yes	No
COPS-GT	Yes	Yes	Yes

Table 1: Summary of three systems under comparison.

COPS-CD follows a simple procedure to determine if a put operation *new* to a key (with previous version *prev*) is in conflict with the key's current visible version *curr*:

$prev \neq curr$ if and only if *new* and *curr* conflict.

We omit a full proof, but present the intuition here. In the forward direction, we know that *prev* must be written before *new*, $prev \neq curr$, and that for *curr* to be visible instead of *prev*, we must have $curr > prev$ by the progressing property of causal+. But because *prev* is the most recent causally previous version of *new*, we can conclude $curr \not\rightsquigarrow new$. Further, because *curr* was written before *new*, it cannot be causally after it, so $new \not\rightsquigarrow curr$ and thus they conflict. In the reverse direction, if *new* and *curr* conflict, then $curr \not\rightsquigarrow new$. By definition, $prev \rightsquigarrow new$, and thus $curr \neq prev$.

6. EVALUATION

This section presents an evaluation of COPS and COPS-GT using microbenchmarks that establish baseline system latency and throughput, a sensitivity analysis that explores the impact of different parameters that characterize a dynamic workload, and larger end-to-end experiments that show scalable causal+ consistency.

6.1 Implementation and Experimental Setup

COPS is approximately 13,000 lines of C++ code. It is built on top of FAWN-KV [5, 18] (~8500 LOC), which provides linearizable key-value storage within a local cluster. COPS uses Apache Thrift [7] for communication between all system components and Google's Snappy [45] for compressing dependency lists. Our current prototype implements all features described in the paper, excluding chain replication for local fault tolerance[10] and conflict detection.

We compare three systems: LOG, COPS, and COPS-GT. LOG uses the COPS code-base but excludes all dependency tracking, making it simulate previous work that uses log exchange to establish causal consistency (e.g., Bayou [41] and PRACTI [10]). Table 1 summarizes these three systems.

Each experiment is run on one cluster from the VICCI testbed [52]. The cluster's 70 servers give users an isolated Linux VServer. Each server has 2x6 core Intel Xeon X5650 CPUs, 48GB RAM, 3x1TB Hard Drives, and 2x1GigE network ports.

For each experiment, we partition the cluster into multiple logical "datacenters" as necessary. We retain full bandwidth between the nodes in different datacenters to reflect the high-bandwidth backbone that often exists between them. All reads and writes in FAWN-KV go to disk, but most operations in our experiments hit the kernel buffer cache.

The results presented are from 60-second trials. Data from the first and last 15s of each trial were elided to avoid experimental artifacts, as well as to allow garbage collection and replication mechanisms to ramp up. We run each trial 15 times and report the median; the minimum and maximum results are almost always within 6% of

[10] Chain replication was not fully functional in the version of FAWN-KV on which our implementation is built.

System	Operation	Latency (ms) 50%	99%	99.9%	Throughput (Kops/s)
Thrift	ping	0.26	3.62	12.25	60
COPS	get_by_version	0.37	3.08	11.29	52
COPS-GT	get_by_version	0.38	3.14	9.52	52
COPS	put_after (1)	0.57	6.91	11.37	30
COPS-GT	put_after (1)	0.91	5.37	7.37	24
COPS-GT	put_after (130)	1.03	7.45	11.54	20

Table 2: Latency (in ms) and throughput (in Kops/s) of various operations for 1B objects in saturated systems. put_after(x) includes metadata for x dependencies.

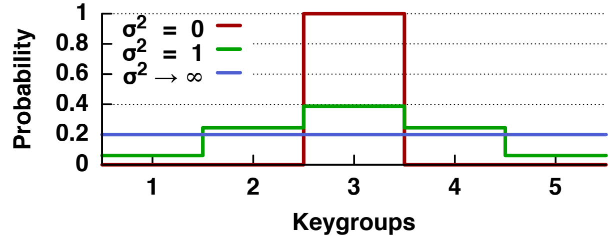


Figure 8: In our experiments, clients choose keys to access by first selecting a keygroup according to some normal distribution, then randomly selecting a key within that group according to a uniform distribution. Figure shows such a stepped normal distribution for differing variances for client #3 (of 5).

the median, and we attribute the few trials with larger throughput differences to the shared nature of the VICCI platform.

6.2 Microbenchmarks

We first evaluate the performance characteristics for COPS and COPS-GT in a simple setting: two datacenters, one server per datacenter, and one colocated client machine. The client sends put and get requests to its local server, attempting to saturate the system. The requests are spread over 2^{18} keys and have 1B values—we use 1B values for consistency with later experiments, where small values are the worst case for COPS (see Figure 11(c)). Table 2 shows the median, 99%, and 99.9% latencies and throughput.

The design decision in COPS to handle client operations locally yields low latency for all operations. The latencies for `get_by_version` operations in COPS and COPS-GT are similar to an end-to-end RPC ping using Thrift. The latencies for `put_after` operations are slightly higher because they are more computationally expensive; they need to update metadata and write values. Nevertheless, the median latency for `put_after` operations, even those with up to 130 dependencies, is around 1 ms.

System throughput follows a similar pattern. `get_by_version` operations achieve high throughput, similar to that of Thrift ping operations (52 vs. 60 Kops/s). A COPS server can process `put_after` operations at 30 Kops/s (such operations are more computationally expensive than gets), while COPS-GT achieves 20% lower throughput when `put_after` operations have 1 dependency (due to the cost of garbage collecting old versions). As the number of dependencies in COP-GT `put_after` operations increases, throughput drops slightly due to the greater size of metadata in each operation (each dependency is ∼12B).

6.3 Dynamic Workloads

We model a dynamic workload with interacting clients accessing the COPS system as follows. We set up two datacenters of S servers each and colocate S client machines in one of the two datacenters. The clients access storage servers in the local datacenter, which replicates `put_after` operations to the remote datacenter. We report the *sustainable* throughput in our experiments, which is the maximum throughput that both datacenters can handle. In most cases, COPS becomes CPU-bound at the local datacenter, and that COPS-GT becomes CPU-bound at the remote one.

To better stress the system and more accurately depict real operation, each client machine emulates multiple logical COPS clients. Each time a client performs an operation, it randomly executes a put or get operation, according to a specified put:get ratio. All operations in a given experiment use fixed-size values.

The key for each operation is selected to control the amount of dependence between operations (i.e., from fully isolated to fully intermixed). Specifically, given N clients, the full keyspace consists of N keygroups, $R_1 \ldots R_N$, one per client. Each keygroup contains K keys, which are randomly distributed (i.e., they do not all reside on the same server). When clients issue operations, they select keys as follows. First, they pick a keygroup by sampling from a normal distribution defined over the N keygroups, where each keygroup has width 1. Then, they select a key within that keygroup uniformly at random. The result is a distribution over keys with equal likelihood for keys within the same keygroup, and possibly varying likelihood across groups.

Figure 8 illustrates an example, showing the keygroup distribution for client #3 (of 5 total) for variances of 0, 1, and the limit approaching ∞. When the variance is 0, a client will restrict its accesses to its "own" keygroup and never interact with other clients. In contrast, when the variance → ∞, client accesses are distributed uniformly at random over all keys, leading to maximal inter-dependencies between `put_after` operations.

The parameters of the dynamic workload experiments are the following, unless otherwise specified:

Parameter	Default	Parameter	Default
datacenters	2	put:get ratio	1:1 or 1:4
servers / datacenter	4	variance	1
clients / server	1024	value size	1B
keys / keygroup	512		

As the state space of all possible combinations of these variables is large, the following experiments explore parameters individually.

Clients Per Server. We first characterize the system throughput as a function of increasing delay between client operations (for two different put:get ratios).[11] Figure 9(a) shows that when the inter-operation delay is low, COPS significantly outperforms COPS-GT. Conversely, when the inter-operation delay approaches several hundred milliseconds, the maximum throughputs of COPS and COPS-GT converge. Figure 9(b) helps explain this behavior: As the inter-operation delay increases, the number of dependencies per operation *decreases* because of the ongoing garbage collection.

[11] For these experiments, we do not directly control the inter-operation delay. Rather, we increase the number of logical clients running on each of the client machines from 1 to 2^{18}; given a fixed-size thread pool for clients in our test framework, each logical client gets scheduled more infrequently. As each client makes one request before yielding, this leads to higher average inter-op delay (calculated simply as $\frac{\text{throughput}}{\text{\# of clients}}$). Our default setting of 1024 clients/server yields an average inter-op delay of 29 ms for COPS-GT with a 1:0 put:get ratio, 11ms for COPS with 1:0, 11ms for COPS-GT with 1:4, and 8ms for COPS with 1:4.

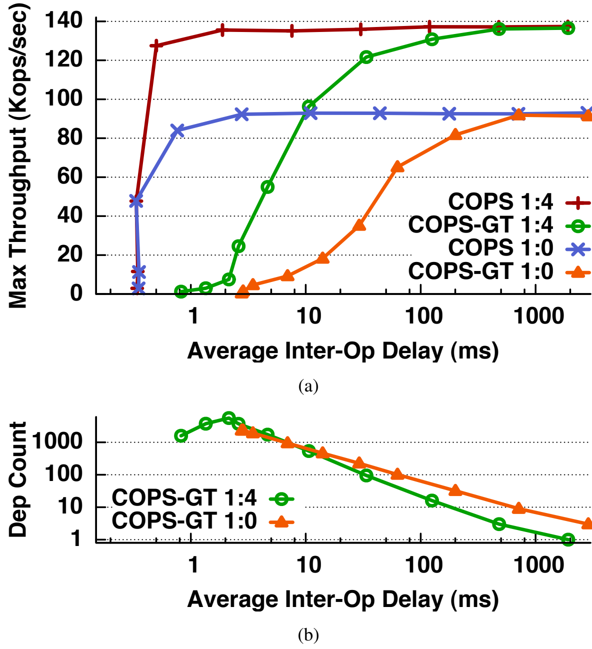


Figure 9: Maximum throughput and the resulting average dependency size of COPS and COPS-GT for a given inter-put delay between consecutive operations by the same logical client. The legend gives the put:get ratio (i.e., 1:0 or 1:4).

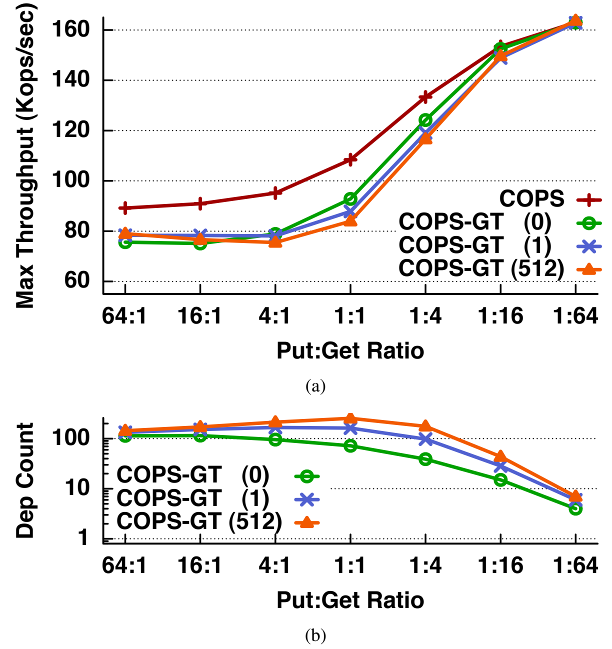


Figure 10: Maximum throughput and the resulting average dependency size of COPS and COPS-GT for a given put:get ratio. The legend gives the variance (i.e., 0, 1, or 512).

To understand this relationship, consider the following example. If the global-checkpoint-time is 6 seconds behind the current time and a logical client is performing 100 puts/sec (in an all-put workload), each put will have $100 \cdot 6 = 600$ dependencies. Figure 9(b) illustrates this relationship. While COPS will store only the single nearest dependency (not shown), COPS-GT must track all dependencies that have not been garbage collected. These additional dependencies explain the performance of COPS-GT: When the inter-put time is small, there are a large number of dependencies that need to be propagated with each value, and thus each operation is more expensive.

The global-checkpoint-time typically lags ~6 seconds behind the current time because it includes both the *trans_time* delay (per Section 5.1) and the time needed to gossip checkpoints around their local datacenter (nodes gossip once every 100ms). Recall that an agreed-upon *trans_time* delay is needed to ensure that currently executing `get_trans` operations can complete, while storage nodes use gossiping to determine the oldest uncommitted operation (and thus the latest timestamp for which dependencies can be garbage collected). Notably, round-trip-time latency *between* datacenters is only a small component of the lag, and thus performance is not significantly affected by RTT (e.g., a 70ms wide-area RTT is about 1% of a 6s lag for the global-checkpoint-time).

Put:Get Ratio. We next evaluate system performance under varying put:get ratios and key-access distributions. Figure 10(a) shows the throughput of COPS and COPS-GT for put:get ratios from 64:1 to 1:64 and three different distribution variances. We observe that throughput increases for read-heavier workloads (put:get ratios < 1), and that COPS-GT becomes competitive with COPS for read-mostly workloads. While the performance of COPS is identical under different variances, the throughput of COPS-GT *is* affected by variance. We explain both behaviors by characterizing the relationship between put:get ratio and the number of dependencies (Figure 10(b)); fewer dependencies translates to less metadata that needs to be propagated and thus higher throughput.

When different clients access the same keys (variance > 0), we observe two distinct phases in Figure 10(b). First, as the put:get ratio decreases from 64:1 to 1:1, the number of dependencies *increases*. This increase occurs because each get operation increases the likelihood a client will inherit new dependencies by getting a value that has been recently put by another client. For instance, if client_1 puts a value v_1 with dependencies d and client_2 reads that value, then client_2's future put will have dependencies on both v_1 and d. Second, as the put:get ratio then decreases from 1:1 to 1:64, the number of dependencies *decreases* for two reasons: (i) each client is executing fewer put operations and thus each value depends on fewer values previously written by this client; and (ii) because there are fewer put operations, more of the keys have a value that is older than the global-checkpoint-time, and thus getting them introduces no additional dependencies.

When clients access independent keys (variance = 0), the number of dependencies is strictly decreasing with the put:get ratio. This result is expected because each client accesses only values in its own keygroup that it previously wrote and already has a dependency on. Thus, no get causes a client to inherit new dependencies.

The average dependency count for COPS (not shown) is always low, from 1 to 4 dependencies, because COPS needs to track only the nearest (instead of all) dependencies.

Keys Per Keygroup. Figure 11(a) shows the effect of keygroup size on the throughput of COPS and COPS-GT. Recall that clients distribute their requests uniformly over keys in their selected keygroup. The behavior of COPS-GT is nuanced; we explain its varying throughput by considering the likelihood that a get operation will inherit new dependencies, which in turn reduces throughput. With the default variance of 1 and a low number of keys/keygroup, most

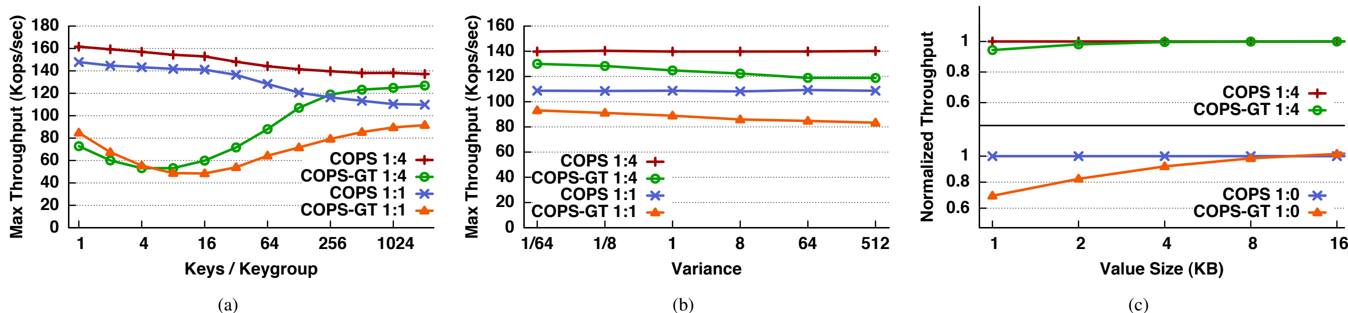


Figure 11: Maximum system throughput (using put:get ratios of 1:4, 1:1, or 1:0) for varied keys/keygroup, variances, and value sizes.

clients access only a small number of keys. Once a value is retrieved and its dependencies inherited, subsequent gets on that same value do not cause a client to inherit any new dependencies. As the number of keys/keygroup begins to increase, however, clients are less likely to get the same value repeatedly, and they begin inheriting additional dependencies. As this number continues to rise, however, garbage collection can begin to have an effect: Fewer gets retrieve a value that was recently written by another client (e.g., after the global checkpoint time), and thus fewer gets return new dependencies. The bowed shape of COPS-GT's performance is likely due to these two contrasting effects.

Variance. Figure 11(b) examines the effect of variance on system performance. As noted earlier, the throughput of COPS is unaffected by different variances: Get operations in COPS never inherit extra dependencies, as the returned value is always "nearer," by definition. COPS-GT has an increased chance of inheriting dependencies as variance increases, however, which results in decreased throughput.

Value Size. Finally, Figure 11(c) shows the effect of value size on system performance. In this experiment, we normalize the systems' maximum throughput against that of COPS (the COPS line at exactly 1.0 is shown only for comparison). As the size of values increases, the relative throughput of COPS-GT approaches that of COPS.

We attribute this to two reasons. First, the relative cost of processing dependencies (which are of fixed size) decreases compared to that of processing the actual values. Second, as processing time per operation increases, the inter-operation delay correspondingly increases, which in turn leads to fewer dependencies.

6.4 Scalability

To evaluate the scalability of COPS and COPS-GT, we compare them to LOG. LOG mimics systems based on log serialization and exchange, which can only provide causal+ consistency with single node replicas. Our implementation of LOG uses the COPS code, but excludes dependency tracking.

Figure 12 shows the throughput of COPS and COPS-GT (running on 1, 2, 4, 8, or 16 servers/datacenter) normalized against LOG (running on 1 server/datacenter). Unless specified otherwise, all experiments use the default settings given in Section 6.3, including a put:get ratio of 1:1. In all experiments, COPS running on a single server/datacenter achieves performance almost identical to LOG. (After all, compared to LOG, COPS needs to track only a small number of dependencies, typically ≤ 4, and any `dep_check` operations in the remote datacenter can be executed as local function calls.) More importantly, we see that COPS and COPS-GT scale well in all scenarios: as we double the number of servers per datacenter, throughput almost doubles.

In the experiment with all default settings, COPS and COPS-GT scale well relative to themselves, although COPS-GT's throughput is only about two-thirds that of COPS. These results demonstrate that the default parameters were chosen to provide a non-ideal workload for the system. However, under a number of different conditions—and, indeed, a workload more common to Internet services—the performance of COPS and COPS-GT is almost identical.

As one example, the relative throughput of COPS-GT is close to that of COPS when the inter-operation delay is high (achieved by hosting 32K clients per server, as opposed to the default 1024 clients; see Footnote 11). Similarly, a more read-heavy workload (put:get ratio of 1:16 vs. 1:1), a smaller variance in clients' access distributions (1/128 vs. 1), or larger-sized values (16KB vs. 1B)—controlling for all other parameters—all have the similar effect: the throughput of COPS-GT becomes comparable to that of COPS.

Finally, for the "expected workload" experiment, we set the parameters closer to what we might encounter in an Internet service such as social networking. Compared to the default, this workload has a higher inter-operation delay (32K clients/server), larger values (1KB), and a read-heavy distribution (1:16 ratio). Under these settings, the throughput of COPS and COPS-GT are very comparable, and both scale well with the number of servers.

7. RELATED WORK

We divide related work into four categories: ALPS systems, causally consistent systems, linearizable systems, and transactional systems.

ALPS Systems. The increasingly crowded category of ALPS systems includes eventually consistent key-value stores such as Amazon's Dynamo [16], LinkedIn's Project Voldemort [43], and the popular memcached [19]. Facebook's Cassandra [30] can be configured to use eventual consistency to achieve ALPS properties, or can sacrifice ALPS properties to provide linearizability. A key influence for our work was Yahoo!'s PNUTS [15], which provides per-key sequential consistency (although they name this per-record timeline consistency). PNUTS does not provide any consistency between keys, however; achieving such consistency introduces the scalability challenges that COPS addresses.

Causally Consistent Systems. Many previous system designers have recognized the utility of causal consistency. Bayou [41] provides a SQL-like interface to single-machine replicas that achieves causal+ consistency. Bayou handles all reads and writes locally; it does not address the scalability goals we consider.

TACT [53] is a causal+ consistent system that uses order and numeric bounding to limit the divergence of replicas in the system. The ISIS [12] system exploits the concept of virtual synchrony [11] to provide applications with a causal broadcast primitive (CBcast).

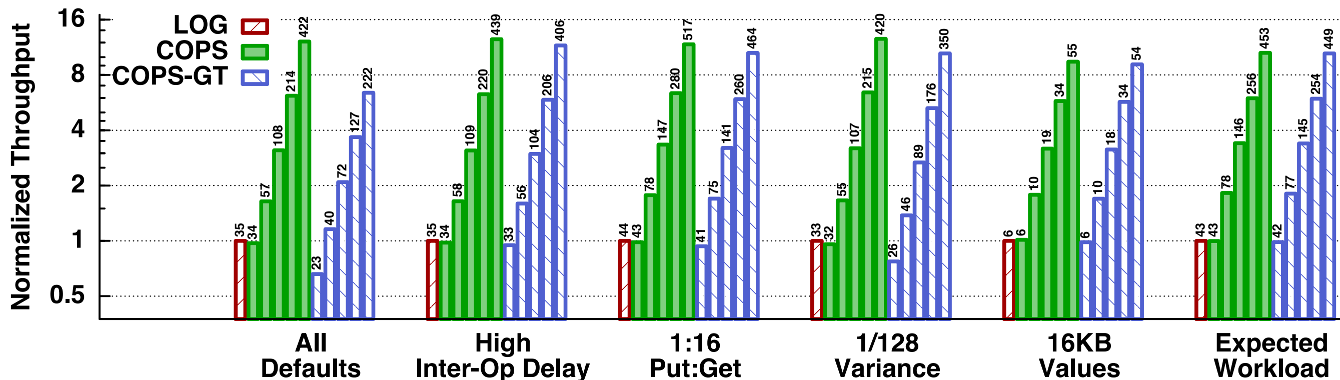


Figure 12: Throughput for LOG with 1 server/datacenter, and COPS and COPS-GT with 1, 2, 4, 8, and 16 servers/datacenter, for a variety of scenarios. Throughput is normalized against LOG for each scenario; raw throughput (in Kops/s) is given above each bar.

CBcast could be used in a straightforward manner to provide a causally consistent key-value store. Replicas that share information via causal memory [2] can also provide a causally consistent ALP key-value store. These systems, however, all require single-machine replicas and thus do not provide scalability.

PRACTI [10] is a causal+ consistent ALP system that supports partial replication, which allows a replica to store only a subset of keys and thus provides some scalability. However, each replica—and thus the set of keys over which causal+ consistency is provided—is still limited to what a single machine can handle.

Lazy replication [29] is closest to COPS's approach. Lazy replication explicitly marks updates with their causal dependencies and waits for those dependencies to be satisfied before applying them at a replica. These dependencies are maintained and attached to updates via a front-end that is an analog to our client library. The design of lazy replication, however, assumes that replicas are limited to a single machine: Each replica requires a single point that can (i) create a sequential log of all replica operations, (ii) gossip that log to other replicas, (iii) merge the log of its operations with those of other replicas, and finally (iv) apply these operations in causal order.

Finally, in concurrent theoretical work, Mahajan et al. [35] define real-time causal (RTC) consistency and prove that it is the strongest achievable in an always-available system. RTC is stronger than causal+ because it enforces a real-time requirement: if causally-concurrent writes do not overlap in real-time, the earlier write may not be ordered after the later write. This real-time requirement helps capture potential causality that is hidden from the system (e.g., out-of-band messaging [14]). In contrast, causal+ does not have a real-time requirement, which allows for more efficient implementations. Notably, COPS's efficient last-writer-wins rule results in a causal+ but not RTC consistent system, while a "return-them-all" conflict handler would provide both properties.

Linearizable Systems. Linearizability can be provided using a single commit point (as in primary-copy systems [4, 39], which may *eagerly* replicate data through two-phase commit protocols [44]) or using distributed agreement (e.g., Paxos [33]). Rather than replicate content everywhere, quorum systems ensure that read and write sets overlap for linearizability [22, 25].

As noted earlier, CAP states that linearizable systems cannot have latency lower than their round-trip inter-datacenter latency; only recently have they been used for wide-area operation, and only when the low latency of ALPS can be sacrificed [9]. CRAQ [48] can complete reads in the local-area when there is little write contention, but otherwise requires wide-area operations to ensure linearizability.

Transactions. Unlike most filesystems or key-value stores, the database community has long considered consistency across multiple keys through the use of read *and* write transactions. In many commercial database systems, a single master executes transactions across keys, then *lazily* sends its transaction log to other replicas, potentially over the wide-area. Typically, these asynchronous replicas are read-only, unlike COPS's write-anywhere replicas. Today's large-scale databases typically partition (or shard) data over multiple DB instances [17, 38, 42], much like in consistent hashing. Transactions are applied only within a single partition, whereas COPS can establish causal dependencies across nodes/partitions.

Several database systems support transactions across partitions and/or datacenters (both of which have been viewed in the database literature as independent *sites*). For example, the R* database [37] uses a tree of processes and two-phase locking for multi-site transactions. This two-phase locking, however, prevents the system from guaranteeing availability, low latency, or partition tolerance. Sinfonia [1] provides "mini" transactions to distributed shared memory via a lightweight two-phase commit protocol, but only considers operations within a single datacenter. Finally, Walter [47], a recent key-value store for the wide-area, provides transactional consistency across keys (including for writes, unlike COPS), and includes optimizations that allow transactions to execute within a single site, under certain scenarios. But while COPS focuses on availability and low-latency, Walter stresses transactional guarantees: ensuring causal relationships between keys can require a two-phase commit across the wide-area. Furthermore, in COPS, scalable datacenters are a first-order design goal, while Walter's sites currently consist of single machines (as a single serialization point for transactions).

8. CONCLUSION

Today's high-scale, wide-area systems provide "always on," low-latency operations for clients, at the cost of weak consistency guarantees and complex application logic. This paper presents COPS, a scalable distributed storage system that provides causal+ consistency without sacrificing ALPS properties. COPS achieves causal+ consistency by tracking and explicitly checking that causal dependencies are satisfied before exposing writes in each cluster. COPS-GT builds upon COPS by introducing get transactions that enable clients to obtain a consistent view of multiple keys; COPS-GT incorporates optimizations to curtail state, minimize multi-round protocols, and reduce replication overhead. Our evaluation demonstrates that COPS and COPS-GT provide low latency, high throughput, and scalability.

Acknowledgments. We owe a particular debt both to the SOSP program committee and to our shepherd, Mike Dahlin, for their extensive comments and Mike's thoughtful interaction that substantially improved both the presentation of and, indeed, our own view of, this work. Jeff Terrace, Erik Nordström, and David Shue provided useful comments on this work; Vijay Vasudevan offered helpful assistance with FAWN-KV; and Sapan Bhatia and Andy Bavier helped us run experiments on the VICCI testbed. This work was supported by NSF funding (CAREER CSR-0953197 and CCF-0964474), VICCI (NSF Award MRI-1040123), a gift from Google, and the Intel Science and Technology Center for Cloud Computing.

REFERENCES

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM TOCS*, 27(3), 2009.

[2] M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, Aug. 2008.

[4] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Conf. Software Engineering*, Oct. 1976.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, Oct. 2009.

[6] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM TOCS*, 14(1), 1996.

[7] Apache Thrift. `http://thrift.apache.org/`, 2011.

[8] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2), 1994.

[9] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, Jan. 2011.

[10] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, May 2006.

[11] K. P. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, Nov. 1987.

[12] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Comp. Soc. Press, 1994.

[13] E. Brewer. Towards robust distributed systems. PODC Keynote, July 2000.

[14] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP*, Dec. 1993.

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

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, Oct. 2007.

[17] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *Knowledge and Data Engineering*, 2(1), 1990.

[18] FAWN-KV. `https://github.com/vrv/FAWN-KV`, 2011.

[19] B. Fitzpatrick. Memcached: a distributed memory object caching system. `http://memcached.org/`, 2011.

[20] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP*, Oct. 1997.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[22] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, Dec. 1979.

[23] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[24] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, Aug. 2009.

[25] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM TOCS*, 4(1), Feb. 1986.

[26] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[27] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[28] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM TOCS*, 10(3), Feb. 1992.

[29] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4), 1992.

[30] A. Lakshman and P. Malik. Cassandra – a decentralized structured storage system. In *LADIS*, Oct. 2009.

[31] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7), 1978.

[32] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computer*, 28(9), 1979.

[33] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[34] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[35] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report TR-11-22, Univ. Texas at Austin, Dept. Comp. Sci., 2011.

[36] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM TOPLAS*, 8(1), Jan. 1986.

[37] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Trans. Database Sys.*, 11(4), 1986.

[38] MySQL. `http://www.mysql.com/`, 2011.

[39] B. M. Oki and B. H. Liskov. Viewstamped replication: A general primary copy. In *PODC*, Aug. 1988.

[40] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3), 1983.

[41] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, Oct. 1997.

[42] PostgresSQL. `http://www.postgresql.org/`, 2011.

[43] Project Voldemort. `http://project-voldemort.com/`, 2011.

[44] D. Skeen. A formal model of crash recovery in a distributed system. *IEEE Trans. Software Engineering*, 9(3), May 1983.

[45] Snappy. `http://code.google.com/p/snappy/`, 2011.

[46] J. Sobel. Scaling out. Engineering at Facebook blog, Aug. 20 2008.

[47] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, Oct. 2011.

[48] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX ATC*, June 2009.

[49] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Conf. Parallel Distributed Info. Sys.*, Sept. 1994.

[50] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Sys.*, 4(2), 1979.

[51] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, Dec. 2004.

[52] VICCI. `http://vicci.org/`, 2011.

[53] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, Oct. 2000.

A. FORMAL DEFINITION OF CAUSAL+

We first present causal consistency with convergent conflict handling (causal+ consistency) for a system with only get and put operations (reads and writes), and we then introduce get transactions. We use a model closely derived from Ahamad et al. [2], which in turn was derived from those used by Herlihy and Wing [26] and Misra [36].

Original Model of Causal Consistency [2] with terminology modified to match this paper's definitions:

A system is a finite set of threads of execution, also called threads, that interact via a key-value store that consists of a finite set of keys. Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of threads. The local history L_i of a thread i is a sequence of get and put operations. If operation σ_1 precedes σ_2 in L_i, we write $\sigma_1 \underset{i}{\rightarrow} \sigma_2$. A history $H = \langle L_1, L_2, \ldots, L_n \rangle$ is the collection of local histories for all threads of execution. A serialization S of H is a linear sequence of all operations in H in which each get on a key returns its most recent preceding put (or $\perp$ if there does not exist any preceding put). The serialization S respects an order $\rightarrow$ if, for any operation σ_1 and σ_2 in S, $\sigma_1 \rightarrow \sigma_2$ implies σ_1 precedes σ_2 in S.

The *puts-into* order associates a put operation, `put`(k,v), with each get operation, `get`(k)=v. Because there may be multiple puts of a value to a key, there may be more than one puts-into order.[12] A puts-into order $\mapsto$ on H is any relation with the following properties:

- If $\sigma_1 \mapsto \sigma_2$, then there is a key k and value v such that operation σ_1 := `put`(k,v) and σ_2 := `get`(k)=v.
- For any operation σ_2, there exists at most one σ_1 for which $\sigma_1 \mapsto \sigma_2$.
- If σ_2 := `get`(k)=v for some k,v and there exists no σ_1 such that $\sigma_1 \mapsto \sigma_2$, then v = $\perp$. That is, a get with no preceding put must retrieve the initial value.

Two operations, σ_1 and σ_2, are related by a causal order $\rightsquigarrow$ if and only if one of the following holds:

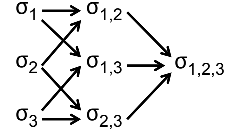
- $\sigma_1 \underset{i}{\rightarrow} \sigma_2$ for some t_i (σ_1 precedes σ_2 in L_i);
- $\sigma_1 \mapsto \sigma_2$ (σ_2 gets the value put by σ_1); or
- There is some other operation σ' such that $\sigma_1 \rightsquigarrow \sigma' \rightsquigarrow \sigma_2$.

Incorporating Convergent Conflict Handling. Two operations on the same key, σ_1:=`put`(k,v_1) and σ_2:=`put`(k,v_2), are in *conflict* if they are not related by causality: $\sigma_1 \not\rightsquigarrow \sigma_2$ and $\sigma_2 \not\rightsquigarrow \sigma_1$.

A *convergent conflict handling function* is an associative, commutative function that operates on a set of conflicting operations on a key to eventually produce one (possibly new) final value for that key. The function must produce the same final value independent of the order in which it observes the conflicting updates. In this way, once every replica has observed the conflicting updates for a key, they will all independently agree on the same final value.

We model convergent conflict handling as a set of *handler* threads that are distinct from normal client threads. The handlers operate on a pair of conflicting values (v_1, v_2) to produce a new value *newval* = $h(v_1, v_2)$. By commutativity, $h(v_1, v_2) = h(v_2, v_1)$. To produce the new value, the handler thread had to read both v_1 and v_2 before putting the new value, and so *newval* is causally ordered after both original values: $v_1 \rightsquigarrow$ *newval* and $v_2 \rightsquigarrow$ *newval*.

With more than two conflicting updates, there will be multiple invocations of handler threads. For three values, there are several possible orders for resolving the conflicting updates in pairs:



The commutativity and associativity of the handler function ensures that regardless of the order, the final output will be identical. Further, it will be causally ordered after all of the original conflicting writes, as well as any intermediate values generated by the application of the handler function. If the handler observes multiple pairs of conflicting updates that produce the same output value (e.g., the final output in the figure above), it must output only one value, not multiple instances of the same value.

To prevent a client from seeing and having to reason about multiple, conflicting values, we restrict the put set for each **client** thread to be conflict free, denoted p_{cf_i}. A put set is *conflict free* if $\forall \sigma_j, \sigma_k \in p_{cf_i}$, σ_j and σ_k are not in conflict; that is, either they are puts to different keys or causally-related puts to the same key. For example, in the three conflicting put example, p_{cf_i} might include σ_1, $\sigma_{1,2}$, and $\sigma_{1,2,3}$, but not σ_2, σ_3, $\sigma_{1,3}$, or $\sigma_{2,3}$. The conflict-free property applies to client threads and not handler threads purposefully. Handler threads must be able to get values from conflicting puts so they may reason about and resolve them; client threads should not see conflicts so they do not have to reason about them.

Adding handler threads models the new functionality that convergent conflict handling provides. Restricting the put set strengthens consistency from causal to causal+. There are causal executions that are not causal+: for example, if σ_1 and σ_2 conflict, a client may get the value put by σ_1 and then the value put by σ_2 in a causal, but not causal+, system. On the other hand, there are no causal+ executions that are not causal, because causal+ only introduces an additional restriction (a smaller put set) to causal consistency.

If H is a history and t_i is a thread, let $A^H_{i+p_{cf_i}}$ comprise all operations in the local history of t_i, and a conflict-free set of puts in H, p_{cf_i}. A history H is *causally consistent with convergent conflict handling* (causal+) if it has a causal order $\rightsquigarrow$, such that

Causal+: For each *client* thread of execution t_i, there is a serialization S_i of $A^H_{i+p_{cf_i}}$ that respects $\rightsquigarrow$.

A data store is *causal+ consistent* if it admits only causal+ histories.

Introducing Get Transactions. To add get transactions to the model, we redefine the puts-into order so that it associates N put operations, `put`(k,v), with each get transaction of N values, `get_trans`([$k_1, \ldots, k_N$])=[$v_1, \ldots, v_N$]. Now, a puts-into order $\mapsto$ on H is any relation with the following properties:

- If $\sigma_1 \mapsto \sigma_2$, then there is a k and v such that σ_1 := `put`(k,v) and σ_2 := `get_trans`([...,k,...])=[...,v,...]. That is, for each component of a get transaction, there exists a preceding put.
- For each component of a get transaction σ_2, there exists at most one σ_1 for which $\sigma_1 \mapsto \sigma_2$.
- If σ_2 := `get_trans`([...,k,...])=[...,v,...] for some k,v and there exists no σ_1 such that $\sigma_1 \mapsto \sigma_2$, then v=$\perp$. That is, a get with no preceding put must retrieve the initial value.

[12]The COPS system uniquely identifies values with version numbers so there is only one puts-into order, but this is not necessarily true for causal+ consistency in general.

Appeared in ***Proc. 41st IEEE/IFIP International Conference on Dependable Systems and Networks, Dependable Computing and Communication Symposium (DCCS) track (DSN 11)***

Coercing Clients into Facilitating Failover for Object Delivery

Wyatt Lloyd, Michael J. Freedman
Princeton University

***Abstract*—Application-level protocols used for object delivery, such as HTTP, are built atop TCP/IP and inherit its host-to-host abstraction. Given that these services are replicated for scalability, this unnecessarily exposes failures of individual servers to their clients. While changes to both client and server applications can be used to mask such failures, this paper explores the feasibility of transparent recovery for *unmodified* object delivery services (TRODS).**

The key insight in TRODS is cross-layer visibility and control: TRODS carefully derives reliable storage for application-level state from the mechanics of the transport layer. This state is used to reconstruct object delivery sessions, which are then transparently spliced into the client's ongoing connection. TRODS is fully backwards-compatible, requiring no changes to the clients or server applications. Its performance is competitive with unmodified HTTP services, providing nearly identical throughput while enabling timely failover.

I. Introduction

Ideally, a client's interaction with a replicated service will fail only when the service fails. Yet most Internet services tie the fate of a client's connection to a single server, because they are built using TCP and inherit its host-to-host bindings. If this single server fails, the client's connection breaks, and it appears to the client that the service has failed. However, if a new server can transparently *failover* the connection—that is, interact with the client exactly as the original server would have—the client's connection can continue uninterrupted and unaware of the failure.

We aim to enable failover for a large class of Internet services, called *object delivery services*, that play an integral role in users' online experiences by giving clients read-only access to content objects, such as webpages, images, and videos. Object delivery services are typically replicated for scalability and fault-tolerance, e.g., there are tens to thousands of servers that all deliver the same set of objects. If one such server fails while delivering an object, another server has the potential to continue delivering it. This paper demonstrates that such recovery can be done transparently, effectively, and practically.

Our system, Transparent Recovery for Object Delivery Services (TRODS), has been designed with the goal of *immediate deployability*, which introduces two challenges. Clients of the service should not be modified: They are often not under the service's control and often run different applications, browsers, and operating systems. Similarly, the server's application code should not be modified: Source code may be unavailable, and application changes would require integration effort for every service that seeks failover. Instead, TRODS is implemented as a server-side kernel module and requires no changes to the client or application.

At a high level, TRODS operates by ensuring that, at failover time, a recovery server has the minimal application-level information necessary to continue a connection. This information is preserved in two ways. First, it can be retransmitted by the client to its recovery server. TRODS does not modify the client to accomplish this, instead, it leverages its on-path position within the server's kernel to manipulate a connection's TCP packets, in order to coerce the client into retransmitting the information to the new server. Second, the information can be saved to a persistent store that will survive the failure of the original server.

We describe two complementary versions of TRODS that use different resources as persistent stores. The first version, TRODS-KV, uses a key-value store for persistence. It improves on previous failover schemes by requiring only a *single* remote operation apart from the original server—a single save to the key-value store—to guarantee any subsequent connection failover. The second version, TRODS-TS, eliminates the need for *any* remote operations by carefully repurposing the TCP timestamp option that accompanies every packet in a connection as the persistent store. These two approaches are complementary: TRODS-KV is more general purpose, handles more abnormal object delivery scenarios, and avoids some additional security concerns. On the other hand, TRODS-TS has very low overhead and requires no additional physical resources for deployment. Together, TRODS-TS can serve the highly-popular objects of a service, while TRODS-KV can handle the unpopular and exceptional cases.

This paper focuses on the use of HTTP as the canonical and ubiquitous protocol for object delivery. However, we believe that TRODS' approach is similarly applicable to other protocols for object delivery.

TRODS has significantly lower overhead than previous transparent failover schemes. Several of these schemes require primary and backup servers to process requests in parallel, e.g., FT-TCP (hot backup) [24] and ST-TCP [14]. This redundant processing reduces the systems' throughput per machine by at least 50%. Other prior schemes that

Copyright notice: 2011 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

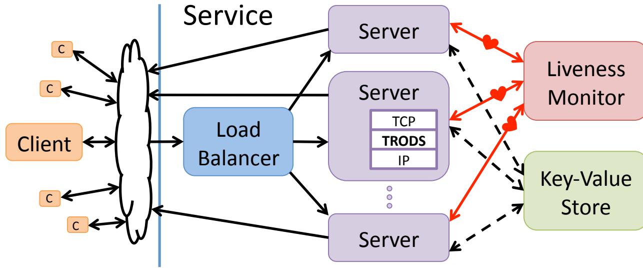


Figure 1. **A typical service architecture that uses TRODS.**

avoid an active backup—e.g., FT-TCP (cold backup) and CoRAL [1]—still require many remote operations to save state so it can be replayed at recovery time. In contrast, TRODS-KV needs only a single remote operation and TRODS-TS eliminates them altogether.

II. Protocol

This section gives a high-level overview of how TRODS operates. We begin by describing how TRODS fits into the architecture of a typical Internet service. Next, we examine the structure of a connection to an object delivery service. Finally, we detail how TRODS can failover a client's connection during each of its phases.

A. Architecture

Figure 1 shows the architecture of a typical Internet service using TRODS. Every component is standard and, excluding the key-value store, would likely be found in a TRODS-free version of the service. The clients are unmodified, run their normal networking stack and applications, and connect to the service using TCP. An unmodified load balancer routes all packets in a connection to the same server. A liveness monitor maintains the load balancer's pool of available servers. TRODS does not need a stateful load balancer, so any stateless flow-based hashing suffices, e.g., consistent hashing [9] or standard *mod n* hashing using the flow's 5-tuple for server affinity. The servers terminate the clients' connections and include a TRODS kernel module that sits in their network stacks. This presence allows TRODS to manipulate and control the packets bidirectionally. Figure 1 also includes a key-value store, which TRODS-KV uses as a persistent store, as discussed later.

This figure illustrates one concrete example of a service architecture that uses TRODS. TRODS can work for any general object delivery service that meets three requirements: it is comprised of replicated servers that serve static objects, its servers can all use the same IP address(es), and it has an updatable load balancer.

In this paper, we do not consider the failure of the load-balancer or the liveness monitor. Both can be supported by standard replication and failover techniques, and their state need be linear only in the number of servers, not the number of flows. Further, unlike typical deployments, TRODS can tolerate inconsistent state between load-balancers, e.g., in their known set of live servers. If one sends the subsequent packets of an existing connection to a new server, the connection is dealt with as a normal case of failover. We do consider the failure of the key-value store in §IV-A.

B. The Anatomy of an ODS Connection

To clarify how TRODS enables failover, we first identify the two stages of a connection to an object delivery service. The first phase is connection setup, where the client and server negotiate what object the client is going to download. In HTTP, for example, this constitutes the HTTP GET request and the server's response header. The second phase of the connection is the object download. In HTTP, this corresponds to the transmission of the response body.

Transparent failover requires a new server to continue a client's interaction with the service over its pre-existing TCP connection. If the client is in the setup phase, the new server should continue negotiating with the client and then start the object delivery. If the client is in the download phase, the new server should continue the client's download exactly where the old server left off.

The two phases of a connection are quite different. The setup phase is typically short, in terms of both bytes and packets, and application-layer data flows in both directions. The download phase can be long, and application-layer data only flows from the server to the client. Accordingly, TRODS handles failover for each phase quite differently.

C. Failover

TRODS takes the following steps to failover a client's connection on a new server:

1) Detect a server failure.
2) Redirect the client's connection to a new server.
3) Initiate failover on the new server.
4) Determine the connection's current phase.

If in the setup phase:

5) Continue negotiating with the client.

If in the download phase:

5) Determine what object the client is downloading.
6) Determine the client's current offset into the object.
7) Resume sending the object from that point.

Failure Detection. To detect server failures, we apply standard unreliable failure detection [3]. Periodically, a liveness monitor sends a heartbeat packet to each server and each server responds with their own packet. The server is determined to have failed if the liveness monitor does not hear from a server for longer than a threshold amount of time (25 ms in our implementation). This scheme detects hardware failures, but not necessarily application failures. Our implementation uses its position in the kernel of each host to locally detect application failures (e.g., process crashes). It then prevents the machine from exposing those failures to the client (e.g., the TRODS module drops RST packets arising in such scenarios), while also triggering failover by the liveness monitor.

Connection Redirection. Connections to the failed server must be rerouted to new servers so that failover can begin. Once the liveness monitor detects a server has failed or a new one has started, it updates the load balancer's state about the pool of active servers and their corresponding MAC addresses. The load balancer will then start routing packets to the new set of servers. Now, all new connections will be handled by a live server.[1]

The choice of load-balancing scheme affects how ongoing connections are remapped to servers. If consistent hashing [9] is used, only connections to the failed server will be reassigned elsewhere. By contrast, if a less smooth hashing function is used—such as selecting a server by randomly hashing *mod* the server-pool size—then almost all ongoing connections will be reassigned to new servers. While more disruptive, TRODS still handles this scenario, treating such reassignments as normal cases of failover.

Failover Initiation. After a load-balancer redirects a connection, the new server will receive any packets the client sends. The new server will recognize that these packets are in the middle of a TCP connection that does not exist on this server and thus must be failed over. While there will often be outstanding packets in the network when a server fails—especially given the large TCP window size of an ongoing download—TRODS cannot rely on these packets either to exist or to arrive at the new server in order to initiate failover. Instead, TRODS ensures the client will send a packet that reaches the new server by leaving at least one packet from the client unacknowledged at all times, coercing its TCP stack to continue to retransmit it. Fortunately, this does not affect the application-layer connection, as the TCP specification allows the client to receive the server's application-layer response, even when its request has not been acknowledged at the transport layer.

Determining the Current Phase. TRODS requires some state to be shared between a connection's original and recovery servers, in order to accurately determine the current phase of the connection. TRODS accomplishes this by blocking a connection from entering the download phase until it has saved some information to a *persistent store* that will survive the failure of the original server. When a new server starts to failover a connection, it first looks up the connection in the persistent store. If the connection is not found, the new server knows the connection is still in the setup phase; otherwise, it is in the download phase. We discuss the corner cases of phase determination in §III.

Continuing Negotiations. If the connection is in the setup phase, the new server must continue the negotiation with the client. Negotiation is stateful, which might suggest that TRODS needs to save already-negotiated state to the persistent store, in order to continue negotiation after failover. However, TRODS exploits the short length of the setup phase to avoid this.

Because setup differs between protocols, TRODS deals with each uniquely. The common theme is that TRODS uses control of the TCP layer to effectively coerce the client into providing storage unbeknownst to it. In HTTP, for example, TRODS does not acknowledge the client's request until after the client has entered the download phase. Thus, if a server failure occurs during the setup phase, the client's TCP stack will timeout and retransmit the request so a new server can handle it. Here, TRODS again exploits the separation between application-layer data and TCP-layer acknowledgments, which allows a client's application to operate normally while its transport layer attempts to retransmit packets. We discuss further details in §III.

Determining the Object. To continue a connection in the download phase, a new server needs to determine both the object being downloaded and the client's offset into that object. We assume that each service object will have an *objectID*, a unique, concise identifier of the object, such as a filename or URL. We further assume that all objects are immutable, we have omitted the discussion of TRODS' use with versioned and dynamic objects due to space constraints. Thus, if a new server knows the objectID associated with a connection, it knows the object the client is downloading. TRODS makes this objectID available to the new server by persistently storing it.

Determining the Client's Offset. Once TRODS has determined which object a client is downloading, it still needs to determine how far into the download the failure occur. TRODS derives this offset by again leveraging cross-layer information. TRODS compares the *objectISN*—the TCP sequence number for the first byte of the object download, which had been saved earlier to the persistent store—and the most recent TCP sequence number the client has acknowledged. The difference between these two values gives the client's current offset into the object; all preceding bytes have been successfully received at the client.

Resuming Object Downloads. Once TRODS knows the objectID and offset for a connection, it must transfer the object, starting at this offset, from an application running on the new server. TRODS accomplishes this by initiating a new local connection to the application, and using the objectID to synthesize an application-level request for the client's object. It quickly acknowledges and discards the downloading object until the client's current offset is reached, at which point it begins transmitting the data from the server application to the client. In many applications, this initial "discard" phase can be avoided by requesting the client's offset directly, e.g., through `Range-Request` headers in HTTP.

[1] The handling of new connections is what load-balancer products and high-availability software packages refer to as failover. In these systems, unlike in TRODS, ongoing TCP connections remapped to a different server will be unable to continue.

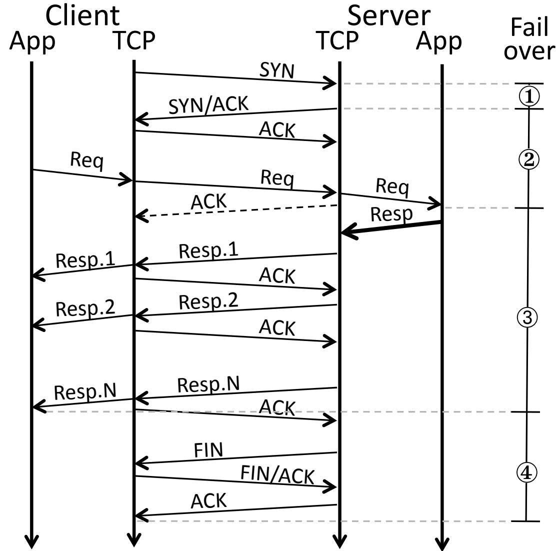


Figure 2. **A typical client-server HTTP connection at both the application and TCP layers. The dashed acknowledgment for the client's request is sent by the server's TCP stack, but dropped by TRODS. The right-most "failover" label indicates a stage of the connection; we detail how TRODS handles failover for each in §III-B.**

III. TRODS For HTTP

While TRODS provides a general framework for performing failover, it does require a mechanism for extracting a connection's objectID and objectISN, which typically requires application-specific parsing. In HTTP, for example, this objectID is commonly the request URL, while the objectISN is the first byte of the HTTP response body. In our TRODS prototype, this application-specific HTTP knowledge constitutes about 100 lines of code. For concreteness, this section details TRODS' handling of HTTP connections.

We start by exploring how TRODS handles a normal connection at a packet-by-packet level. We then show how this behavior allows TRODS to failover that connection to a new server for all possible connection states.

We make the these assumptions for a typical connection:

1) The request fits in a single packet.
2) The response header fits in a single packet.
3) The response body is less than 4 GB in size.
4) The object download takes less than 13 minutes.
5) Neither persistent nor pipelined connections are used.
6) HTTP chunked transfer encoding is not used.

The first four assumptions hold true for the majority of HTTP connections, and TRODS takes advantage of them to improve performance. The next two assumptions simplify the basic description of TRODS. We complete our specification by relaxing each assumption in §III-C.

A. Normal Operation

Figure 2 shows a HTTP connection at both the application and transport layers, and Table I briefly summarizes how TRODS interacts with this connection from its position underneath the server's TCP layer.

Cli	Srv	TRODS Operation
Syn		
	Syn	Locally store knowledge of this connection
Ack		
Req		Extract and locally save objID
	Ack	Drop
	Resp_1	Extract objISN Block until objID/objISN are persistently stored Do not ack client's request
Ack		
	Resp_2	Do not ack client's request
Ack		
…	…	
	Fin	Locally store sequence number of FIN
Fin/ Ack		Delete objID/objISN from persistent storage Delete connection from local storage
	Ack	Ack client's request and Fin

Table I
Normal operation during a typical HTTP connection.

The connection begins with TCP's three-way handshake. During the handshake when the server sends a response SYN packet, TRODS locally stores knowledge of this connection by saving the client's IP address and port into an in-memory hashtable. This allows TRODS to distinguish between normal packets to the server, whose connections will be in the hashtable, and packets that should initiate failover, whose connections will not be in the hashtable because they originated at another server.

The connection continues with the client sending a HTTP request that fits in a single packet. From this request, TRODS extracts the objectID from the packet, which normally consists of the URI.[2] Under normal processing, the server's transport layer immediately responds to receiving this request with an ACK; TRODS instead drops this packet. If TRODS did not drop this ACK and the server failed after acknowledging the request, but before persistently storing anything, the client's requested objectID would be lost.

The application server then attempts to send the client a response. This response is often too large to fit in a single packet, so the TCP stack on the server distributes it over many TCP segments. The first segment (and packet) will include the response header and the beginning of the response body. TRODS determines the objectISN—the sequence number of the first byte of the response body—by searching through the HTTP payload for the double CRLF that delineates the end of the HTTP response header. TRODS saves the objectISN and objectID to the persistent store, before releasing the TCP stack to transmit the packets back to the client. TRODS also modifies all packets that carry the response to not acknowledge the client's request. This ensures that if the server fails, the client's TCP stack will eventually retransmit a failover-initiating packet.

[2]The objectID may also include some HTTP request headers, such as cookies. If only the URI is used when other headers affect the server's response, the interaction can appear to be non-deterministic, which we omit discussion of due to space constraints.

After the server's TCP stack has transmitted the entire response to the client, it sends a FIN packet to start tearing down the connection. TRODS stores the TCP sequence number for the FIN in its local hashtable, to help it later determine if the client has received the entire response. The client will respond to the server's FIN with a FIN/ACK of its own; TRODS checks that this acknowledges the server's FIN, and then knowing the client has received the entire response, deletes the connection from the persistent store and local hashtable. The connection terminates when the server sends the client an ACK that cumulatively acknowledges the client's request and FIN.

Deleting connection information from the persistent store is performed to reduce saved state, not to maintain correctness. Thus, it can be done in the background or during periods of low-server load; it does not delay the connection.

B. Failure Recovery

Figure 2 groups the different stages of a connection into failover cases. We now enumerate these stages, showing how TRODS provides failover in each case.

Before Setup ①. A server can fail after receiving a client's SYN but before responding with a SYN/ACK. If this happens, the client's TCP stack times out and retransmits its SYN. This SYN will be routed to a new server and the connection will proceed normally. If a server fails after issuing a SYN/ACK but the network drops the packet, the system's behavior is identical. In later cases, we do not discuss drops that are equivalent to scenarios without them.

During Setup ②. A server can fail after the client receives the SYN/ACK but before the server sends the response. Because the client's request remains unacknowledged, the client's TCP stack will eventually timeout and retransmit the request. The load balancer will direct this request to a new server, which will initiate failover.

On the new server, TRODS will lookup the client in the persistent store. If the lookup succeeds, the connection is currently in the download phase and is recovered as described in ③. If the lookup fails, the client is still in the setup phase and has not received any part of the response yet. TRODS will then open a TCP connection to the new application server on the localhost and proceed with TCP's three-way handshake. Once the connection is established, TRODS will *splice* together this new connection and the client's connection.[3] The request will then be forwarded to the server and the connection will proceed normally.

During Download ③. If a server fails during the download phase of a connection, the client's TCP stack will eventually timeout and retransmit the packet TRODS purposefully did not acknowledge. This packet, or one that was in the network when the server failed, can be combined with the information in the persistent store to find the objectID and objectISN for this connection.

The new TRODS instance that receives this packet will start a connection with the local application instance, sending a request for the object constructed from the objectID. If supported, this request includes a `Range-Request` header, indicating that the application server should start transmitting the object at the client's current offset. The server will respond with a new response header and the object starting at the specified offset. TRODS drops the response header, and it splices together this connection with the client's original one.

After Download ④. If the server fails after the client finishes downloading the object, but before TRODS deletes history of the connection from the persistent store, TRODS might attempt failover as in ③. However, the new server's HTTP response will be an error (status code 416), as the range request specified an offset that is one byte past the end of the object. TRODS will recognize that the client has completed the download, drop the server's response, close the connection to the server, delete the client's connection from the persistent store, and, if the client's packet was a FIN, respond to the client with an ACK.

Some packets from a client may be delayed by the network and not arrive until after the connection has completed and TRODS has removed it from its local hashtable. If this occurs, TRODS will attempt to failover the connection. However, as the client has already completed its download and closed the connection, it will respond to any new packets from the server with a RST packet. TRODS forwards this RST to the server, closing the newly-established connection. While this does not affect the correctness of TRODS, it does waste server resources. We describe how to restrict these wasted failover attempts in §V-A, so that they only occur when it is plausible that their original server has failed.

C. Extensions

For brevity, we omit the detailed explanations of how TRODS handles HTTP connections that violate our assumptions described earlier. Instead, we briefly sketch the main ideas for dealing with any violations. If the request is spread across multiple packets—a rare event for GET requests—TRODS persistently stores each packet before allowing its corresponding acknowledgment to flow back to the client. TRODS handles multi-packet response headers similarly, by saving them in their entirety to the persistent store before allowing them to flow to the client. If an object is over 4GB the TCP sequence numbers will wrap around so TRODS uses separate objectIDs for different sections of the object. If an object download takes more than 13 minutes, the client's connection must be acknowledged to prevent its TCP stack from resetting the connection. TRODS does acknowledge these rare connections, and then saves them to a list in the key-value store for special handling. TRODS handles persis-

[3] TCP splicing joins two separate connections together so that they act as one; it is accomplished by translating the IP addresses, port numbers, and sequence numbers in every packet.

tent and pipelined connections by splitting apart any packets that include data for multiple objects. Chunked-encoding may only be used if it is deterministic across replicas, as its in-line metadata of chunk lengths prohibit TRODS' simple determination of the client's application-level offset into the response object. We have verified that lighttpd's static file and flash video modules are deterministic by examining their source code and expect that most other chunking schemes are as well.

IV. Persistent Storage

The TRODS protocol refers opaquely to a "persistent store" that assists with saving connection state necessary for failover. This store is *persistent* in that it survives the failure of the original server. In this section, we describe the two persistent stores we implemented.

A. Key-Value Store

The first persistent store is a key-value storage system (e.g., memcached [5]). The storage key that TRODS uses for each connection is comprised of the client's IP address and port number. The key-value store can be used for arbitrarily-sized objects, which is not true for TCP Timestamps. Thus, if a large store is needed—e.g., when multiple response header packets need to be stored before being sent to the client—TRODS uses the key-value store.

The configuration and deployment of the key-value store trades off efficiency and availability. Key-value storage servers can be colocated in the same rack, cluster, or data-center as application servers. As the key-value store moves closer to its application servers, latency decreases but the probability of correlated failure increases. Data in the key-value store can be replicated for additional fault-tolerance, but even unreplicated storage provides resilience to a single failure: A connection fails only when its application and key-value server fail simultaneously. For this reason, many deployments may choose an in-memory key-value store (e.g., memcached [5]) for low latency and high throughput.

B. TCP Timestamps

The second persistent store is the TCP timestamp option [8] that accompanies every packet in a connection. Failover in TRODS is always initiated by a packet from the client, which is what makes this store *persistent*. The TCP timestamp option is negotiated during connection setup: Each host attaches the TCP timestamp option to its SYN packet. Once negotiated, each host will attach its own 4-byte timestamp value and a 4-byte timestamp echo reply to every packet. The timestamp echo reply effectively repeats the last timestamp value that a host received. The use of the TCP timestamp option is widespread: It is used by default in modern versions of Linux, FreeBSD, OS X, and Windows. In the rare event that a host does not use the option, TRODS can fall back to its key-value store for persistent storage.

TCP timestamps were intended for two purposes. First, they help improve the accuracy of RTT estimation. A host will subtract the timestamp echo reply in an ACK packet from the current time to obtain a new RTT. This allows the host to accurately sample the RTT at a high rate and is "vitally important" for large TCP window sizes [8]. Thus, when co-opting the TCP timestamp option as persistent storage, TRODS must ensure that it does not interfere with accurate RTT measurement.

Second, the TCP timestamp option helps protect against wrapping sequence numbers (PAWS). PAWS is used to prevent old duplicate segments from a previous connection from corrupting a current connection between the same hosts using exactly the same ports. This will only happen if (1) a client reconnects to the same server in a short window of time (less than 2 maximum segment lifetimes, or about 4 minutes); and (2) in between these connections, the client makes some number of other connections that is an exact multiple of its ephemeral port range.[4] This is sufficiently unlikely that TRODS does not handle this possibility. However, because the client cannot be changed, TRODS' use of the timestamp must not interfere with the client's PAWS processing. To enforce PAWS, the client will drop all packets with a server timestamp that is deemed too "old". TRODS ensures timestamps are non-decreasing in modular 32-bit space,[5] so they will be accepted.

To summarize, the TCP timestamp option provides 32 bits that the client will echo back with two constraints: The timestamps must be non-decreasing in modular 32-bit space and they still must provide accurate RTT measurement. These 32 bits cannot naively hold the objectID and objectISN: The objectISN alone is 32 bits and the objectID has been unconstrained until now. Thus, TRODS must reduce the number of bits needed for the objectID and objectISN to fit in the TCP timestamp, while obeying these constraints.

5 Bits for the ObjectISN. The objectISN can be derived by summing two values: the TCP connection's initial sequence number (ISN) and the length of the response header. TRODS uses this property, as well as small changes at the TCP and HTTP levels, to store the objectISN in 5 bits rather than 32.

At the TCP level, we fix the connection's ISN to a value derived from the client's IP and port. This avoids needing any bits to store the connection's ISN, but raises some security concerns that we address in §V-C.

If the response header is longer than a TCP segment size (typically 1448 bytes with the TCP Timestamp option), then the entire response needs to be stored in the key-value store. Consequently, we only consider response headers that are less than 1448 bytes. Storing its length still requires $\lceil \lg 1448 \rceil = 11$ bits. However, TRODS uses an HTTP-level optimization to reduce this further: It pads the response header to a multiple of 64 bytes, which reduces the number of bits needed to $\lceil \lg \lceil 1448/64 \rceil \rceil = 5$. TRODS pads the

[4] The smallest ephemeral port range we could find was 3975 [23].

[5] That is, $ts_a \geq ts_b$ when $0 \leq (ts_a - ts_b) < 2^{31}$ in unsigned 32-bit math.

header by adding linear white space to the last header field, which HTTP clients ignore [4]. Our choice to pad to 64-byte multiples is arbitrary; we could pad to 128 bytes and then only need 4 bits for the response length.

When TRODS pads the header, it misaligns the TCP sequence number space between the client and server: The client has now received more bytes that the server has sent. TRODS modifies the sequence numbers in all subsequent packets to correct for this difference.

7 Bits for the Timestamp. TRODS ensures accurate RTT measurement by passing packets to the server's TCP layer with the appropriate timestamps replaced. When the TCP layer passes TRODS a packet for transmission, TRODS saves the timestamp in a per-connection 128-entry array. It then overwrites the packet's original timestamp with its own value that includes a 7-bit index into that connection's array. When TRODS receives a packet to pass up to the local TCP stack, it uses the 7-bit index embedded in its own timestamp to look up the origin timestamp, which it swaps in before sending the packet up the stack.

The use of a 7-bit index limits the number of outstanding timestamps to 128, and TRODS blocks packets to stay under this limit. With a normal MSS size of 1448 bytes, this means at most 185 KB can be in flight from the server at any point (e.g., a connection with a 50 ms RTT could download at 30 Mbps). This behavior seems reasonable for most web services, but if this limit is too low for a particular service, it can increase the size of the array and bit-length of the index, at the cost of requiring either further response header padding or supporting fewer objectIDs.

20 Bits for the ObjectID. The objectID is represented by a long, unique string, such as a file path or full URL. This objectID cannot be embedded in a timestamp, so TRODS instead embeds a shorter index that selects from an array of objectIDs. This array is normally static and replicated on each server. With 20 bits, TRODS can uniquely identify over one million objects. If a service has more objects than can fit in the array, it can use the timestamp option as the persistent store for its most popular million objects and a key-value store for less popular objects. Given the Zipfian nature of Web traffic [2], the million popular objects that can use the TCP timestamp option should cover the majority of a service's traffic. TRODS can also consistently update this array to account for new or newly popular objects, but we omit a description of this behavior for brevity.

Ordering the Fields. Finally, TRODS orders its fields in the timestamp option carefully, as shown in Figure 3, to ensure they pass the client's PAWS check by being non-decreasing in modular 32-bit space. The timestamp index resides in the highest-order bits, followed by the objectISN, while the objectID resides in the lowest-order bits. The objectID and objectISN field do not change once set, but the timestamp index does: it increases and eventually wraps around. By

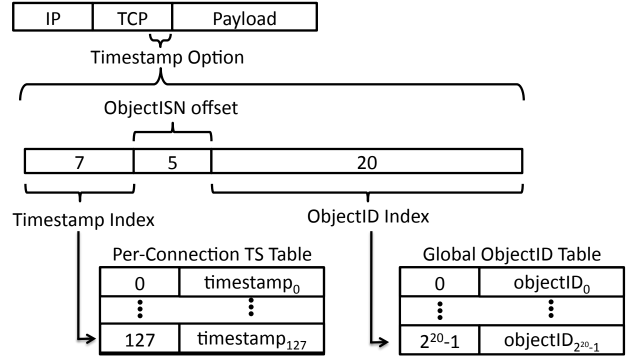


Figure 3. **The relationship between a packet, its TCP timestamp option, the fields TRODS shoehorns into that option, the per-connection timestamp table, and the global objectID table.**

placing it in the high-order bits, TRODS ensures that when it wraps around, the numerical representation of the timestamp itself wraps around and thus remains non-decreasing.

C. *Combining KV and TS Storage*

The key-value store and timestamp storage play complementary roles. TRODS-KV is more general purpose and scalable, yet introduces higher overhead than TRODS-TS. Thus, one could use these two variants together, and gain the benefits of both. In fact, we evaluate such a dual deployment in §VI. When both variants of TRODS are used together, however, a recovery server needs to know which durable store to access in order to find the connection's state. If the TCP timestamp option is not present, the TRODS module can immediately conclude that a key-value store lookup is required. If the option is present, TRODS stores a hint indicating which store to access in the high-order 7 bits of the timestamp. When the key-value store is used these bits are set to a special reserved value that TRODS-TS does not use as a timestamp index. For these connections, TRODS still needs to perform translation on the timestamps.

V. Security Concerns

The use of TRODS introduces some security concerns: attackers can spoof packets to try to initiate TRODS failover, they can modify TCP timestamps to attempt to gain access to unauthorized content, and they can more readily guess TCP sequence numbers to spoof or hijack a TCP connection. This section describes how TRODS mitigates these concerns.

A. *Denial-of-Service Attacks*

Bogus ACKs and Requests. An attacker can send requests or ACK packets to a TRODS-enabled service with spoofed, random client addresses, attempting to cause TRODS to failover non-existent connections. After all, TRODS' normal response to an unknown request or ACK packet is to initiate failover to its local application instance, wasting both application and persistent store resources.

TRODS can limit its vulnerability to such DoS attacks by initiating failover only when it can verify that it received this packet because another server recently failed. To support

this, we replicate the load balancing information to the TRODS instance on each server, i.e., its key range in the case of consistent hashing, or the server pool size (n) and its assigned number in the case of *mod n* hashing. If a failure-initiating packet arriving at a server is outside its known range—i.e., it should not be selected given the packet's 5-tuple and its knowledge of the load balancer's hashing scheme and state—then the server would only have received this packet if the load balancer's server pool recently changed. In this case, the server initiates failover. Otherwise, when the packet is in the server's known range, it is dropped as illegitimate, as it should have been in the server's local hashtable.

Therefore, TRODS mitigates this failure-initiation DoS attack, as it can be performed only temporarily on the servers directly affected by another's failure. TRODS can weaken this attack further by giving normal packet processing higher priority than failover processing. This reduces the attack from a denial-of-service to a denial-of-failover.

Clients Forcing the Slow Path. A client can force TRODS onto the slow path by sending requests that are longer than two packets and thus need to be saved to the key-value store. It can also force the slow path by sending a request that results in a multi-packet response header, which also needs to be saved to the key-value store. In either case, TRODS has no way to distinguish legitimate slow-path connections from malicious ones, so it must serve them all. However, TRODS can limit the attacker's damage by lowering the processing priority of slow-path connections, as it does with failover. Thus, slow-path attacks can still degrade the service of other slow-path connections, but they have difficulty in degrading the service of normal connections.

B. Accessing Unauthorized Content

When TCP timestamps are used for persistent storage, a client can potentially download an object they do not have permission to access, by sending an ACK packet to trigger failover with a timestamp that indexes an unauthorized objectID. This is partially unavoidable when timestamps are used, but given the enhancements to TRODS in §V-A, clients can only trigger failover after an actual failure has occurred. Thus, this attack will only work when a server has failed recently, and the attacker can guess the objectID index for the object it desires. If these security measures are not sufficient, a service should use TRODS' key-value store for all protected content. With TRODS-KV, the objectID of the client's download cannot be modified by the client.

C. TCP Sequence Number Guessing

When TRODS-TS is used, the server uses an ISN that is generated deterministically from the client's IP and port. This will raise security concerns for anyone familiar with TCP sequence number guessing attacks [15]. In these attacks, an attacker spoofs a SYN packet from a client, and then spoofs an ACK packet that acknowledges a guess of the server's ISN. If this guess is correct, it completes the TCP three-way handshake and the attacker can send a data packet that appears to be from the client.

TRODS is not vulnerable to traditional sequence number hijacking, but its approach allows malicious clients, once having completed a successful connection, to initiate new downloads before fully establishing new connections.[6] This vector may be used as a denial-of-service attack. In particular, rather than randomly, TRODS generates its ISN from a cryptographic hash of the client IP, port, time epoch, and a private key that is known to all servers in a TRODS cluster. Thus, an attacker learns the ISN for a given IP and port only if it can receive traffic at that network location. This is akin to the protections offered by normal randomized ISNs, except that this ISN is constant across the entire epoch. After learning the ISN, a client then can spoof connections from other network locations during the same epoch. That said, TRODS is used for services that are inherently read-only and so the attack can only be used to start illegitimate downloads and cannot modify state. If limiting the sequence number guessing attack to a DoS attack from a limited range of IPs and ports is unacceptable for a service, it should use TRODS-KV instead.

VI. Evaluation

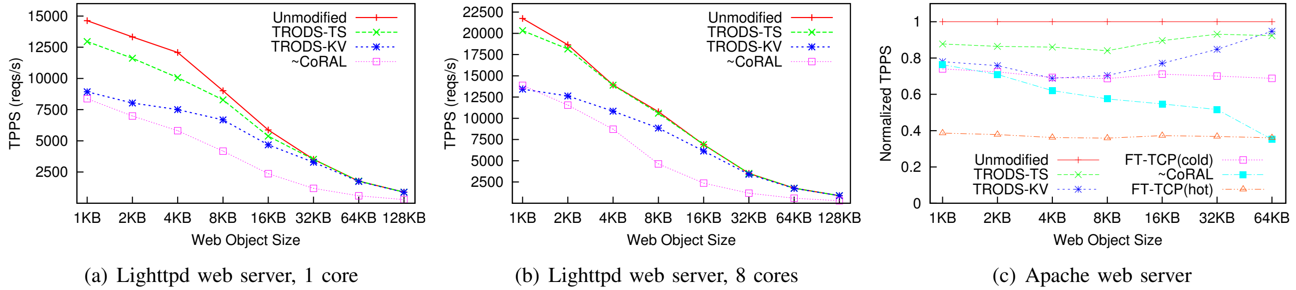
This section demonstrates the practicality and effectiveness of TRODS. We first quantify TRODS' cost in terms of decreased throughput and increased latency. We then evaluate how TRODS handles failure in a cluster setting and how much excess latency it incurs due to failover.

Implementation. The TRODS implementation is approximately 3,000 lines of C code. It is a loadable kernel module for Linux 2.6.32.3 and using it does not require recompiling the base kernel or rebooting the machine. The current TRODS implementation handles the normal cases, where none of our assumptions from Section §III are violated.

We also implemented ~CoRAL, a partial implementation of CoRAL [1] in a kernel module for Linux 2.6.32.3. CoRAL routes requests to a primary server through a backup server and saves the entire response on the backup before sending any of it to the client. Our implementation only saves the response on a backup machine and thus gives a rough upper bound on the true performance of CoRAL.

Experimental Setup. Our lighttpd throughput, hybrid throughput, and latency experiments use a total of three machines: one to run clients, one for a TRODS server, and one for a key-value store. The excess-latency-due-to-failover experiment uses those machines and an additional one for load-balancing. The failure recovery experiment uses three server machines, as well as one machine each for clients, a load balancer, and the key-value store. Each machine used in these experiments has eight 2.3GHz cores and 8 GB

[6]Note that clients of a TRODS service are not vulnerable to this attack, as they use standard TCP ISNs.



(a) Lighttpd web server, 1 core

(b) Lighttpd web server, 8 cores

(c) Apache web server

Figure 4. **HTTP throughput experiments. The first experiment (a) uses lighttpd with a single server core, which becomes CPU bound. The second (b) uses lighttpd with 8 cores and is not CPU bound. The final (c) uses Apache in order to compare TRODS to FT-TCP. For all, the median value over 25 trials is shown; min and max values are within 5% of the median and omitted.**

of memory, and is connected to a 1 Gbps switch. Our Apache throughput experiment was run on Emulab [22] using pc3000 nodes connected via 1 Gbps links.

We use memcached 1.4.4 [5] without expiration or eviction as our key-value store and we use lighttpd 1.4.23 [12] as our regular web server. We use a simple Click [11] configuration run in the kernel as our load balancer.

All throughput experiments show the median value of 25 trials; the min and max values are always within 5% of the median and are omitted. Each throughput trial consisted of enough client processes to saturate server throughput continuously fetching a web object. We ran the tests for 40s and exclude the first and last 5s of each trial to avoid including experimental artifacts or non-steady-state rates.

Throughput Per Server (TPPS). We evaluate the throughput per server of each system. This throughput metric accounts for *all* the machines needed to provide failover. For instance, in TRODS-KV, it accounts for the use of the key-value store, while in FT-TCP, it accounts for the use of the backup server. These "complete system costs" allow us to more accurately evaluate each approaches' overhead compared to an unmodified system lacking failover capabilities; prior work largely avoided such comparisons.

We calculate TPPS by dividing the throughput of primary servers by $1+TP_P/TP_S$, where TP_P and TP_S are the primary and secondary servers' throughputs, respectively. So, in a "hot backup" scheme where each primary has its own backup ($TP_P = TP_S$), the TPPS is $\frac{TP_P}{1+(TP_P/TP_S)} = \frac{TP_P}{2}$. To calculate the TPPS for TRODS-KV, we benchmarked its fixed size saves to our memcached key-value store at 123,468 saves/s. Then, in Figure 4, when the primary server for TRODS-KV achieved a throughput of TP_P, we report a TPPS of $\frac{TP_P}{1+TP_P/123468}$. To calculate the TPPS for ~CoRAL, we benchmarked its object-sized saves to our memcached backup. For 1K objects, memcached can handle 95,037 saves/sec and the CoRAL primary can handle 9,209 requests/sec, so the 1K TPPS is $\frac{9209}{(1+(9209/95037))} = 8395$.

A. Throughput

We ran three sets of experiments to examine how TRODS affects the maximum throughput of a web server.

In our first experiment, shown in Figure 4(a), we turned off all but one of the cores on the server machine, and ran the web server as a single process, which ended up consuming 100% of the CPU. TRODS operations in the kernel steal cycles from the web server, leading to a larger performance degradation that in the non-CPU-bound experiment. Using only a single core, TRODS-TS experiences an 11% decrease in TPPS for 1 KB web objects (from 14,608 requests/s to 12,980 reqs/s). TRODS-KV sees a 39% reduction in TPPS (to 8,940 reqs/s). TRODS-KV's higher overhead arose from both its reduced throughput on the web server machine, as well as its consumed resources on the key-value store. However, as object sizes increase, the web server becomes less CPU bound and, as a result, the throughput of both TRODS variants become competitive with the unmodified service. For example, when objects are 32 KB, TRODS-TS's TPPS is less than 0.01% lower than the unmodified system, and TRODS-KV's TPPS is only 20% lower. In contrast, ~CoRAL sees a 43% lower throughput for 1 KB web objects than an unmodified service, and this grows to a 66% lower TPPS for 32 KB objects: ~CoRAL's overhead grows as object sizes increase, as saving the entire object on a backup machine becomes more costly.

In our second experiment, shown in Figure 4(b), all 8 cores on the server are used by 8 server processes. The web server is no longer CPU bound and TRODS processing has a smaller effect on throughput. For 1 KB web objects, TRODS-TS decreases TPPS by 7% (from 21,745 reqs/s to 20,315 reqs/s), while TRODS-KV decreases TPPS by 38%. As with a single core, as object size increases, the TRODS variants become more and more competitive with the unmodified service: TRODS-TS is within 3% of the unmodified for all objects 2 KB and larger, while TRODS-KV is within 11% for all objects 16 KB and larger. On the other hand, ~CoRAL again experiences a 40% decrease in TPPS for 1 KB web object, with its relative TPPS continuing to worsen as object size increases.

In our third experiment, shown in Figure 4(c), we evaluated throughput with the Apache web server included with the FT-TCP codebase. As FT-TCP requires a Linux 2.4.20 kernel, we ran these experiments on Emulab. Fig-

ure 4(c) shows the TPPS for FT-TCP's hot and cold backup approaches,[7] normalized against the TPPS for unmodified Apache on a 2.4 kernel. The figure also includes the TPPS for TRODS and ~CoRAL, normalized against unmodified Apache on a 2.6 kernel, which is slightly more efficient than on the older kernel. Both TRODS and ~CoRAL exhibit behavior similar to the previous experiments. FT-TCP's variants exhibit relatively stable normalized TPPS, as the amount of additional work the scheme uses is a fixed percentage of the total work that a given response requires.

In summary, when objects are 16 KB or larger, TRODS is competitive with an unmodified service and achieves much lower overhead than either ~CoRAL or FT-TCP. When objects are small and the server is CPU limited, TRODS suffers moderately decreased throughput, but it still has lower overhead than ~CoRAL or FT-TCP.

B. Hybrid Throughput

Figure 5(a) characterize TRODS' performance when using our two persistent stores in varying proportions. We measure the median throughput of 25 trials, run with the same parameters as our previous single-core throughput experiment. We normalize these throughputs against that achieved when only using the key-value store (i.e., 100% KV), to demonstrate the relative performance gains from a hybrid deployment. For reference, we also plot an "ideal 50/50" line that shows the average of TRODS-KV and TRODS-TS.

The experiment demonstrates that the hybrid version of TRODS performs well. When 50% of connections use the KV store and the other 50% use timestamps, TRODS throughput is within 4% of the ideal. This alleviates our concern that requests that use the slower KV store will unduly decrease the throughput of the hybrid system.

C. Latency

Table 5(b) shows the median and 99th percentile latencies for different sections of 10,000 sequential fetches of a 1 byte web object. The latencies are measured by analyzing `tcpdump` logs of the client's connections.

The period between the client sending a SYN packet and receiving a SYN/ACK experienced no additional latency for either variant of TRODS: storing local knowledge of a connection has no discernible overhead. Between sending an HTTP request (Req) and receiving the first data packet in the response (Resp.1), TRODS-KV has notably higher latency than an unmodified service. This comes from TRODS-KV blocking the connection until its save to the key-value store completes. In contrast, TRODS-TS does not block the connection and avoids any latency penalty.

Examining the latency of the entire connection (the SYN-FIN section) reveals that TRODS-TS imposes less than 10 μs of additional latency, while TRODS-KV imposes less than 150 μs of additional latency. Both of these increases are miniscule compared to the tens to hundreds of milliseconds of delay between clients and servers across the wide-area.

Having shown that TRODS adds little to no overhead to server throughput, we now demonstrate that it successfully recovers client connections from server failures. Figure 5(c) shows the per-second throughput of a 3-server cluster over a time period with individual server failures. Each server runs TRODS-TS; the resulting graph for TRODS-KV (not shown) is similar. Web requests for 8 KB objects are concurrently issued by 200 HTTP clients (running on the same physical machine), who access the cluster through a single load-balancer. The load-balancer also delays packets to create a synthetic 20 ms RTT from clients to servers, emulating wide-area connection latencies to nearby datacenters.

We fail server 1 during the 7th second of the experiment by taking down its network interface. Upon detecting this failure, the load balancer updates its server pool. Previously established connections to server 1 are reassigned to the remaining servers; the TRODS components of these servers recover the reassigned connections. We further fail server 2 during the 20th second of the experiment, at which time the load balancer directs all connections to the remaining server. We find that the cluster's total throughput, as shown in the solid line at the top of Figure 5(c), remains constant throughout the experiment; the overhead of recovering failed connections does not have a noticeable effect.[8]

D. Failover Latency

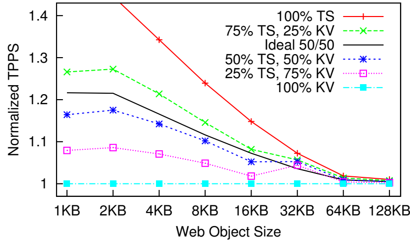
While we have demonstrated that TRODS successfully fails over connections between servers, this does not quantify the delay experienced by clients during this process. Figure 6 shows the *excess* latency due to failover for a client requesting various object sizes. The figure shows the results for TRODS-KV using a single server; the results for TRODS-TS are similar and omitted. A load-balancer sits on path and delays packets to create a 20 ms RTT. For each run, a client requests an object of a given size for 5 minutes, while we synthesize a failure in the every 4 seconds at the server.

We synthesize the failure in the kernel module by dropping all packets during failure period and sending RST packets to the server application for all existing connections (we can synthesize such failures at a much higher rate than we can induce actual ones). We simulate the use of a 25 ms heartbeat timer by setting the failure period to a random amount of time less than 25 ms, effectively mimicking the behavior we observed in our previous experiment.

We measure latency as the time between the client sending its first SYN packet and receiving the final byte of the response. A transfer's excess latency is defined as its increase over the median of all non-failed connections during the run. We graph the 95th percentile of the excess latency for non-failed connections. A full 75% of failed-over connections

[7]Note that we disabled system call interception for this experiment, as it is unnecessary for deterministic services.

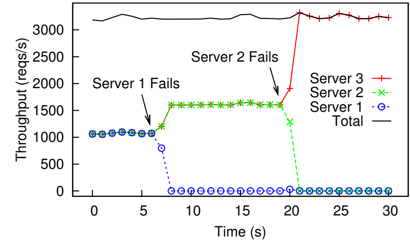
[8]We limit the cluster's throughput to ~3K reqs/s to ensure that `tcpdump`, which we use to record the experiment, does not drop any packets.



(a) Hybrid throughput experiment showing the relative performance of TRODS when both persistent stores are used. We plot the throughput of each proportion normalized against 100% KV.

Start	End	Normal	TRODS-TS	TRODS-KV
SYN	SYN/ACK	90 (95)	89 (93)	90 (94)
Req	Resp.1	137 (150)	135 (49)	256 (282)
SYN	FIN	353 (372)	362 (384)	490 (520)

(b) Median (and 99th percentile) latency in μs for different sections of a single HTTP connection, for both unmodified and TRODS services. See Figure 2 for a depiction of these sections.



(c) Server and cluster throughput using TRODS-TS when undergoing failure recovery. The cluster's total throughput is unaffected by these failures, and no connections are broken.

Figure 5.

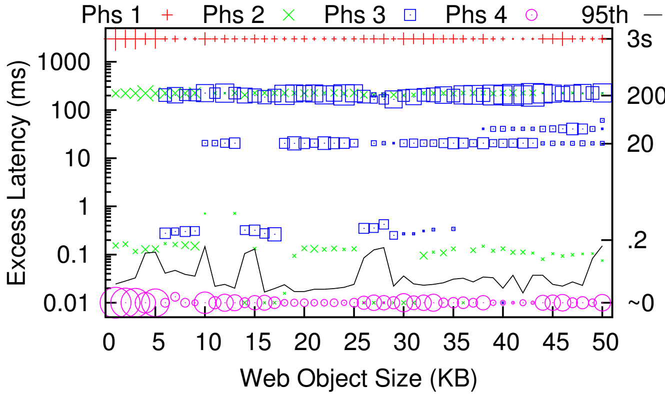


Figure 6. **Excess latency when experiencing failover using TRODS-KV. Overlapping data points that occur in the same phase of the connection are combined and the size of the shown points is proportional to the number of data points they represent. The different failover phases are shown in Figure 2.**

had excess latency above this 95th percentile, suggesting that this excess latency is due to failover and not noise.

Each data point in Figure 6 indicates the connection phase (as specified in §III-B) during which the failure occurred. If data points in the same phase overlap (within 10% of each other), we combine them and increase the plotted size of the representative point. We find that the excess latency has a multi-modal distribution, with distinct modes at ~0 ms, .2 ms, 20 ms, 200 ms, and 3 s, as marked on the right y-axis of the figure. We briefly characterize these sources of added latency, in turn:

- **~0 ms**: Phase 4 failovers occur after the client has received the entire response and thus add no latency.
- **.2 ms**: This excess latency corresponds to the time needed to perform a key-value lookup, setup a new connection to the server, and splice in the client's connection. We observe it when failover is triggered by an in-flight packet.
- **20 ms**: Normal connections establish a large TCP window by the time they reach the middle of the download phase. When failover occurs at this point, it sets up a new connection, and thus resets the window size. This increases the number of RTTs needed to transmit the entire object and creates a cluster of failovers around one RTT (20 ms).
- **200 ms**: Our experiment uses Linux clients that have a 200 ms minimum value for their retransmit timeout. Thus, even though the RTT of the connection is well below 200 ms, we always incur at least that latency when failover is triggered by a retransmitted client packet.
- **3 s**: When a SYN packet is lost, the client does not have an estimate of the connection's RTT. Thus, it waits a conservative 3 s before retransmitting.

As object size increases, the proportion of phase 3 failovers also increases. This is not surprising: As a larger proportion of a connection's transfer time is spent downloading an object (i.e., in phase 3), the likelihood of failure occurring during that phase similarly increases. Notice that there are no phase 3 failovers for pages 5 KB and smaller. This occurs because objects of this size or smaller can be sent in a single TCP window with all packets within the window separated by less than 15 μs. This small gap makes it highly likely that the server will fail before (phase 2) or after (phase 4) sending them all, and we did not observe any contradictions to this in the course of our experiment.

In summary, most failovers in TRODS occurs with less than 200 ms of excess latency, sufficiently low to not noticeably impact a user's experience. While some connections experience a 3 s delay (12% of larger objects), this delay is unavoidable due to SYN retransmission timers. In either case, we argue it is still preferable to a broken connection.

VII. Related Work

New Transport Layers. Several solutions for failure recovery introduce new transport layer protocols or primitives. Trickles [16] uses a new transport layer protocol and a new sockets API to make one end of a connection stateless. SCTP [19] is a transport layer protocol that, among many other things, allows a client to have connections to multiple servers it can switch between. TCP Migrate [17] can be used to migrate a connection from one server to another, which can then be used with a soft-state synchronization protocol between servers to accomplish failover [18]. M-TCP [20] is a another TCP-like transport protocol designed to support migration. All of these solutions modify the client's TCP/IP stack, TRODS does not require any client-side changes.

TCP Failover. Moving up the stack, there is a large body of work on failover for TCP connections that do not require changes to clients. FT-TCP [24] accomplishes TCP failover by logging (persistently storing) every packet in a TCP connection on a primary server to a backup server. Then, if the primary server fails, the (cold) backup runs through the TCP connection, and, once it catches up to the client's current position in the stream, it begins serving the client. As this can make the time to failover a connection arbitrarily long, they also describe a hot backup that processes all packets upon receiving them. FT-TCP is more general than TRODS, as it applies to all deterministic TCP services. However, FT-TCP pays for this generality with increased overhead. Every packet must be logged or processed in FT-TCP, while TRODS-KV only "logs" once per object and TRODS-TS avoids it entirely. Koch *et al.*describe a system [10] that is very similar to FT-TCP's hot backup approach. ST-TCP [14] is another primary-backup system that avoids some logging overhead placing the primary and backup on the same L2 network and having the backup snoop on the primary's traffic. Zhang *et al.* [25] describe a similar system that uses a stateful load-balancer to explicitly transmit packets to both the primary and backup. The Backdoors [21] avoids logging by using programmable NICs to extract TCP and application state from the server's memory after an OS crash.

HTTP Failover. CoRAL [1] is primary-backup system targeted at HTTP. All packets bound for the primary are first routed through the backup who logs them. The primary then uses application-level knowledge to identify the full reply and persistently store it on the backup. TRODS is more efficient that CoRAL because it avoids persistently storing the entire reply. Luo *et al.* [13] describes a system for HTTP failover where a "dispatcher" (load balancer) terminates the client's connections. Once a client has sent an entire request, the load balancer stores it and forwards it onto a server. Then if that server fails before fully responding, the load balancer reconnects to a new server to continue. This moves the problem of failure from the servers to the load balancer.

TCP Timestamps. We are not the first to use the TCP timestamp option for embedding state. Giffin *et al.* [6] use the low order bits of the TCP timestamp as a covert channel for undetectable communication. In addition, starting with version 2.6.26, the Linux kernel added support for window scaling and SACK options in SYN cookies by encoding their value in the lowest 9 bits of the TCP timestamp [7].

VIII. CONCLUSION

TRODS is a fully backwards-compatible system for introducing transparent failover to object delivery services. TRODS leverages cross-layer knowledge of both application and TCP state, as well as TCP's reliable transmission mechanisms, to exert control over unmodified clients. This control allows TRODS to coerce clients into providing storage and initiating failover when needed.

Through evaluation of a TRODS cluster, we demonstrate that it is both practical and efficient. In failure-free scenarios, TRODS does not significantly increase latency or decrease server throughput. When a failure occurs, TRODS transparently restores clients' ongoing connections, without adding significant latency to the connections.

ACKNOWLEDGMENT

The authors would like to thank Erik Nordström, Muneeb Ali, Anirudh Badam, Nate Foster, Prem Gopalan, Rob Harrison, Michael Kaminsky, Wonho Kim, Steven Ko, David Shue, Jeff Terrace, and Vijay Vasudevan for helpful comments on earlier drafts of this paper. This work was supported by NSF CAREER Grant #0953197 (CSR) and an ONR Young Investigator Award.

REFERENCES

[1] N. Aghdaie and Y. Tamir. CoRAL: A transparent fault-tolerant web service. *Journal of Systems and Software*, 82(1), 2009.
[2] A. Broder et al. Graph structure in the web. In *Proc. World Wide Web Conference (WWW)*, May 2000.
[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 1996.
[4] R. Fielding et al. RFC 2616: HTTP/1.1, Jun 1999.
[5] B. Fitzpatrick. Memcached: a distributed memory object caching system. http://memcached.org/, 2009.
[6] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging through tcp timestamps. In *Proc. PET*, Apr. 2002.
[7] Improving syncookies. http://lwn.net/Articles/277146/, Apr 2008.
[8] V. Jacobson, R. Braden, and D. Borman. RFC 1323: Tcp extensions for high performance, May 1992.
[9] D. Karger et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. Symposium on Theory of Computing (STOC)*, May 1997.
[10] R. R. Koch, S. Hortikar, L. E. Moser, and P. M. Melliar-Smith. Transparent tcp connection failover. *Proc. DSN*, June 2003.
[11] E. Kohler et al. The click modular router. *ACM TOCS*, 2000.
[12] Lighttpd. http://www.lighttpd.net/, 2010.
[13] M. Luo and C. Yang. Constructing zero-loss web services. In *Proc. INFOCOM*, Apr. 2001.
[14] M. Marwah, S. Mishra, and C. Fetzer. Tcp server fault tolerance using connection migration to a backup server. In *Proc. DSN*, Jun 2003.
[15] R. Morris. A weakness in the 4.2bsd unix tcp/ip software. Technical Report TR-117, Bell Labs, 1985.
[16] A. Shieh, A. C. Myers, and E. G. Sirer. A stateless approach to connection-oriented protocols. *ACM TOCS*, 26, 2008.
[17] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. MobiCom*, Aug. 2000.
[18] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proc. USITS*, Mar. 2001.
[19] R. Stewart. RFC 4960: SCTP, Sep 2007.
[20] F. Sultan et al. Migratory tcp: connection migration for service continuity in the internet. In *Proc. ICDCS*, July 2002.
[21] F. Sultan et al. Recovering internet service sessions from operating system failures. *IEEE Internet Computing*, 9(2), 2005.
[22] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, Dec. 2002.
[23] Windows ephemeral port range. http://support.microsoft.com/kb/929851, 2009.
[24] D. Zagorodnov et al. Practical and low-overhead masking of failures of tcp-based servers. *ACM TOCS*, 27(2), 2009.
[25] R. Zhang, T. F. Abdelzaher, and J. A. Stankovic. Efficient tcp connection failover in web server clusters. In *Proc. INFOCOM*, Mar. 2004.

Appeared in ***7th USENIX Symposium on Network Design and Implementation (NSDI '10)***

Prophecy: Using History for High-Throughput Fault Tolerance

Siddhartha Sen, Wyatt Lloyd, and Michael J. Freedman
Princeton University

Abstract

Byzantine fault-tolerant (BFT) replication has enjoyed a series of performance improvements, but remains costly due to its replicated work. We eliminate this cost for read-mostly workloads through Prophecy, a system that interposes itself between clients and any replicated service. At Prophecy's core is a trusted *sketcher* component, designed to extend the semi-trusted load balancer that mediates access to an Internet service. The sketcher performs fast, load-balanced reads when results are historically consistent, and slow, replicated reads otherwise. Despite its simplicity, Prophecy provides a new form of consistency called *delay-once consistency*. Along the way, we derive a distributed variant of Prophecy that achieves the same consistency but without any trusted components.

A prototype implementation demonstrates Prophecy's high throughput compared to BFT systems. We also describe and evaluate Prophecy's ability to scale-out to support large replica groups or multiple replica groups. As Prophecy is most effective when state updates are rare, we finally present a measurement study of popular websites that demonstrates a large proportion of static data.

1 Introduction

Replication techniques are now the norm in large-scale Internet services, in order to achieve both reliability and scalability. However, leveraging active agreement to *mask* failures, whether to handle fail-stop behavior [41, 50] or fully malicious (Byzantine) failures [42], is not yet widely used. There is some movement in this direction from industry—such as Google's Chubby [10] and Yahoo!'s Zookeeper [66] coordination services, based on Paxos [41]—but both are used to manage infrastructure, not directly mask failures in customer-facing services.

And yet non-fail-stop failures in customer-facing services continue to occur, much to the chagrin and concern of system operators. Failures may arise from malicious break-ins, but they also may occur simply from system misconfigurations: Facebook leaking source code due to *one* misconfigured server [60], or Flickr mixing up returned images due to *one* improper cache server [24]. In fact, both of these examples could have been prevented through redundancy and agreement, without requiring full N-version programming [8]. The perceived need for systems robust to Byzantine faults—a superset of misconfigurations and Heisenbugs—has spurned almost a cottage industry on improving performance results of Byzantine fault tolerant (BFT) algorithms [1, 6, 12, 17, 30, 37, 38, 56, 62, 64, 65, 67].

While the latency of recent BFT algorithms has approached that of unreplicated reads to individual servers [15, 38, 64], the throughput of such systems falls far short. This is simple math: a minimum of four replicas [12] (or sometimes even six [1]) are required to tolerate one faulty replica, and at least three must participate in each operation. For datacenters in the (tens of) thousands of servers, requiring four times as many servers for the same throughput may be a non-starter. Even services that already replicate their data, such as the Google File System [25], would see their throughput drop significantly when using BFT agreement.

But if the replication cost of BFT is provably necessary [9], something has to give. One might view our work as a thought experiment that explores the potential benefit of placing a small amount of trusted software or hardware in front of a replicated service. After all, wide-area client access to an Internet service is typically mediated by some middlebox, which is then at least trusted to provide access to the service. Further, a small and simple trusted component may be less vulnerable to problems such as misconfigurations or Heisenbugs. And by treating the back-end service as an abstract entity that exposes a limited interface, this simple device may be able to interact with both complex and varied services. Our implementation of such a device has less than 3000 lines of code.

Barring such a solution, most system designers opt either for cheaper techniques (to avoid the costs of state machine replication) or more flexible techniques (to ensure service availability under heavy failures or partitions). The design philosophies of Amazon's Dynamo [18], GFS [25], and other systems [20, 23, 61] embrace this perspective, providing only eventually-consistent storage. On the other hand, the tension between these competing goals persists, with some systems in industry re-introducing stronger consistency properties. Examples include timeline consistency in Yahoo!'s PNUTS [16] and per-user cache invalidation on Facebook [21]. Nevertheless, we are unaware of any major

use of *agreement* at the front-tier of customer-facing services. In this paper, we challenge the assumption that the tradeoff between strong consistency and cost in these services is fundamental.

This paper presents Prophecy, a system that lowers the performance overhead of fault-tolerant agreement for customer-facing Internet services, at the cost of slightly weakening its consistency guarantees. At Prophecy's core is a trusted *sketcher* component that mediates client access to a service replica group. The sketcher maintains a compact history table of observed request/response pairs; this history allows it to perform fast, load-balanced reads when state transitions do not occur (that is, when the current response is identical to that seen in the past) and slow, replicated reads otherwise (when agreement is required). The sketcher is a flexible abstraction that can *interface with any replica group*, provided it exposes a limited set of defined functionality. This paper, however, largely discusses Prophecy's use with BFT replica groups. Our contributions include the following:

- When used with BFT replica groups that guarantee linearizability [32], Prophecy significantly increases throughput through its use of fast, load-balanced reads. However, it relaxes the consistency properties to what we term *delay-once* semantics.
- We also derive a distributed variant of Prophecy, called D-Prophecy, that similarly improves the throughput of traditional fault-tolerant systems. D-Prophecy achieves the same delay-once consistency but *without any trusted components*.
- We introduce the notion of *delay-once consistency* and define it formally. Intuitively, it implies that faulty nodes can at worst return only stale (not arbitrary) data.
- We demonstrate how to scale-out Prophecy to support large replica groups or many replica groups.
- We implement Prophecy and apply it to BFT replica groups. We evaluate its performance on realistic workloads, not just null workloads as typically done in the literature. Prophecy adds negligible latency compared to standard load balancing, while it provides an almost linear-fold increase in throughput.
- Prophecy is most effective in read-mostly workloads where state transitions are rare. We conduct a measurement study of the Alexa top-25 websites and show that over 90% of requests are for mostly static data. We also characterize the dynamism in the data.

Table 1 summarizes the different properties of a traditional BFT system, D-Prophecy, and Prophecy. The remainder of this paper is organized as follows. In §2 we motivate the design of D-Prophecy and Prophecy, and we describe this design in §3. In §4 we define delay-once consistency and analyze Prophecy's implementation of this consistency model. In §5 we discuss extensions to the basic system model that consider scale and complex component topologies. We detail our prototype implementation in §6 and describe our system evaluation in §7. In §8 we present our measurement study. We review related work in §9 and conclude in §10.

Property	BFT	D-Prophecy	Prophecy
Trusted components	No	No	Yes
Modified clients	Yes	Yes	No
Session length	Long	Long	Short, long
Load-balanced reads	No	Yes	Yes
Consistency	Linearized	Delay-once	Delay-once

Table 1: **Comparison of a traditional BFT system, D-Prophecy, and Prophecy.**

2 Motivating Prophecy's Design

One might rightfully ask whether Prophecy makes unfair claims, given that it achieves performance and scalability gains at the cost of additional trust assumptions compared to traditional fault-tolerant systems. This section motivates our design through the lens of BFT systems, in two steps. First, we improve the performance of BFT systems on realistic workloads by introducing a cache at each replica server. By optimizing the use of this cache, we derive a distributed variant of Prophecy that does not rely on any trusted components. Then, we apply this design to customer-facing Internet services, and show that the constraints of these services are best met by a shared, trusted cache that proxies client access to the service replica group. The resulting system is Prophecy.

In our discussion, we differentiate between *write requests*, or those that modify service state, and *read requests*, or those that simply access state.

2.1 Traditional BFT Services and Real Workloads

A common pitfall of BFT systems is that they are evaluated on null workloads. Not only are these workloads unrealistic, but they also misrepresent the performance overheads of the system. Our evaluation in §7 shows that the cost of executing a non-null read request in the PBFT system [12] dominates the cost of agreeing on the ordering of the request, even when the request is served entirely from main memory. Thus the PBFT read optimization, which optimistically avoids agreement on read requests, offers little or no benefit for most realistic workloads. Improving the performance of read requests requires optimizing the *execution* of the reads themselves.

Unlike write requests, which modify service state and hence must be executed at each replica server, read requests can benefit from causality tracking. For example, if there are no causally-dependent writes between two identical reads, a replica server could simply cache the response of the first read and avoid the second read altogether.[1] However, this requires (1) knowledge of the causal dependencies of all write requests, and (2) a response cache of all prior reads at each replica server. The first requirement is unrealistic for many applications: a single write may modify the service state in complex ways. Even if we address this problem by invalidating the entire response cache upon receiving any write, the space needed by such a cache could be prohibitive: a cache of Facebook's 60+ billion images on April 30, 2009 [49], assuming a scant 1% working-set size, would occupy approximately 15TB of memory. Thus, the second requirement is also unrealistic.

Instead of caching each response r, the replica servers can store a compact, collision-resistant sketch $s(r)$ to enable *cache validation*. That is, when a client issues a read request for r, only one replica server executes the read and replies with r, while the remaining replica servers reply with $s(r)$ from their caches. The client accepts r only if the replica group agrees on $s(r)$ and if $s(r)$ validates r. Thus, even if the replica that returns r is faulty, it cannot make the client accept arbitrary data; in the worst case, it causes the client to accept a stale version of r. Therefore we only need to ask one replica to execute the read, effectively implementing what we call a *fast read*. Fast reads drastically improve the throughput of read requests and can be load-balanced across the replica group to avoid repeated stale results. The replica servers maintain a fresh cache by updating it during regular (replicated) reads, which are issued when fast reads fail. Using a compact cache reduces the memory footprint of the Facebook image working set to less than 27GB.

We call the resulting system Distributed Prophecy, or D-Prophecy, and call the consistency semantics it provides *delay-once consistency*.

2.2 BFT Internet Services

An oft-overlooked issue with BFT systems, including D-Prophecy, is that they are *implicitly* designed for services with long-running sessions between clients and replica servers (or at least always presented and evaluated as such). Clients establish symmetric session keys with each replica server, although the overhead of doing so is not typically included when calculating system performance. Figure 1 shows the throughput of the PBFT im-

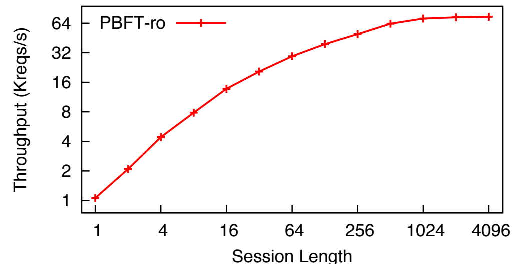


Figure 1: **PBFT's throughput in the thousands of requests per second for null requests in sessions of varying length. Note that both axes are log scale.**

plementation as a function of session length, with all relevant optimizations enabled including the read optimization (indicated by 'ro'). As sessions get shorter, throughput is drastically reduced because replicas need to decrypt and verify clients' new session keys. For PBFT sessions consisting of 128 read requests, throughput is half of its maximum, and for sessions consisting of 8 read requests, throughput is one-tenth of its maximum.

The assumption of long-lived sessions breaks down for Internet services, however, which are mostly characterized by *short-lived sessions* and *unmodified clients*. These properties make it impractical for clients to establish per-session keys with each replica. Moreover, depending on clients to perform protocol-specific tasks leads to poor backwards compatibility for legacy clients of Internet services (*e.g.*, web browsers), where cryptographic support is not easily available [2]. Instead, we might turn to using an entity knowledgeable of the BFT protocol to proxy client requests to a service replica group. And since Internet services already rely on the correct operation of local middleboxes (at least with respect to service availability), we extend this reliance by converting the middlebox into a trusted proxy. The trusted proxy interfaces multiple short-lived sessions between clients and itself with a single long-lived session between itself and the replica group, acting as a client in the traditional BFT sense.

When using proxied client access to a D-Prophecy group, there is no need to maintain redundant caches at each replica server: a shared cache at the trusted proxy suffices, and it preserves delay-once consistency. A fast read now mimics the performance of an unreplicated read, as the proxy only asks one replica server for r and validates the response with its (local) copy of $s(r)$. Since the cache is compact, the proxy remains a small and simple trusted component, amenable to verification. We call this system Prophecy, and present its design in §3.

2.3 Applications

The delay-once semantics of Prophecy imply that faulty nodes can at worst return stale (not arbitrary) data. This

[1] Other causality-based optimizations, such as client-side speculation [64] or server-side concurrent execution [37] are also possible, but are complementary to any cache-based optimizations.

semantics is sufficient for a variety of applications. For example, Prophecy would be able to protect against the Facebook and Flickr mishaps mentioned in the introduction, because it would not allow arbitrary data to reach the client. Applications that serve inherently static (write-once) data are also good candidates, because here a "stale" response is as fresh as the latest response. In §8 we demonstrate the propensity for static data in today's most popular websites.

Social networks and "Web 2.0" applications are good candidates for delay-once consistency because they typically do not require all writes to be immediately visible. Consider the following example from Yahoo!'s PNUTS system [16]. A user wants to upload spring-break photos to an online photo-sharing site, but does not want his mother to see them. So, he first removes her from the permitted access list of his database record and then adds the spring-break photos to this record. A consistency model that allows these updates to appear in different orders at different replicas, such as eventual consistency [22], is insufficient: it violates the user's intention of hiding the photos from his mother. Delay-once consistency only allows stale data to be returned, not data out-of-order: if the photos are visible, then the access control update must have already taken place. Further, once the user has "refreshed" his own page and sees the photos, he is guaranteed that his friends will also see them.

For applications where writes are critical, such as a bank account, delay-once consistency is appropriate because it ensures that writes follow the protocol of the replica group. Although reads may return stale results, they can only do so in a limited way, as we discuss in §4. On the other hand, there are some applications for which delay-once consistency is not beneficial, such as those that critically depend on reading the latest data (*e.g.*, a rail signaling service), or those that return non-deterministic content (*e.g.*, a CAPTCHA generator).

3 System Design

We first define a sketcher abstraction that lies at the heart of Prophecy. For a more traditional setting, we use this sketcher to design a distributed variant of Prophecy, or D-Prophecy. We then present the design of Prophecy.

3.1 The Sketcher

Prophecy and D-Prophecy use a sketcher to improve the performance of read requests to an existing replica group. A *sketcher* maintains a history table of compact, collision-resistant sketches of requests and responses processed by a replica group. Each entry in the history table is of the form $(s(q), s(r))$, where q is a request, r is the response to q, and s is the sketching function used for compactness (s typically makes use of a secure hash function like SHA-1). The sketcher looks up or updates entries in the history table using a standard get/set interface, keyed by $s(q)$. In Prophecy, only read requests and responses are stored in the history table.

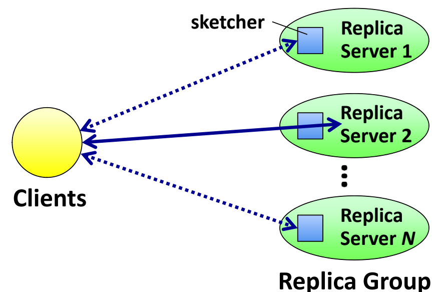


Figure 2: **Executing a fast read in D-Prophecy. Only one replica server executes the read (bold line); the others return the response sketch in the history table (dashed lines).**

The specific use of the sketcher and its interaction with the replica group differs between Prophecy and D-Prophecy. However, both systems require the replica group to support the following request interface:

- RESP $\leftarrow$ *fast*(REQ q)
- (RESP r, SEQ_NO σ) $\leftarrow$ *replicated*(REQ q)

We expect the *fast* interface to be new for most replica groups. The *replicated* interface should already exist, but may need to be extended to return sequence numbers. No modifications are made to the replica group beyond what is necessary to support the interfaces, in either system.

3.2 D-Prophecy

Figure 2 shows the system model of D-Prophecy. Except for the sketcher, all other entities are standard components of a replicated service: clients send requests to (and receive responses from) a service implemented by N replica servers, according to some replication protocol like PBFT. Each replica server is augmented with a sketcher that maintains a history table for read requests. The history table is read by the *fast* interface and updated by the *replicated* interface, as follows.

A client issues a fast read q by sending it to all replica servers and choosing one of them to execute q and return r. The policy for selecting a replica server is unspecified, but a uniformly random policy has especially useful properties (see §4.2). The other replicas use their sketcher to lookup the entry for $s(q)$ and return the corresponding response sketch $s(r)$, or null if the entry does not exist. If the client receives a quorum of non-null response sketches that match the sketch of the actual response, it accepts the response. The quorum size depends on the replication protocol; we give an example

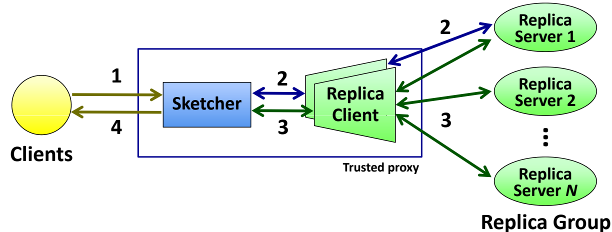


Figure 3: **Prophecy mediating access to a replica group.**

below. Otherwise, we say a *transition* has occurred and the client reissues the request as a replicated read. A replicated read is executed according to the protocol of the replica group, with one additional step: all replica servers use their sketcher to update the entry for $s(q)$ with the new value of $s(r)$, before sending a response to the client.

Readers familiar with the PBFT protocol will notice that fast reads in D-Prophecy look very similar to PBFT optimized reads. However, there is a crucial difference: PBFT requires every replica server to execute the read, while D-Prophecy requires only one such execution, performing in-memory lookups of $s(r)$ at the rest. For non-null workloads, this represents a significant performance improvement, as shown in §7. On the flip side, each replica server requires additional memory to store its history table, though in practice this overhead is small. The quorum size required for fast reads is identical to the quorum size required for optimized reads: $(2N+1)/3$ responses suffices with some caveats (see §5.1.3 of [11]), and N always suffices.

The architecture of D-Prophecy resembles that of a traditional BFT system: clients establish session keys with the replica servers and participate fully in the replication protocol. As we observed in §2.2, this makes D-Prophecy unsuitable for Internet services, with their environment of short-lived sessions and unmodified clients. This motivates the design of Prophecy, discussed next.

3.3 Prophecy

Figure 3 shows the simplest realization of Prophecy's system model. (We consider extensions to the basic model in §5.) There are four types of entities: clients, sketchers, replica clients, and replica servers. Unmodified clients' requests to a service are handled by the sketcher; together with the replica clients, this serves as the trusted proxy described in §2.2. The replica clients interact with the service, implemented by a group of N replica servers, according to some replication protocol.

The sketcher issues each request through a replica client; the next subsection details the handling of requests. Functionally, the sketcher in Prophecy plays the same role as the per-replica-server sketchers in D-Prophecy. Architecturally, however, its role is quite different. In Prophecy, a fast read is sent only to the single replica server that executes it, and neither the *fast* nor *replicated* interface accesses the history table directly. Thus, the replica group is treated as a black box. Since the sketcher is external to the replica group, writes processed by the group may no longer be visible or discernible to the sketcher; *i.e.*, there may exist an *external write channel*. Since only replica clients interact directly with the replica servers, each replica client can maintain a single, long-lived session with each replica server. Wide-area clients are shielded from any churn in the replica group and are unaware of the replication protocol: the only responses they see are those that have already been accepted by the sketcher.

The type of session used between clients and the sketcher is left open by our design, as it may vary from service to service. For example, services that only allow read or simple write operations (*e.g.*, HTTP GETs and POSTs) may use unauthenticated sessions. A service like Facebook may use authentication only during user login, and use unauthenticated cookie-based sessions after that. Finally, services that store private or protected data, such as an online banking system, may secure sessions at the application level (*e.g.*, using HTTPS). Prophecy's architecture makes it easy to cope with the overhead of client-sketcher authentication, because one can simply add more sketchers if this overhead grows too high (see §5). To achieve the same scale-out effect, traditional BFT systems like PBFT and D-Prophecy would need to add entire replica groups.

3.3.1 Handling a Request

The sketcher stores two additional fields with each entry $(s(q), s(r))$ in the history table: the sequence number σ associated with r, and a 2-bit value b indicating whether $s(q)$ is *whitelisted* (always issued as a fast read), *blacklisted* (always issued as a replicated request), or neither (the default). The sketch $s(r)$ is empty for whitelisted or blacklisted requests. Algorithm 1 describes the processing of a request and is illustrated in Figure 3 (numbers on the right correspond to the numbered steps in the figure).

Prophecy requires a sequence number to be returned by *replicated*, as it seeks to issue concurrent requests to the replica group using multiple replica clients. Concurrency allows reads to execute in parallel to improve throughput. Unfortunately, a sketcher that issues requests concurrently has no way of discerning the correct order of replicated reads by itself, *i.e.*, the order they were processed by the replica group. Thus, it relies on the sequence number returned by *replicated* to ensure that entries in the history table always reflect the latest system state.

The sketcher requires some application-specific knowledge of the format of q and r. This information is used

Algorithm 1 Processing a request at the sketcher.

```
Receive request q from client                                  (1)
if q is a read request then
    (s(q), s(r), σ, b) ← Lookup s(q) in history table
    if (s(r) ≠ null) and (b ≠ blacklisted) then
        r′ ← fast(q)                                           (2)
        if (s(r′) = s(r)) or (b = whitelisted) then
            return r′ to client                                (4)
        end if
    end if
    (r′, σ′) ← replicated(q)                                   (3)
    if (s(r) = null) or (σ′ > σ) then
        Update history table with (s(q), s(r′), σ′, b)
    end if
else
    (r′, σ′) ← replicated(q)                                   (3)
end if
return r′ to client                                            (4)
```

to determine if q is a read or write request, and to discard extraneous or non-deterministic information from q or r while computing $s(q)$ or $s(r)$. For example, in our prototype implementation of Prophecy, an HTTP request is parsed by an HTTP protocol handler to extract the URL and HTTP method of the request; the same handler removes the date/time information from HTTP headers of the response. In practice, the required application-specific knowledge is minimal and limited to parsing protocol headers; the payload of the request or response (*e.g.*, the HTTP body) is treated opaquely by the sketcher.

Whitelisting and blacklisting add flexibility to the handling of requests, but may require additional application-specific knowledge. One use of blacklisting that does not require such knowledge is to dynamically blacklist requests that exhibit a high frequency of transitions (*e.g.*, dynamic content). This allows the sketcher to avoid issuing fast reads that are very likely to fail. (We do not currently implement this optimization.)

3.4 Performance

In our analysis and evaluation, the sketcher is able to accommodate all read requests in its history table without evicting any entries. If needed, a replacement policy such as LRU may be used, but this is unlikely: our current implementation can store up to 22 million unique entries using less than 1GB of memory.

The performance savings of a sketcher come from the ability to execute fast, load-balanced reads whose responses match the entries of the history table. Thus, Prophecy and D-Prophecy are most effective in read-mostly workloads. We can estimate the savings by looking at the cost, in terms of per-replica processing time, of executing a read in these systems. Let t be the probability that a state transition occurs in a given workload. Let C_R be the cost of a replicated read and C_r the cost of a fast read (excluding any sketcher processing in the case of D-Prophecy), and let C_{hist} be the cost of computing a sketch and performing a lookup/update in a history table. Below, we calculate the expected cost of a read in Prophecy and D-Prophecy when used with a BFT replica group that uses PBFT's read optimization. For comparison, we include the cost of the unmodified BFT group; here, t' is the probability that a PBFT optimized read fails.

$$\begin{aligned} \text{Prophecy:} &\quad [C_r + 2C_{hist}] + [t(NC_R + C_{hist})] \\ \text{D-Prophecy:} &\quad [C_r + (N-1)C_{hist}] + [t(NC_R + NC_{hist})] \\ \text{BFT:} &\quad [NC_r] + [t'NC_R] \end{aligned}$$

The addends on the left and right of each equation show the cost of a fast read and a replicated read, respectively. The equations do not include optimizations that benefit all systems equally, such as separating agreement from execution [67]. Prophecy performs two lookups in the history table during a fast read (one before and one after executing the read), and one update to the history table during a replicated read. D-Prophecy performs a history table lookup at all but one replica server during a fast read, and an update to the history table of each replica server during a replicated read. These equations show that Prophecy operates at maximum throughput when there are no transitions, because only one replica server processes each request, as compared to over $2/3$ of the replica servers in the BFT system (assuming, idealistically, that only a necessary quorum of replica servers execute the optimized read, and the remaining replicas ignore it). Since $C_{hist} \ll C_r$ for non-null workloads—the former involves an in-memory table lookup, the latter an actual read—this is a factor of over $(2/3)N$ improvement. D-Prophecy's savings are similar for the same reason. Although t' may be significantly less than t in practice—given that PBFT optimized reads may still succeed even when a state transition occurs—our evaluation in §7 reveals that the benefit of PBFT optimized reads over replicated reads is small for real workloads. Finally, while Prophecy's throughput advantage degrades as t increases, we demonstrate in §8 that t is indeed low for popular web services.

4 Consistency Properties

Despite their relatively simple designs, the consistency properties of Prophecy and D-Prophecy are only slightly weaker than those of the replica group. In this section, we formalize the notion of *delay-once consistency* introduced in §2. Delay-once consistency is a derived consis-

tency model; here, we derive it from linearizability [32], the consistency model of most BFT protocols, and obtain *delay-once linearizability*. Then, we show how Prophecy implements delay-once linearizability.

4.1 Delay-once Linearizability

A history of requests and responses executed by a service is linearizable if it is equivalent to a sequential history [40] that respects the irreflexive partial order on requests imposed by their real-time execution [32]. Request X precedes request Y in this order, written $X \prec Y$, if the response of X is received before Y is sent. Suppose one client sends requests (R^a, W^b, R^c) to the service and another client sends requests (W^d, R^e, R^f, W^g), with partial order $\{R^a \prec R^e, W^g \prec R^c\}$. Then a valid linearized history could look like the following:

$$\langle R_0^a, W_1^d, W_2^b, R_2^e, R_2^f, W_3^g, R_3^c \rangle.$$

The R's and W's represent read and write requests, and subscripts represent the service state reflected in the response to each request (following [28]). In contrast to this history, the following is a valid delay-once linearizable history, though it is not linearizable:

$$\langle R_0^a, W_1^d, W_2^b, R_0^e, R_2^f, W_3^g, R_2^c \rangle.$$

Requests R^e and R^c have stale responses because they do not reflect the state update caused by sequentially precedent writes (note that the staleness of R^e's response is discernible to the issuing client, whereas the staleness of R^c's response is not). At a high level, a delay-once history looks like a linearized history with reads that reflect the state of prior reads, but not necessarily prior writes. The manner in which reads can be stale is not arbitrary, however. Specifically, a history H is *delay-once linearizable* if the subsequence of write requests in H, denoted by $H|_W$, satisfies linearizability, and if read requests satisfy the following property:

> **Delay-once property**. For each read request R_x in H, let R_y and W_z be the read and write request of maximal order in H such that $R_y \prec R_x$ and $W_z \prec R_x$. Then either $x = y$ or $x = z$.

Delay-once linearizability implies both monotonic read and monotonic write consistency, but not read-after-write consistency. If $\prec_H$ is the partial order of the history H, delay-once linearizability respects $\prec_{H|_W}$ but not $\prec_H$, due to the possible presence of stale reads.

The delay-once property ensures two things: first, reads never reflect state older than that of the latest read (they are only *delayed* to *one* stale state), and second, reads that are updated reflect the latest state immediately. Thus, a system that implements delay-once consistency is *responsive*. To verify if a read in a delay-once consistent history H is stale, one can check the following:

> **Staleness indicator**. Given a read request R_x in H, let W_y be the write request of maximal order in H such that $W_y \prec R_x$. R_x is stale if and only if $x < y$.

The staleness property explains why object-based systems like web services fare particularly well with delay-once consistency. In these systems, state updates to one object are isolated from other objects, so staleness can only occur between writes and reads to the same object.

The above derivation of delay-once consistency is based on linearizability, but derivations from other consistency models are possible. For example, a weaker condition called read-after-write consistency also yields meaningful delay-once semantics.

4.2 Prophecy's Consistency Semantics

We now show that Prophecy implements delay-once linearizability when used with a replica group that guarantees linearizability, such as a PBFT replica group. A similar (but simpler) argument shows that D-Prophecy achieves delay-once linearizability, omitted here due to space constraints.

Prophecy inherits the system and network model of the replica group. When used with a PBFT replica group, we assume an asynchronous network between the sketcher and the replica group that may fail to deliver messages, may delay them, duplicate them, or deliver them out-of-order. Replica clients issue requests to the replica group one at a time; requests are retransmitted until they are received. We do not make any assumptions about the organization of the service's state; for example, the service may be a monolithic replicated state machine [39, 58] or a collection of numerous, isolated objects [32]. The sketcher may process requests concurrently. We model this concurrency by allowing the sketcher to issue requests to multiple replica clients simultaneously; the order in which these requests return from replica clients is arbitrary. Updates to service state may not be discernible or visible to the sketcher—*i.e.*, there may exist an external write channel—as discussed in §3.3. We show that Prophecy achieves delay-once linearizability despite concurrent requests and external writers.

Our analysis of Prophecy's consistency requires a non-standard approach because it is the sketcher, not the replica servers, that enforces this consistency, and because fast reads are executed by individual replicas. In particular, we introduce the notion of an *accepted history*. Let H_i for $1 \leq i \leq N$ be the history of all write requests executed by replica server i and all fast read requests executed by i that were accepted by the sketcher. Let R_s be the history of all replicated read requests accepted by the sketcher. An accepted history A_i is the union of H_i and R_s, for each replica server i. The position in A_i of each replicated read in R_s is well defined

because all reads are accepted at a single location (the sketcher) and all replicated requests are totally ordered by linearizability. We claim that the accepted history A_i is delay-once linearizable.

To see this, observe that replicated requests satisfy linearizability because they follow the protocol of the replica group. The sketcher ensures that replicated reads update the history table according to this order by using the sequence numbers returned by the *replicated* interface. Further, the sketcher only accepts a fast read if it reflects the state of the latest replicated read. Since A_i contains all replicated reads accepted by the sketcher (not just those accepted by i), and since accepted fast reads never reflect new state, it follows that all fast reads in A_i must satisfy the delay-once property. While A_i may not contain all write requests accepted by the replica group (*e.g.*, if i is missing an update), this only affects i's ability to participate in replicated reads, and does not violate delay-once linearizability. Thus, we conclude that A_i is delay-once linearizable.

Limiting staleness via load balancing. Stale responses are returned by faulty replica servers or correct replica servers that are out-of-date. We can easily verify if an accepted history contains stale responses by checking the staleness indicator defined in §4. To limit the number of stale responses, the *fast* interface dispatches fast reads from all clients uniformly at random over the replica servers.[2] Let g be the fraction of faulty or out-of-date replica servers currently in the replica group. If g is a constant, then g^k, the probability that k consecutive fast reads are sent to these servers, is exponentially decreasing. For BFT protocols, $g < 2/3$ assuming a worst-case scenario where the maximum number of correct nodes are out-of-date. For a replica group of size 4, the probability that $k > 6$ is less than 1.6%.

5 Scale and Complex Architectures

This section describes extensions to the basic Prophecy model in order to integrate fault tolerance into larger-scale and more complex environments.

Scaling through multiple sketchers. In the basic system model of Prophecy (Figure 3), the sketcher is a single bottleneck and point-of-failure. We address this limitation by using multiple sketchers to build a sketching core, as follows. First, we horizontally partition the global history table, based on $s(q)$'s, into non-overlapping regions, *e.g.*, using consistent hashing [33]. We assign each region to a distinct sketcher, which we refer to as *response sketchers*. The partitioning preserves delay-once semantics because only a single sketcher stores the entry for each $s(q)$. Second, we build a two-level sketching system as shown in Figure 4, where the first tier of *request sketchers* demultiplex client requests. That is, given a request q, any of a small number of request sketchers computes $s(q)$ and forwards q to the appropriate response sketcher. Using a one-hop distributed hash table (DHT) [27, 33] to manage the partitioning works well, given the network's small, highly-connected nature. The response sketchers (the members of this DHT) issue requests to the replica group(s) and sketch the responses, ultimately returning them to the clients. (Importantly, the replica servers in Figure 4 need not be part of a single replica group, but may instead be organized into multiple groups.) The larger number of response sketchers reflects the asymmetric bandwidth requirements of network protocols like HTTP. We evaluate the scaling benefits of multiple response sketchers in §7.7.

Handling sketcher failures. The sketching core handles failure and recovery of sketchers seamlessly, because it can rely on the join and leave protocol of the underlying DHT. Since request sketchers direct client requests, they maintain the partitioning of the DHT. To preserve delay-once semantics, this partitioning must be kept consistent [10, 66] to avoid sending requests from the same region of the history table to multiple response sketchers. Prophecy's support for blacklisting simplifies this task, however. In particular, whenever a region of the history table is being relinquished or acquired between response sketchers, we can allow more than one response sketcher to serve requests from the same region provided the entire region is blacklisted (forcing all requests to be replicated). Once the partitioning has stabilized, the new owner of the region can unset the blacklist bit. As a result, membership dynamics can be handled smoothly and simply, at the cost of transient inefficiency but not inconsistency.

Mediating loosely-coupled groups. A sketching core can be shared by the multiple, loosely-coupled components that typically comprise a real service. Alternatively, components that operate in parallel can use Prophecy via dedicated sketchers. Components that operate in series, such as multi-tier web services, can use Prophecy prior to each tier. However, applying agreement protocols in series introduces nontrivial consistency issues. We leave this problem to future work.

6 Implementation

Our implementation of Prophecy and D-Prophecy is based on PBFT [12]. We used the PBFT codebase given its stable and complete implementation, as well as newer results [6] showing its competitiveness with Zyzzyva and

[2] We assume for simplicity that the random selection is secure, though in practice faulty replica servers may hamper this process. The latter is an interesting problem, but outside the scope of this paper.

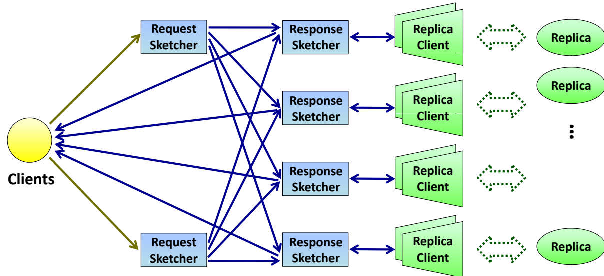


Figure 4: **Scaling out Prophecy using multiple sketchers.**

System	median	1st	99th
pr-PBFT	433	379	706
pr-PBFT-ro	296	255	544
Prophecy	256	216	286
Prophecy-100	617	553	768
PBFT	286	272	309
PBFT-ro	144	135	168
D-Prophecy	144	129	197
D-Prophecy-100	429	412	574

Table 2: **Latency in μs for serial null reads.**

other recent protocols (much more so than was originally indicated [38]). We implemented and compared three proxied systems (Prophecy, proxied PBFT without optimized reads, and proxied PBFT with optimized reads), as well as three non-proxied ("direct") systems (D-Prophecy, PBFT without optimized reads, and PBFT with optimized reads). In our evaluation, we will compare proxied systems only with other proxied systems, and similarly for direct systems, as the architectures and assumptions of the two models are fundamentally different. The proxied systems do not authenticate communication between clients and the sketcher, though they easily can be modified to do so with equivalent overheads.

We implemented a user-space Prophecy sketcher in about 2,000 lines of C++ code using the Tamer asynchronous I/O library [36]. The sketcher forks a process for each core in the machine (8 in our test cluster), and the processes share a single history table via shared memory. The sketcher interacts with PBFT replica clients through the PBFT library. The pool of replica clients available to handle requests is managed as a queue. The sketching function uses a SHA-1 hash [48] over parts of the HTTP header (for requests) and the entire response body (for responses). The proxied PBFT variants share the same code base as the sketcher, but do not perform sketching, issue fast reads, or create or use the history table.

We modified the PBFT library in three ways: to add support for fast reads (about 20 lines of code), to return the sequence numbers (about 20 LOC), and to add support for D-Prophecy (about 100 LOC). Additional modifications enabled the same process to use multiple PBFT clients concurrently (500 LOC), and modified the simple server distributed with PBFT to simulate a webserver and allow "null" writes (500 LOC), as null operations actually have 8-byte payloads in PBFT. We also wrote a PBFT client in about 1000 lines of C++/Tamer that can be used as a client in direct systems and as a replica client in proxied systems.

7 Evaluation

This section quantifies the performance benefits and costs of Prophecy and D-Prophecy, by characterizing their latency and throughput relative to PBFT under various workloads. We explore how the system's throughput characteristics change when we modify a few key variables: the processing time of the request, the size of the response, and the client's session length. Finally, we examine how Prophecy scales with the replica group size.

7.1 Experimental Setup

All of our experiments were run in a 25-machine cluster. Each machine has eight 2.3GHz cores and 8GB of memory, and all are connected to a 1Gbps switch.

The proxied systems are labeled Prophecy, pr-PBFT (proxied PBFT), and pr-PBFT-ro (proxied PBFT with the read optimization). The direct systems are labeled D-Prophecy, PBFT, and PBFT-ro (PBFT with the read optimization). Multicast and batching are not used in our experiments, as they do not impact performance when using read optimizations; all other PBFT optimizations are employed. Unless otherwise specified, all experiments used four replica servers, a single sketcher/proxy machine for the proxied systems, and a single client machine. The proxied experiments used 40 replica clients across eight processes at the sketcher/proxy, and had 100 clients establish persistent HTTP connections with the sketcher/proxy. The direct experiments used 40 clients across eight processes. These numbers were sufficient to fully saturate each system without degrading performance. All experiments use infinite-length sessions between communicating entities (except for the one evaluating the effect of session length). Throughput experiments were run for 30-second intervals and throughput was averaged over each second.

In some experiments, we report numbers for Prophecy-X or D-Prophecy-X, which signifies that the systems experienced state transitions $X\%$ of the time.

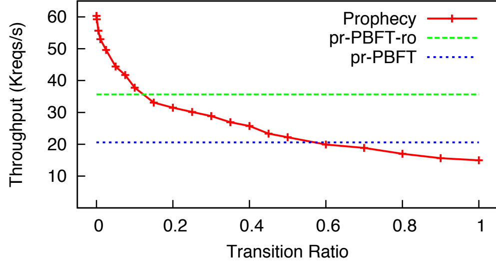


Figure 5: **Throughput of null reads for proxied systems (Prophecy, pr-PBFT, and pr-PBFT-ro).**

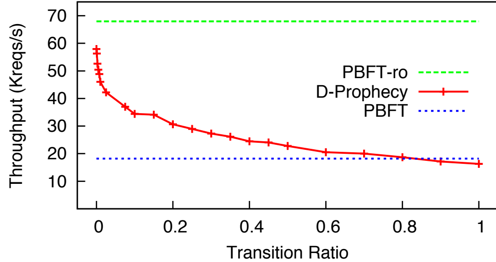


Figure 6: **Throughput of null reads for direct systems (D-Prophecy, PBFT, and PBFT-ro).**

7.2 Null Workload

Latency. Table 2 shows the median and 99th percentile latencies for 100,000 serial null requests sent by a single client. All systems displayed low latencies under $1ms$, although the proxied systems have higher latencies as each request must traverse an extra hop. Prophecy, pr-PBFT-ro, D-Prophecy, and PBFT-ro all avoid the agreement phase during request processing and thus have notably lower latency than their counterparts. Prophecy-100 and D-Prophecy-100 represent a worst-case scenario where every fast read fails and is reissued as a replicated read.

Throughput. Figure 5 shows the aggregate throughput of the proxied systems for executing null requests. We achieve the desired transition ratio by failing that fraction of fast reads at the sketcher.

Since replica servers can execute null requests cheaply, the sketcher/proxy becomes the system bottleneck in these experiments. Nevertheless, Prophecy achieves 69% higher throughput than pr-PBFT-ro due to its load-balanced fast reads, which require fewer packets to be processed by replica servers. As the transition ratio increases, however, Prophecy's advantage decreases because fewer fast reads match the history table. For example, when transitions occur 15% of the time—a representative ratio from our measurement study in §8—Prophecy's throughput is 7% lower than pr-PBFT-ro's.

Figure 6 depicts the aggregate throughput of the direct systems. In this experiment, 40 clients across two machines concurrently execute null requests. D-Prophecy's throughput is 15% lower than PBFT-ro's when there are no transitions, and 50% lower when there are 15% transitions. D-Prophecy derives no performance advantage from its fast reads because the optimized reads of PBFT take no processing time, while D-Prophecy pays the overhead for sketching and history table operations.

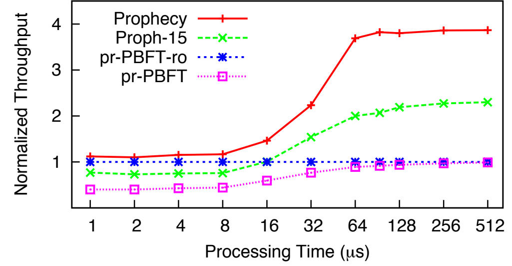


Figure 7: **Throughput of proxied systems as processing time increases, normalized against pr-PBFT-ro.**

7.3 Server Processing Time

The previous subsection shows that when requests take almost no time to process, Prophecy improves throughput only by decreasing the number of packets at each replica server, while D-Prophecy fails to achieve better throughput. However, when the replicas perform real work, such as the computation or disk I/O associated with serving a webpage, Prophecy's improvement is more dramatic.

Figures 7 and 8 demonstrate how varying processing time affects the throughput of proxied systems (normalized against pr-PBFT-ro) and direct systems (normalized against PBFT-ro), respectively. As the processing time increases—implemented using a busy-wait loop—the cost of executing requests begins to dominate the cost of agreeing on their order. This decreases the effectiveness of PBFT's read optimization, as evidenced by the increase in pr-PBFT's throughput relative to pr-PBFT-ro, and similarly between PBFT and PBFT-ro. At the same time, the higher execution costs dramatically increase the effectiveness of load balancing in Prophecy and D-Prophecy. Their throughput approaches 3.9 times the baseline, which is only 2.5% less than the theoretical maximum.

The effectiveness of load-balancing is more pronounced in Prophecy than in D-Prophecy for two main reasons. First, Prophecy's fast reads involve only one replica server, while D-Prophecy's fast reads involve all replicas, even though only a single replica actually executes the request. Second, Prophecy performs sketching and history table operations at the sketcher, whereas D-Prophecy implements such functionality on the replica servers, stealing cycles from normal processing.

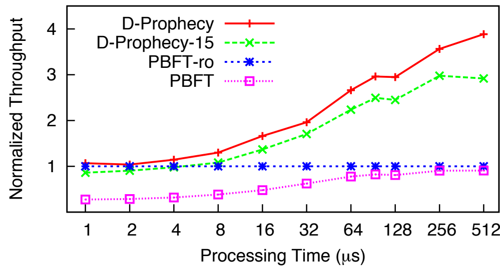


Figure 8: **Throughput of direct systems as processing time increases, normalized against PBFT-ro.**

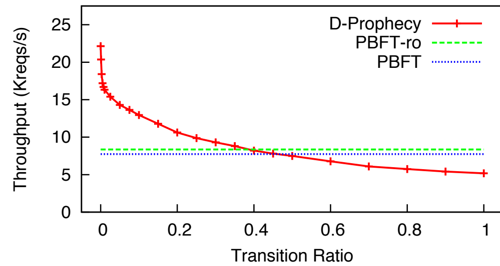


Figure 10: **Throughput of concurrent reads of a 1-byte webpage to Apache webservers for direct systems.**

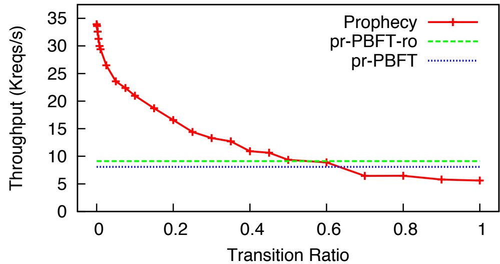


Figure 9: **Throughput of reads of a 1-byte webpage to Apache webservers for proxied systems.**

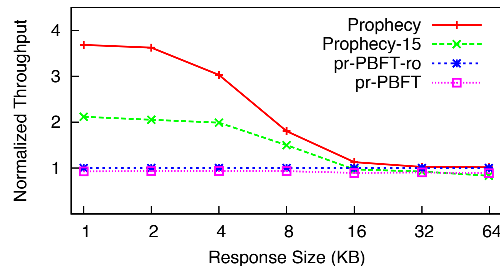


Figure 11: **Throughput of proxied systems as response size increases, normalized against pr-PBFT-ro.**

7.4 Integration with Apache Webserver

We applied Prophecy to a replica group in which each server runs the Apache webserver [7], appropriately modified to return deterministic results. Upon receiving a request, a PBFT server dispatches the request body to Apache via a persistent TCP connection over localhost.

Figure 9 shows the aggregate throughput of the proxied systems for serving a 1-byte webpage. When there are no transitions, Prophecy's throughput is 372% that of pr-PBFT-ro. At the representative ratio of 15%, Prophecy's throughput is 205% that of pr-PBFT-ro. The processing time of Apache is enough to dominate all other factors, causing Prophecy's use of fast reads to significantly boost its throughput.

Figure 10 shows the throughput of direct systems. With no transitions, D-Prophecy's throughput is 265% that of PBFT-ro, and 141% when there are 15% transitions.

In these experiments, the local HTTP requests to Apache took an average of $94\mu s$. For the remainder of this section, we use a simulated processing time of $94\mu s$ within replica servers when answering requests.

7.5 Response Size

Next, we evaluate the proxied systems' performance when serving webpages of increasing size, as shown by Figure 11. As the response size increases, fewer replica clients were needed to maximize throughput. At the same time, Prophecy's throughput advantage decreases as the response size increases, as the sketcher/proxy becomes the bottleneck in each scenario. Increasing the replica servers' processing time shifts this drop in Prophecy's throughput to the right, as it increases the range of response sizes for which processing time is the dominating cost. Note that we only evaluate the systems up to 64KB responses, because PBFT communicates via UDP, which has a maximum packet size of 64KB.

7.6 Session Length

Our experiments with direct systems so far did not account for the cost of establishing authenticated sessions between clients and replica servers. To establish a new session, the client must generate a symmetric key that it encrypts with each replica server's public key, and each replica server must perform a public-key decryption. Given the cost of such operations, the performance of short-lived sessions can be dominated by the overhead of session establishment, as we discussed in §2.2.

Figure 12 demonstrates the effect of varying session length on the direct systems, in which each request per session returns a 1-byte webpage. We find that the throughput of PBFT and PBFT-ro are indistinguishable for short sessions, but as session length increases, the cost of session establishment is amortized over a larger number of requests, and PBFT-ro gains a slight throughput advantage. Similarly, D-Prophecy achieves its full throughput advantage only when sessions are very long.

We do not evaluate the effect of session lengths in the proxied systems, because they currently do not authenti-

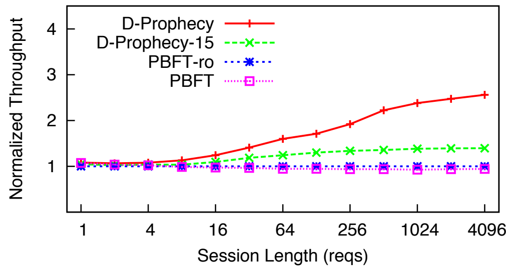


Figure 12: **Throughput of direct systems as session length increases, normalized against PBFT-ro.**

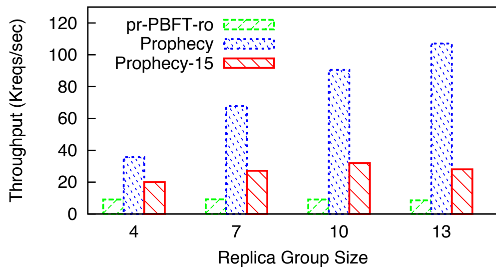


Figure 13: **Throughput of Prophecy and pr-PBFT-ro with varying replica group sizes.**

cate communication with the clients. Authentication can easily be incorporated into these systems, however, at a similar cost to Prophecy and pr-PBFT. That said, proxied systems can better scale up the maximum rate of session establishment than direct systems, as we observed in §3.3: each additional proxy provides a linear rate increase, while direct systems require an entire new replica group for a similar linear increase.

7.7 Scaling Out

Finally, we characterize the scaling behavior of Prophecy and proxied PBFT systems. By increasing the size of their replica groups, PBFT systems gain resilience to a greater number of Byzantine faults (*e.g.*, from one fault per 4 replicas, to four faults per 13 replicas). However, their throughput does not increase, as each replica server must still execute every request. On the other hand, Prophecy's throughput can benefit from larger groups, as it can load balance fast reads over more replica servers. As the sketcher can become a bottleneck in the system at higher read rates, we used two sketchers for a 7-replica group and three sketchers for a 10- and 13-replica group.

Figure 13 shows the throughput of proxied systems for increasing group sizes. Prophecy's throughput is 395%, 739%, 1000%, and 1264% that of pr-PBFT-ro, for group sizes of 4, 7, 10, and 13 replicas, respectively. Prophecy does not achieve such a significant throughput improvement when experiencing transitions, however. We see that a 15% transition ratio prevents Prophecy from handling more than $32,000$ req/s, which it achieves with a replica group of size 10. Thus, under moderate transition rates, further increasing the replica group size will only increase fault tolerance, not throughput.

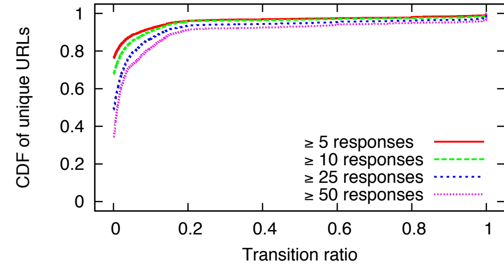


Figure 14: **A CDF of requests over transition ratios.**

8 Measurement Study of Alexa Sites

The performance savings of Prophecy are most pronounced in read-mostly workloads, such as those involving DNS: of the 40K names queried by the ConfiDNS system [52], 95.6% of them returned the same set of IP addresses every time over the course of one day. In web services, it is less clear that transitions are rare, given the pervasiveness of so-called "dynamic content".

To investigate this dynamism, we collected data from the Alexa top 25 websites by scripting a Firefox browser to reload the main page of each site every 20 seconds for 24 hours on Dec. 29, 2008. Among the top sites were `www.youtube.com`, `www.facebook.com`, `www.skyrock.com`, `www.yahoo.co.jp`, and `www.ebay.com`.[3] The browser loads and executes all embedded objects and scripts, including embedded links, JavaScript, and Flash, with caching disabled. We captured all network traffic using the tcpflow utility [19], and then ran our HTTP parser and SHA-1-based sketching algorithm to build a compact history of requests and responses, similar to the real sketcher.

Our measurement results show that transitions are rare in most of the downloaded data. We demonstrate a clear divide between very static and very dynamic data, and use Rabin fingerprinting [55] to characterize the dynamic data. Finally, we isolate the results of individual geographic "sites" using a CIDR prefix database.

8.1 Frequency of Transitions

For each unique URL requested during the experiment, we measured the ratio of state transitions over repeated

[3]While one might argue that BFT agreement is overkill for many of the sites in our study, our examples in the introduction show that Heisenbugs and one-off misconfigurations can lead to embarrassing, high-profile events. Prophecy protects against these mishaps without the performance penalty normally associated with BFT agreement.

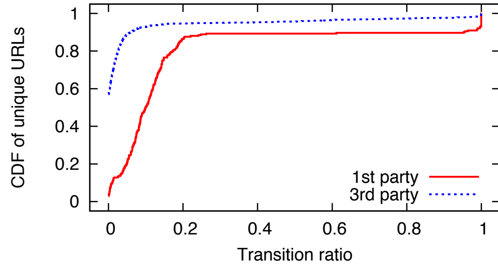


Figure 15: **A CDF over transition ratios of first-party vs. third-party URLs.**

requests. Figure 14 shows a CDF of unique URLs at different transition ratios. We separately plotted those URLs based on the number of requests sent to each one, given that embedded links generate a variable number of requests to some sites. (Where not specified, the minimum number of requests used is 25.) We see that roughly 50% of all data accessed is purely static, and about 90% of all requests have fewer than 15% state transitions. These numbers confirmed our belief that most dynamic websites are actually dynamic compositions of very static content. The same graph scaled by the average response size of each request yields very similar curves (omitted), suggesting that Figure 14 also reflects the total response throughput at each transition ratio.

Figure 15 is the same plot as Figure 14 but divided into first-party URLs, or those targeted at an Alexa top website, and third-party URLs, or those targeted at other sites (given that first-party sites can embed links to other domains for image hosting, analytics, advertising, etc.). The graph shows that third-party content is much more static than first-party content, and thus third-party content providers like CDNs and advertisers could benefit substantially from Prophecy.

The results in this section are conservative for two reasons. First, they reflect a workload of only three requests per minute per site, when in reality there may be tens or hundreds of thousands of requests per minute. Second, many URLs—though not enough to cause space problems in a real history table—saw only a few requests, but returned identical responses, suggesting that our HTTP parser was conservative in parsing them as unique URLs. An important characteristic of all of the graphs in this section is the relatively flat line across the middle: this suggests that most data is either very static or very dynamic.

8.2 Characterizing Dynamic Data

Dynamic data degrades the performance of Prophecy because it causes failed fast reads to be resent as replicated reads. Often, however, the amount of dynamism is small and may even be avoidable. To investigate this, we characterized the dynamism in our data by using Rabin fingerprinting to efficiently compare responses on either side of a transition. We divided each response into chunks of size 1K in expectation [47], or a minimum of 20 chunks for small requests.

Our measurements indicate that 50% of all transitions differ in at least 30% of their chunks, and about 13% differ in all of their chunks. Interestingly, the *edit distance* of these transitions was much smaller: we determined that 43% of all transitions differ by a single contiguous insertion, deletion, or replacement of chunks, while preserving at least half or no more than doubling the number of original chunks. By studying transitions with low edit distance, we can identify sources of dynamism that may be refactorable. For example, a preliminary analysis of around 4,000 of these transitions (selected randomly) revealed that over half of them were caused by load-balancing directives (*e.g.*, a number appended to an image server name) and random identifiers (*e.g.*, client IDs) placed in embedded links or parameters to JavaScript functions. In fact, most of the top-level pages we downloaded, including seemingly static pages like `www.google.com`, were highly dynamic for this exact reason. A more in-depth analysis is slated for future work.

8.3 Site-Based Analysis

A “site” represents a physical datacenter or cluster of machines in the same geographic location. A single site may host large services or multiple services. Having demonstrated Prophecy’s ability to scale out in such environments, we now study the potential benefit of deploying Prophecy at the sites in our collected data. To organize our data into geographic sites, we used forward and reverse DNS lookups on each requested URL and matched the resulting IP addresses against a CIDR prefix database. (This database, derived from data supplied by Quova [54], included over 2 million distinct prefixes, and is thus significantly finer-grained than those provided by RouteViews [57].) Requests that mapped to the same CIDR prefix were considered to be part of the same site. Figure 16 shows an overlay of the transition plots of each site. From the figure, a few sites serve very static data or very dynamic data only, but most sites serve a mix of very static and very dynamic data. All but one site (`view.atdmt.com`) show a clear divide between very static and very dynamic data.

9 Related Work

A large body of work has focused on providing strong consistency and availability in distributed systems. In the fail-stop model, state machine replication typically used primary copies and view change algorithms to improve

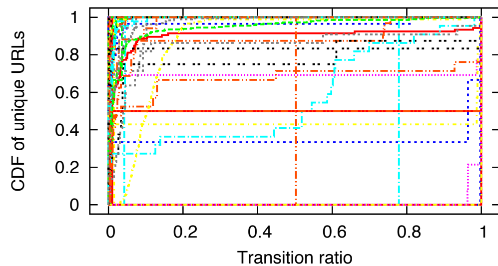


Figure 16: **A CDF of URLs over transition ratios for all sites for which CIDR data was available.**

performance and recover from failures [41, 50]. Quorum systems focused on tradeoffs between overlapping read and write sets [26, 31]. These protocols have been extended to malicious settings, both for Byzantine fault-tolerant replicated state machines [12, 42, 56], Byzantine quorum systems [1, 46], or some hybrid of both [17]. Modern approaches have optimized performance via various techniques, including by separating agreement from execution [67], using optimistic server-side speculation on correct operation [38], reducing replication costs by optimizing failure-free operation [65], and allowing concurrent execution of independent operations [37]. Prophecy's history table is motivated by the same assumption as this last approach—namely, that many operations/objects are independent and hence often remain static over time.

Given the perceived cost of achieving strong consistency and a particular desire to provide "always-on" write availability, even in the face of partitions, a number of systems opted for cheaper techniques. Several BFT replicated state machine protocols were designed with weaker consistency semantics, such as BFT2F [44], which weakens linearizability to fork* consistency, and Zeno [59], which weakens linearizability to eventual consistency. Several filesystems were designed in a similar vein, such as SUNDR [45] and systems designed for disconnected [29, 35] or partially-connected operation [51]. BASE [53] explored eventual consistency with high scalability and partition tolerance; the foil to database ACID properties. More recently, highly-scalable storage systems being built out within data-centers have also opted for cheaper consistency techniques, including the Google File System [25], Yahoo!'s PNUTS [16], Amazon's Dynamo [18], Facebook's Cassandra [20], eBay's storage techniques [61], or the popular approach of using Memcached [23] with a backend relational database. These systems take this approach partly because they view stronger consistency properties as infeasible given their performance (throughput) costs; Prophecy argues that this tradeoff is not necessary for read-mostly workloads.

Recently, several works have explored the use of trusted primitives to cope with Byzantine behavior. A2M [13] prevents faulty nodes from lying inconsistently by using a trusted append-only memory primitive, and TrInc [43] uses a trusted hardware primitive to achieve the same goal. Chun *et al.* [14] introduced a lightweight BFT protocol for multi-core single-machine environments that runs a trusted coordinator on one core, similar in philosophy to Prophecy's approach of extending the trusted computing base to include the sketcher.

Prophecy is unique in its application to customer-facing Internet services and its ability to load-balance read requests across a replica group while retaining good consistency semantics. Perhaps closest to Prophecy's semantics is the PNUTS system [16], which supports a load-balanced read primitive that satisfies timeline consistency (all copies of a record share a common timeline and only move forward on that timeline). Delay-once linearizability is strictly stronger than timeline consistency, however, because it does not allow a client to see a copy of a record that is more stale than a copy the client has already seen (whereas timeline consistency does).

There has been some work on using history as a consistency or security metric for particular applications. Aiyer et al. [4, 5] develop k-quorum systems that bound the staleness of a read request to one of the last k written values. Using Prophecy with a k-quorum system may be synergistic: Prophecy's load-balanced reads are less costly than quorum reads, and k-quorum systems can protect against an adversarial scheduler that attempts to hamper Prophecy's load balancing. The Farsite file system [3] uses historical sketches to validate read requests, but requires a lease-based invalidation protocol to keep sketches strongly consistent. The system modifies clients extensively and requires knowledge of causal dependencies (if these constraints are ignored, then D-Prophecy can easily be modified to achieve the same consistency as Farsite). Pretty Good BGP [34] whitelists BGP advertisements whose new route to a prefix includes its previous originating AS, while other routes require manual inspection. ConfiDNS [52] uses both agreement and history to make DNS resolution more robust. It requires results to be static for a number of days and agreed upon by some number of recursive DNS resolvers. Perspectives [63] combines history and agreement in a similar way to verify the self-signed certificates of SSH or SSL hosts on first contact. Prophecy can be viewed as a framework that leverages history and agreement in a general manner.

10 Conclusions

Prophecy leverages history to improve the throughput of Internet services by expanding the trusted middlebox be-

tween clients and a service replica group, while providing a consistency model that is very promising for many applications. D-Prophecy achieves the same benefits for more traditional fault-tolerant services. Our prototype implementations of Prophecy and D-Prophecy easily integrate with PBFT replica groups and are demonstrably useful in scale-out topologies. Performance results show that Prophecy achieves 372% of the throughput of even the read optimized PBFT system, and scales linearly as the number of sketchers increases. Our evaluation demonstrates the need to consider a variety of workloads, not just null workloads as typically done in the literature. Finally, our measurement study of the Internet's most popular websites demonstrates that a read-mostly workload is applicable to web service scenarios.

Acknowledgments

We thank our shepherd Petros Maniatis for helpful comments on earlier versions of this paper. Siddhartha Sen was supported through a Google Fellowship in Fault Tolerant Computing. Equipment and other funding was provided through the Office of Naval Research's Young Investigator program. None of this work reflects the opinions or positions of these organizations.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP*, Oct. 2005.

[2] B. Adida. Helios: Web-based open-audit voting. In *USENIX Security*, July 2008.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, Dec 2002.

[4] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. On the availability of non-strict quorum systems. In *DISC*, Sept. 2005.

[5] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. Byzantine and multi-writer K-quorums. In *DISC*, Sept. 2006.

[6] L. Alvisi, A. Clement, M. Dahlin, M. Marchetti, and E. Wong. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, Apr. 2009.

[7] Apache HTTP Server. `http://httpd.apache.org/`, 2009.

[8] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *IEEE COMPSAC*, Nov. 1977.

[9] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), 1985.

[10] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, Nov. 2006.

[11] M. Castro. *Practical Byzantine Fault-Tolerance*. PhD thesis, Mass. Inst. of Tech., 2000.

[12] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, Feb. 1999.

[13] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP*, Oct. 2007.

[14] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine Byzantine-Fault Tolerance. In *USENIX Annual*, June 2008.

[15] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. BFT: The time is now. In *LADIS*, Sept. 2008.

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

[17] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, Nov. 2006.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lak-shman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[19] J. Elson. tcpflow—A TCP Flow Recorder. `http://www.circlemud.org/~jelson/software/tcpflow/`, 2009.

[20] Facebook. Facebook release cassandra: A structured storage system on a p2p network. `http://code.google.com/p/the-cassandra-project/`, 2008.

[21] Facebook. Scaling out. `http://www.facebook.com/note.php?note_id=23844338919`, Aug. 2008.

[22] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1), 1999.

[23] B. Fitzpatrick. Memcached: a distributed memory object caching system. `http://www.danga.com/memcached/`, 2009.

[24] Flickr. Flickr phantom photos. `http://flickr.com/help/forum/33657/`, Feb. 2007.

[25] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[26] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, Dec. 1979.

[27] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *NSDI*, Mar. 2004.

[28] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, DEC Systems Research Centre, 1985.

[29] J. Heidemann and G. Popek. File system development with stackable layers. *ACM Trans. Comp. Sys.*, 12(1), Feb. 1994.

[30] J. Hendricks, G. Ganger, and M. Reiter. Low-overhead Byzantine fault-tolerant storage. In *SOSP*, Oct. 2007.

[31] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comp. Sys.*, 4(1), Feb. 1986.

[32] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Sys.*, 12(3), 1990.

[33] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[34] J. Karlin, S. Forrest, and J. Rexford. Pretty Good BGP: Improving BGP by cautiously adopting routes. In *ICNP*, Nov. 2006.

[35] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comp. Sys.*, 10 (3), Feb. 1992.

[36] E. Kohler. Tamer. `http://read.cs.ucla.edu/tamer/`, 2009.

[37] R. Kotla and M. Dahlin. High-throughput Byzantine fault tolerance. In *DSN*, June 2004.

[38] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, Oct. 2007.

[39] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Sys.*, 6(2), 1984.

[40] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), Sept. 1979.

[41] L. Lamport. The part-time parliament. *ACM Trans. Comp. Sys.*, 16(2), 1998.

[42] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Sys.*, 4 (3), 1982.

[43] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: small trusted hardware for large distributed systems. In *NSDI*, Apr. 2009.

[44] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, Apr. 2007.

[45] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, Dec. 2004.

[46] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC*, May 1997.

[47] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, Oct. 2001.

[48] NIS95. *FIPS Publication 180-1: Secure Hash Standard*. Natl. Institute of Standards and Technology, Apr. 1995.

[49] F. E. Notes. Needle in a haystack: efficient storage of billions of photos. `http://www.facebook.com/note.php?note_id=76191543919`.

[50] B. M. Oki and B. H. Liskov. Viewstamped replication: a general primary copy. In *PODC*, 1988.

[51] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, Oct. 1997.

[52] L. Poole and V. S. Pai. ConfiDNS: Leveraging scale and history to improve DNS security. In *WORLDS*, Nov. 2005.

[53] D. Pritchett. BASE: An ACID alternative. *ACM Queue*, 6(3), 2008.

[54] Quova. http://www.quova.com/, 2006.

[55] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Aiken Computation Laboratory, 1981.

[56] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *SOSP*, Oct. 2001.

[57] RouteViews. http://www.routeviews.org/, 2006.

[58] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computer Surveys*, 22(4), Dec. 1990.

[59] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI*, apr 2009.

[60] TechCrunch. Facebook source code leaked. `http://www.techcrunch.com/2007/08/11/facebook-source-code-leaked/`, Aug. 2007.

[61] F. Travostino and R. Shoup. eBay's scalability odyssey: Growing and evolving a large ecommerce site. In *LADIS*, Sept. 2008.

[62] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *SOSP*, Oct. 2007.

[63] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual*, June 2008.

[64] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, Apr. 2009.

[65] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. ZZ: Cheap practical bft using virtualization. Technical Report TR14-08, University of Massachusetts, 2008.

[66] Yahoo! Hadoop Team. Zookeeper. `http://hadoop.apache.org/zookeeper/`, 2009.

[67] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP*, Oct. 2003.