Don’t Settle for Eventual Consistency

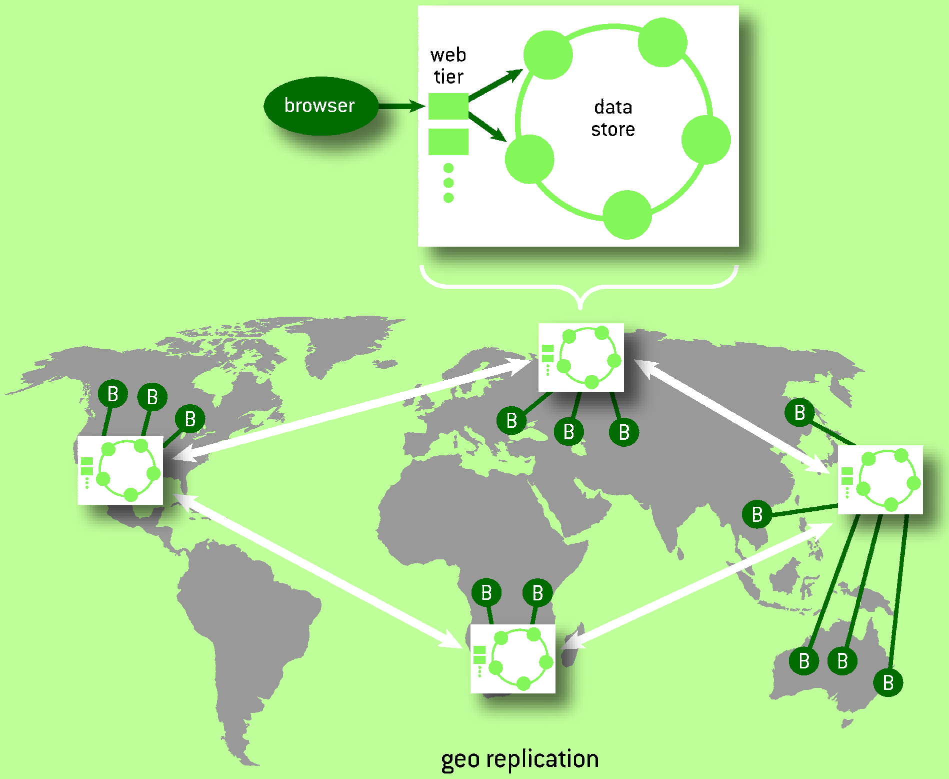
Stronger properties for low-latency geo-replicated storage

Wyatt Lloyd, Facebook
Michael J. Freedman, Princeton University
Michael Kaminsky, Intel Labs
David G. Andersen, Carnegie Mellon University

Geo-replicated storage provides copies of the same data at multiple, geographically distinct locations. Facebook, for example, geo-replicates its data (profiles, friends lists, likes, etc.) to data centers on the east and west coasts of the United States, and in Europe. In each data center, a tier of separate Web servers accepts browser requests and then handles those requests by reading and writing data from the storage system, as shown in figure 1.

Geo-replication brings two key benefits to Web services: fault tolerance and low latency. It provides fault tolerance through redundancy: if one data center fails, others can continue to provide

FIGURE 1 **Geo-Replicated Storage**



the service. It provides low latency through proximity: clients can be directed to and served by a nearby data center to avoid speed-of-light delays associated with cross-country or round-the-globe communication.

Geo-replication brings its challenges, however. The famous CAP theorem, conjectured by Brewer[1] and proved by Gilbert and Lynch,[7] shows it is impossible to create a system that has strong *consistency*, is always *available* for reads and writes, and is able to continue operating during network *partitions*. Each of these properties is highly desirable. Strong consistency—more formally known as *linearizability*—makes programming easier. Availability ensures that front-end Web servers can always respond to client requests. Partition tolerance ensures that the system can continue operating even when data centers cannot communicate with one another. Faced with the choice of at most two of these properties, many systems[5,8,16] have chosen to sacrifice strong consistency to ensure availability and partition tolerance. Other systems—for example, those that deal with money—sacrifice availability and/or partition tolerance to achieve the strong consistency that is necessary for the applications built on top of them.[4,15]

The former choice of availability and partition tolerance is not surprising, however, given that it also enables the storage system to provide low latency—defined as latency for reads and writes that is less than half the speed-of-light delay between the two most distant data centers. A proof that predates the CAP theorem by 14 years[10] shows that it is impossible to guarantee low latency and provide strong consistency at the same time. Front-end Web servers read or write data from the storage system potentially many times to answer a single request; therefore, low latency in the storage system is critical for enabling fast page-load times, which are linked to user engagement with a service—and, thus, revenue. An always-available and partition-tolerant system can provide low latency on the order of milliseconds by serving all operations in the local data center. A strongly consistent system must contact remote data centers for reads and/or writes, which takes hundreds of milliseconds.

Thus, systems that sacrifice strong consistency gain much in return. They can be always available, guarantee responses with low latency, and provide partition tolerance. In COPS (Clusters of Order-preserving Servers),[11] developed for our original work on this subject, we coined the term ALPS for systems that provide these three properties—*always available, low latency,* and *partition tolerance*—and one more: *scalability*. Scalability implies that adding storage servers to each data center produces a proportional increase in storage capacity and throughput. Scalability is critical for modern systems because data has grown far too large to be stored or served by a single machine.

The question remains as to what consistency properties ALPS systems can provide. Before answering this, let's consider the consistency offered by existing ALPS systems. For systems such as Amazon's Dynamo, LinkedIn's Project Voldemort, and Facebook/Apache's Cassandra, the answer is *eventual consistency*.

EVENTUAL CONSISTENCY

Eventual consistency is a widely used term that can have many meanings. Here it is defined as the strongest property provided by all systems that claim to provide it: namely, writes to one data center will eventually appear at other data centers, and if all data centers have received the same set of writes, they will have the same values for all data.

Contrast this with the following part of the definition of strong consistency (linearizability): a total order exists over all operations in the system. This makes programming a strongly consistent storage system simple, or at least simpler: it behaves as a single entity. Eventual consistency does not say anything about the ordering of operations. This means that different data centers can reflect arbitrarily different sets of operations. For example, if someone connected to the West Coast data center sets A=1, B=2, and C=3, then someone else connected to the East Coast data center may see only B=2 (not A=1 or C=3), and someone else connected to the European data center may see only C=3 (not A=1 or B=2). This makes programming eventually consistent storage complicated: operations can appear out of order.

The out-of-order arrival leads to many potential anomalies in eventually consistent systems. Here are a few examples for a social network:

Figure 2 shows that in the West Coast data center, Alice posts, she comments, and then Bob comments. In the East Coast data center, however, Alice's comment has not appeared, making Bob look less than kind. Figure 3 shows that in the West Coast data center, a grad student carefully deletes incriminating photos before accepting an advisor as a friend. Unfortunately, in the East Coast data center, the friend-acceptance appears before the photo deletions, allowing the advisor to see the photos.[3]

Figure 4 shows that in the West Coast data center, Alice uploads photos, creates an album, and then adds the photos to the album, but in the East Coast data center, the operations appear out of order and her photos do not end up in the album. Finally, in figure 5, Cindy and Dave have $1,000 in their joint bank account. Concurrently, Dave withdraws $1,000 from the East Coast data center and Cindy withdraws $1,000 from the West Coast data center. Once both withdrawals propagate to each data center, their account is in a consistent state (-$1,000), but it is too late to prevent the mischievous couple from making off with their ill-gotten gains.

IS EVENTUAL CONSISTENCY THE ONLY OPTION?

Given that theoretical results show that the ALPS properties are incompatible with strong consistency, do we have to settle for eventual consistency? Are we stuck with all the anomalies that come with eventual consistency? No!

Our research systems, COPS[11] and Eiger,[12] have pushed on the properties that ALPS systems can provide. In particular, they provide *causal* consistency instead of eventual, which prevents the first three anomalies. (The fourth anomaly in figure 5 is unfortunately unavoidable in a system that accepts writes in every location and guarantees low latency.) In addition, they provide limited forms of read-only and write-only transactions that allow programmers to consistently read or write data spread across many different machines in a data center.

CAUSAL CONSISTENCY

Causal consistency ensures that operations appear in the order the user intuitively expects. More precisely, it enforces a partial order over operations that agrees with the notion of potential causality. If operation A happens before operation B, then any data center that sees operation B must see operation A first.

Three rules define potential causality:[9]

1. Thread of execution. If *a* and *b* are two operations in a single thread of execution, then *a* -> *b* if operation *a* happens before operation *b*.

2. Reads-from. If *a* is a write operation and *b* is a read operation that returns the value written by *a*, then *a* -> *b*.

FIGURE 2

Anomaly 1: Comment Reordering

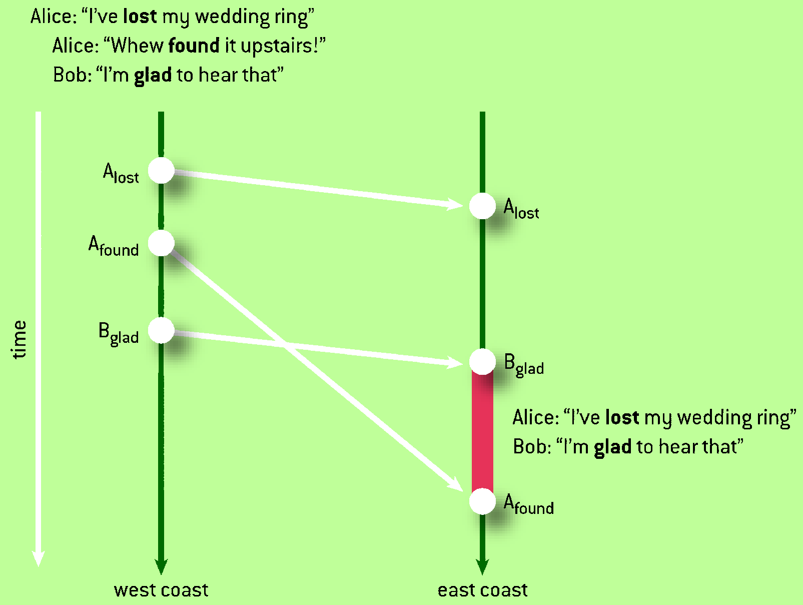
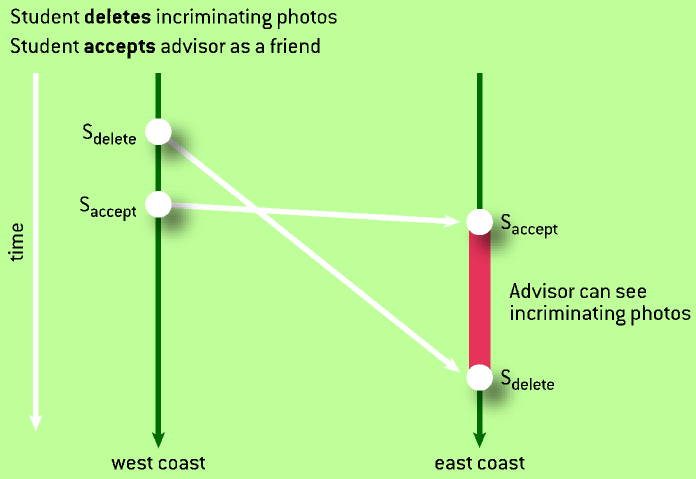

Alice: "I've **lost** my wedding ring"
Alice: "Whew **found** it upstairs!"
Bob: "I'm **glad** to hear that"
time
A_{lost}
A_{found}
B_{glad}
A_{lost}
B_{glad}
Alice: "I've **lost** my wedding ring"
Bob: "I'm **glad** to hear that"
A_{found}
west coast
east coast

FIGURE 3

Anomaly 2: Photo Privacy Violation


Student **deletes** incriminating photos
Student **accepts** advisor as a friend
time
S_{delete}
S_{accept}
S_{accept}
Advisor can see incriminating photos
S_{delete}
west coast
east coast

3. Transitivity. For operations *a*, *b*, and *c*, if *a* -> *b* and *b* -> *c*, then *a* -> *c*. Thus, the causal relationship between operations is the transitive closure of the first two rules.

Causal consistency ensures that operations appear in an order that agrees with these rules. This makes users happy because their operations are applied everywhere in the order they intended. It makes programmers happy because they no longer have to reason about out-of-order operations.

FIGURE 4

Anomaly 3: Broken Photo Album

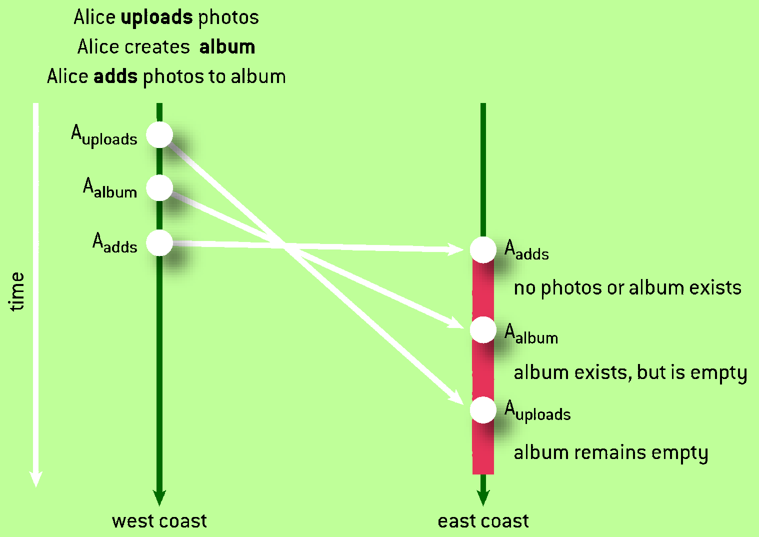
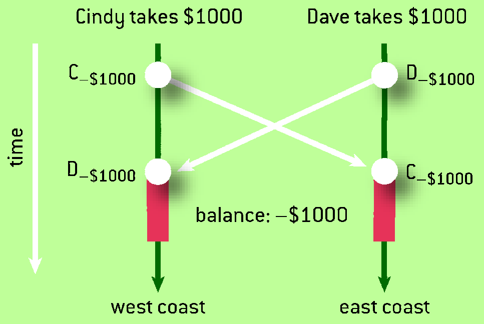

Alice **uploads** photos
Alice creates **album**
Alice **adds** photos to album
time
A_uploads
A_album
A_adds
A_adds
no photos or album exists
A_album
album exists, but is empty
A_uploads
album remains empty
west coast
east coast

FIGURE 5

Anomaly 4: Double Money Withdrawal


Cindy takes $1000
Dave takes $1000
time
C_–$1000
D_–$1000
D_–$1000
C_–$1000
balance: –$1000
west coast
east coast

Causal consistency prevents each of our first three anomalies, turning them into *regularities*.

REGULARITY 1: NO MISSING COMMENTS.

In the West Coast data center, Alice posts, and then she and Bob comment:

Op_1 Alice: I've lost my wedding ring.

Op_2 Alice: Whew, found it upstairs.

Op_3 [Bob reads Alice's post and comment.]

Op_4 Bob: I'm glad to hear that.

Op_1 -> Op_2 by the thread-of-execution rule; Op_2 -> Op_3 by the reads-from rule; Op_3 -> Op_4 by the thread-of-execution rule.

Write operations are only propagated and applied to other data centers, so the full causal ordering that is enforced is Op_1 -> Op_2 -> Op_4.

Now, in the East Coast data center, operations can appear only in an order that agrees with causality. Thus:

Op_1 Alice: I've lost my wedding ring.
Then
Op_1 Alice: I've lost my wedding ring.
Op_2 Alice: Whew, found it upstairs.
Then
Op_1 Alice: I've lost my wedding ring.
Op_2 Alice: Whew, found it upstairs.
Op_4 Bob: I'm glad to hear that.

but never the anomaly that makes Bob look unkind.

REGULARITY 2: NO LEAKED PHOTOS

In the West Coast data center, a grad student carefully deletes incriminating photos before accepting an advisor as a friend:

Op_1 [Student deletes incriminating photos.]
Op_2 [Student accepts advisor as a friend.]
Op_1 -> Op_2 by the thread-of-execution rule.

Now, in the East Coast data center, operations can appear only in an order that agrees with causality, which is:

[Student deletes incriminating photos.]
Then
[Student deletes incriminating photos.]
[Student accepts advisor as a friend.]

but never the anomaly that leaks photos to the student's advisor.

REGULARITY 3: NORMAL PHOTO ALBUM

In the West Coast data center, Alice uploads photos and then adds them to her Summer 2013 album:

Op_1 [Alice uploads photos.]
Op_2 [Alice creates an album.]
Op_3 [Alice adds photos to the album.]
Op_1 -> Op_2 -> Op_3 by the thread-of-execution rule.

Now, in the East Coast data center, the operations can appear only in an order that agrees with causality:

Op_1 [Alice uploads photos.]
Then
Op_1 [Alice uploads photos.]
Op_2 [Alice creates an album.]
Then
Op_1 [Alice uploads photos.]
Op_2 [Alice creates an album.]
Op_3 [Alice adds photos to the album.]

but never in a different order that results in an empty album or complicates what a programmer must think about.

WHAT CAUSAL CONSISTENCY CANNOT DO

Anomaly 4 represents the primary limitation of causality consistency: it cannot enforce global invariants. Anomaly 4 has an implicit global invariant—that bank accounts cannot go below \$0—that is violated. This invariant cannot be enforced in an ALPS system. Availability dictates that operations must complete, and low latency ensures they are faster than the time it takes to communicate between data centers. Thus, the operations must return before the data centers can communicate and discover the concurrent withdrawals.

True global invariants are quite rare, however. E-commerce sites, where it seems inventory cannot go below 0, have back-order systems in place to deal with exactly that scenario. Even some banks do not enforce global \$0 invariants, as shown by a recent concurrent withdrawal attack on ATMs that extracted \$40 million from only 12 account numbers.[14]

Another limitation of causal consistency also stems from the possibility of concurrent operations. Programmers must decide how to deal with concurrent write operations to the same data at different data centers. A common strategy is the last-writer-wins rule in which one concurrent update overwrites the other. For example, a social-network user can have only one birthday. Some situations, however, require a more careful approach. Consider a scenario where Alice has two pending friend requests being accepted concurrently at different data centers. Each accepted friend request should increase Alice's friend count by one. With the last-writer-wins rule, however, one of the increments will overwrite the other. Instead, the two increments must be merged to increase Alice's total friend count by two. With causally consistent storage (as with eventually consistent storage), programmers must determine if the last-writer-wins rule is sufficient, or if they have to write a special function for merging concurrent updates.

The final limitation of causal consistency is that it cannot see or enforce causality outside of the system. The classic example is a cross-country phone call. If Alice on the West Coast updates her profile, calls Bob on the East Coast, and then Bob updates his profile, the system will not see the causal relationship between the two updates and will not enforce any ordering between them.

PROVIDING CAUSAL CONSISTENCY

At a high level, our systems, COPS and Eiger, capture causality through a client library and then enforce the observed ordering when replicating writes to other data centers. The ordering is enforced by delaying the application of a write until all causally previous operations have been applied. This delay is necessary only in remote data centers; all causally previous operations have already been applied at the data center that accepts the write. The client library that tracks causality sits between the Web servers and the storage tiers in each data center. (In current implementations it is on the Web servers.) Individual clients are identified through a special actor_id field in the API to the client library that allows the operations of different users on the same Web server to be disentangled. For example, in a social network the unique user ID could be used as the actor_id.

Let's first describe an inefficient system that provides causality and then explain how to refine it to make it efficient.

Our systems operate by tracking and enforcing the ordering only between write operations. Read operations establish causal links between write operations by different clients, but they are not replicated to other data centers and thus do not need to have an ordering enforced on them. For example, in anomaly/regularity 1, Bob's read (*Op*$_3$) of Alice's post (*Op*$_1$) and comment (*Op*$_2$) creates the causal link that orders Bob's later comment (*Op*$_4$) after Alice's post and comment. A causal link between two write operations is called a *dependency*—the later operation depends on the earlier operation.

Figure 6 shows the relationship between the graph of causality and the graph of dependencies. A dependency is a small piece of metadata that uniquely identifies a write operation. It has two fields: a key, which is the data location that is updated by the write; and a timestamp, which is a globally unique logical timestamp assigned by the logical clock of the server in the data center where it was originally written. Figure 6 illustrates (a) a set of example operations; (b) the graph of causality between them; (c) the corresponding dependency graph; and (d) a table listing dependences with one-hop dependencies shown in bold.

In the initial design the client library tracks the full set of dependencies for each client. Tracking all dependencies for a client requires tracking three sets of write operations:

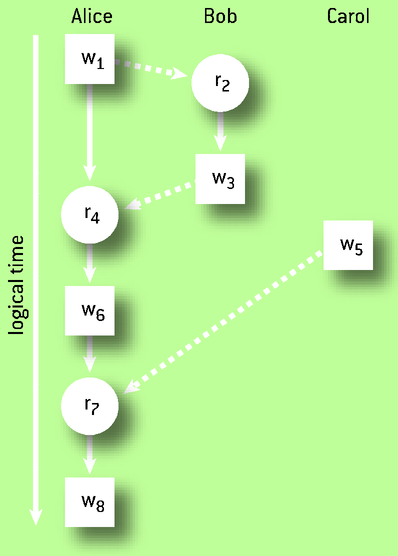
FIGURE 6

Causality and Dependency

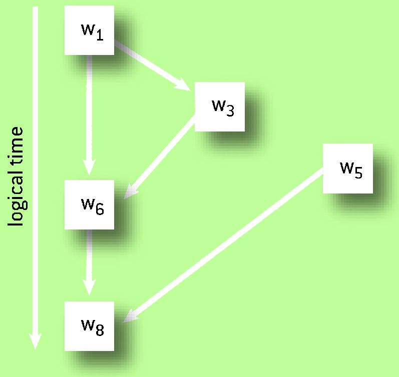
(a)

user	op ID	operation
Alice	w_1	write(Alice:town, NYC)
Bob	r_2	read(Alice:town)
Bob	w_3	write(Bob:town, LA)
Alice	r_4	read(Bob:town)
Carol	w_5	write(Carol:likes, ACM, 8/31/12)
Alice	w_6	write(Alice:likes, ACM, 9/1/12)
Alice	r_7	read(Carol:likes, ACM)
Alice	w_8	write(Alice:friends, Carol, 9/2/12)

(b)



(c)



(d)

op ID	Dependencies
w_1	–
w_3	w_1
w_5	–
w_6	**w_3 w_1**
w_8	**w_6 w_5** w_3 w_1

1. All of the client's previous write operations, because of the thread-of-execution rule.
2. All of the operations that wrote values it read, because of the reads-from rule.
3. All of the operations that the operations in 1 and 2 depend on, because of the transitivity rule.

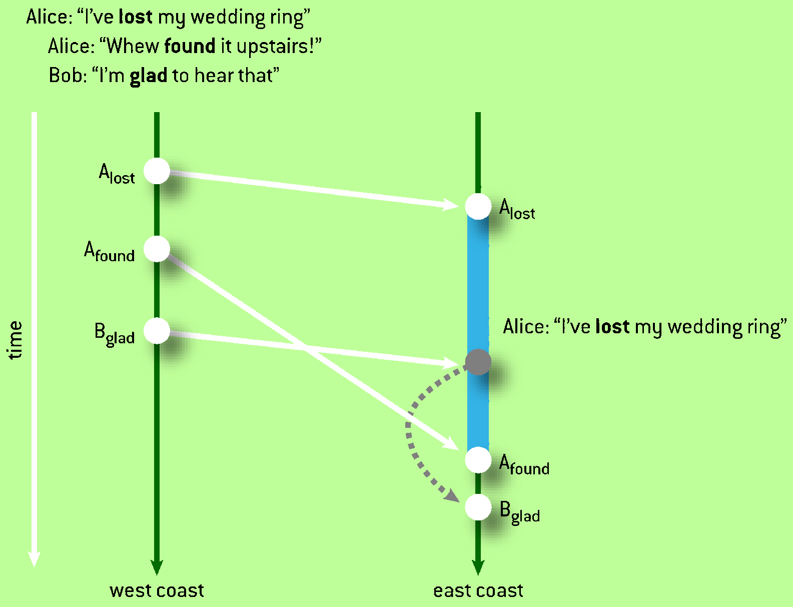
Tracking the first set is straightforward: servers return the unique timestamp assigned to each write to the client library, which then adds a dependency on that write. Tracking the second set is also straightforward: servers return the timestamp of the write that wrote the value when they respond to reads, and then the client library adds a dependency on that write. The third set of operations is a bit trickier: it requires that every write carry with it all of its dependencies, and that these dependencies are stored with the value, returned with reads of that value, and then added to the reader's set of dependencies by the client library.

With the full set of dependencies for each client stored in its client library, all of these dependencies can be attached to each write operation the client issues. Now when a server in a remote data center receives a write with its full set of dependencies, it blocks the write and verifies that each dependency is satisfied. Blocking these replicated write operations is acceptable because they are not client-facing and do not block reads to whatever data they update. Here we have explicitly chosen to delay these write operations until they can appear in the correct order, as shown in figure 7. The dependency check for B_{glad} does not return until after A_{found} is applied on the East Coast, which ensures Bob is never misunderstood.

The system described thus far provides causal consistency and all of the ALPS properties. Causal consistency is provided by tracking causality with a client library and enforcing the causal order with dependency checks on replicated writes. Availability and partition tolerance are ensured by keeping all operations inside the local data center. Low latency is guaranteed by keeping all

FIGURE 7

Regularity 1-Redux


Alice: "I've **lost** my wedding ring"
Alice: "Whew **found** it upstairs!"
Bob: "I'm **glad** to hear that"
time
A_{lost}
A_{found}
B_{glad}
A_{lost}
Alice: "I've **lost** my wedding ring"
A_{found}
B_{glad}
west coast
east coast

operations local, nonblocking, and lock-free. Finally, a fully decentralized design ensures that the system has scalability.

The current system, however, is inefficient. It has a huge amount of dependency metadata that travels around with write operations and a huge number of dependency checks to execute before applying them. Both of these factors steal throughput from user-facing operations and reduce the utility of the system. Luckily, our systems can exploit the transitivity inherent in the graph of causality to drastically reduce the dependencies that must be tracked and enforced. The subset of dependencies being tracked are the *one-hop* dependencies, which have an arc to the current operation in the graph of causality. (Note that in graph-theoretic terms, the one-hop dependencies subset is the direct predecessor set of an operation.) In figure 6 the one-hop dependencies are shown in bold. They transitively capture all of the ordering constraints on an operation. In particular, because all other dependencies are depended upon by at least one of the one-hop dependencies by definition, if this current operation occurs after the one-hop dependencies, then by transitivity it will occur after all others as well.

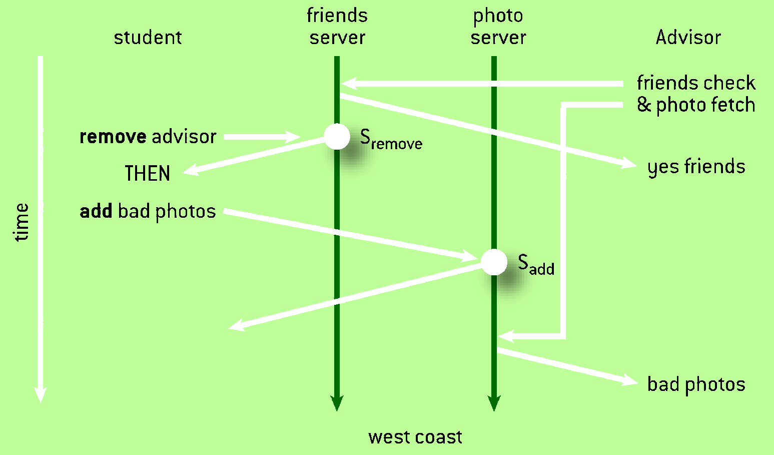
LIMITED TRANSACTIONS

In addition to causal consistency, our systems provide limited forms of transactions. These include read-only transactions, which transactionally read data spread across many servers in a data center, and write-only transactions, which transactionally update data spread across many servers in a data center.

These limited transactions are necessitated—and complicated—by the current scale of data. Data for many services is now far too large to fit on a single machine and instead must be spread across many machines. With data resident on many machines, extracting a consistent view of that data becomes tricky. Even though a data store itself may always be consistent, a client can extract an inconsistent view because the client's reads will be served at different times by different servers. This, unfortunately, can reintroduce many of the anomalies inherent in eventual consistency. In figure 8,

FIGURE 8

Anomaly 5: Leaked Photos



for example, in the West Coast data center, a grad student removes photo-viewing permissions from an advisor and uploads incriminating photos. The advisor concurrently tries to view the student's photos and, incorrectly, is shown the incriminating photos. To avoid these anomalies, causal consistency must be extended from the storage system to the Web servers and then on to users of the service. This can be done using read-only transactions.

Read-only transactions allow programmers to transactionally read data spread across many servers, yielding a consistent view of the data. The interface for a read-only transaction is simple: a list of data locations. Instead of issuing many individual reads for different data locations, a programmer issues a single read for all those locations. This is similar to batching these operations, which is often done to make dispatching reads more efficient—except that it also ensures that the results are isolated.

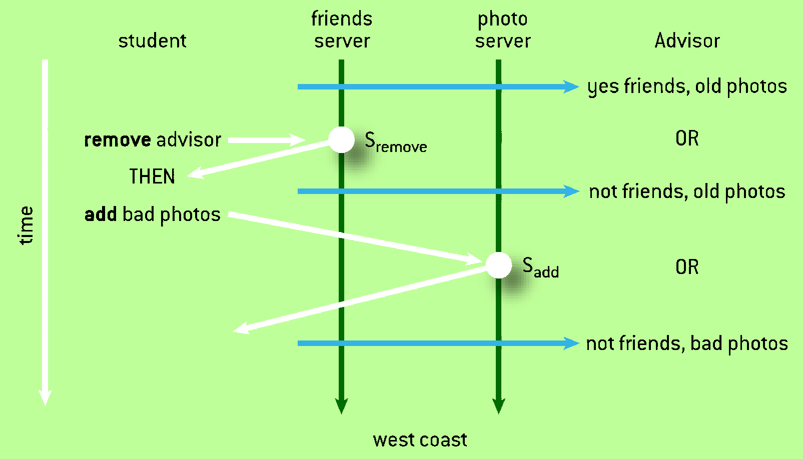
With read-only transactions, anomaly 5 can now be converted into a regularity as well. Figure 9 shows that with read-only transactions, the permissions and photos are read together transactionally, yielding any of the three valid states shown, but never the anomaly that leaks the incriminating photos to the student's advisor.

PROVIDING READ-ONLY TRANSACTIONS

There are many techniques for ensuring isolated access to data locations spread across many machines. The most popular of these include 2PL (two-phase locking), using a TM (transaction manager) to schedule when reads are applied, and maintaining multiple versions of data with MVCC (multiversion concurrency control). The first two approaches are at odds with the ALPS properties. All forms of locking, and 2PL in particular, can encounter locks that are already acquired and then either fail the operation, which gives up on availability, or block the operation until it can acquire the lock, which gives up on low latency. A TM is a centralized entity, and directing all reads though it is a bottleneck that inhibits scalability. This leaves MVCC, and our approach may be viewed as a particularly aggressive variant of it that is possible because our transactions are limited.

FIGURE 9

Regularity 5: No Leaked Photos



The basic idea behind our read-only transaction algorithm is that we want to read the entire distributed data store at a single logical time. (For logical time, each node in the system keeps a logical clock that is updated every time an event occurs (e.g., writing a value or receiving a message). When sending a message, a node includes a timestamp t set to its logical clock c; when receiving a message, a node sets $c \leftarrow max(c, t + 1)$.[9] Logical time LT provides a progressing logical view of the system even though it is distributed and there is no centralized coordination of it. If event a happens before event b, then $LT(a) < LT(b)$. Thus, if distributed data is read at a single logical time LT for all events seen at time t, we know all events that happen before them have lower logical times and thus are reflected in the results. Figure 10 shows an example of this graphically, with validity periods for values, represented by letters, written to different locations.

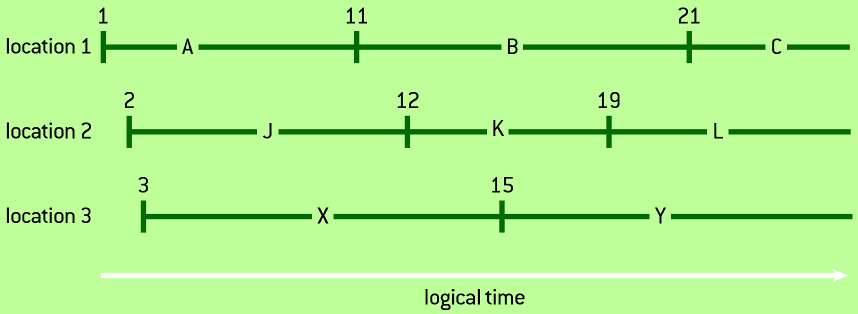
You can determine if values within a set are consistent with one another by annotating them with the logical time they became visible and then were overwritten. For example, in figure 10 consistent sets include {A,J,X}, {B,K,X}, {B,K,Y}, {B,L,Y} and inconsistent sets include {A,K,X}, {B,J,Y}, and {C,L,X}, among others. Our servers annotate values in this way and include them when returning results to the client library so it can determine if values are mutually consistent.

Our read-only transaction algorithm is run by the client library and takes at most two rounds of parallel reads. In the first round, the client library sends out parallel requests for all requested data. Servers respond with their current visible values and validity intervals, which is the logical time the value was written and the current logical time at the server. The value may be valid at future logical times as well, but conservatively we know it is valid for at least this interval. Once the client receives all responses, it determines if all the values are mutually consistent by checking for intersection in their validity intervals. If there is intersection—which is almost always the case unless some of the values are overwritten concurrently with the read-only transaction—then the client library knows the values are consistent and returns them to the client.

If the validity intervals do not all intersect, then the process moves to the second round of the algorithm. The second round begins by calculating a single logical time at which to read values, called the *effective time*. It is calculated by choosing a time that ensures an up-to-date view of the data instead of being stuck on an old consistent cut of it, and it allows the use of many of the values

FIGURE 10

Logical Time



retrieved in the first round. The client library then issues a second round of parallel reads for all data for which it does not have a valid value at the effective time. These second-round reads ask for the value of the data at the effective time, and servers answer these reads by traversing the older version of a value until they find the one that was valid at the effective time. Figure 11 shows the second round in action. Figure 11a is a read-only transaction that completes in a single round, while figure 11b is a read-only transaction that requires a second round and requests data location *1* at the effective time *15* and receives value *B* in response.

This read-only transaction algorithm is specifically designed to maintain all the ALPS properties and provide high performance. It is available because all reads ask for a current value or an old value. It is low-latency because it requires at most two nonblocking rounds of parallel reads. It is partition-tolerant because all reads are in the local data center (partitions are assumed to occur only in the wide area, not in the local data center). It is scalable because it is fully decentralized. Finally, it is performant because it normally takes only a single round of parallel reads and only two rounds of reads in the worst case.

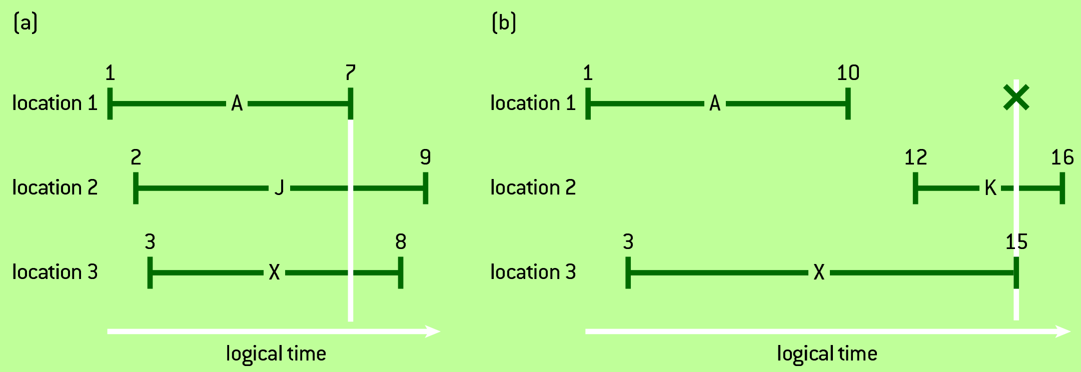
Our previous work on Eiger[12] has more details on how to choose the effective time, how to limit server-side storage of old versions, and an algorithm for write-only transactions that also maintains all the ALPS properties.

THE COST OF CAUSAL CONSISTENCY AND LIMITED TRANSACTIONS

After evaluating Eiger in depth, we reproduce two of our biggest takeaway results here: Eiger has throughput competitive with eventually consistent systems; and it scales to large clusters.

For one realistic view of Eiger's overhead, we parameterized a synthetic workload based upon Facebook's production TAO (The Associations and Objects) system.[2] We then compared Eiger's throughput with that of eventually consistency Cassandra, from which it was forked, in an experiment with clusters of eight servers each in Washington and California. The Cassandra setup achieved 23,657 operations per second that touched 498,239 data locations per second on average. The Eiger setup, with causal consistency and all inconsistent batch operations converted to read or write transactions, achieved 22,891 operations per second that touched 480,904 data locations

FIGURE 11 **Read-Only Transactions**



per second on average. This experiment shows that for this real-world workload Eiger's causal consistency and stronger semantics do not impose significant overhead.

To demonstrate the scalability of Eiger, we ran the Facebook TAO workload on N client machines that fully loaded an N-server cluster that is replicating writes to another N-server cluster (i.e., the N=128 experiment involves 384 machines). This experiment was run on PRObE's Kodiak testbed,[6] which provides an Emulab with exclusive access to hundreds of machines. Figure 12 shows the throughput for Eiger as N scales from eight to 128 servers/cluster. The bars show throughput normalized against the throughput of the eight-server cluster. Eiger scales out as the number of servers increases; each doubling of the number of servers increases cluster throughput by 96 percent on average.

MORE INFORMATION

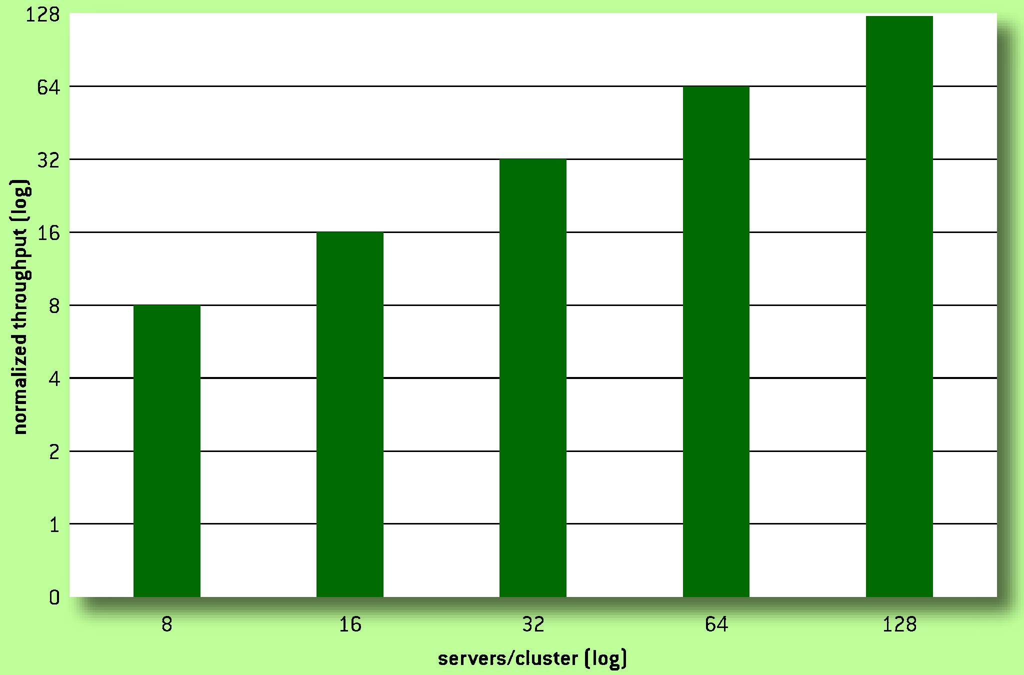
More information is available in our papers on COPS[11] and Eiger,[12] and Wyatt Lloyd's dissertation.[13] The code for Eiger is available from https://github.com/wlloyd/eiger.

ACKNOWLEDGEMENTS

This work was supported by funding from National Science Foundation Awards CSR-0953197 (CAREER), CCF-0964474, CNS-1042537 (PRObE), and CNS-1042543 (PRObE); and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

FIGURE 12

Eiger Scales Linearly



REFERENCES

1. Brewer, E. 2000. Towards robust distributed systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*.
2. Bronson, N., et al. 2013. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the Usenix Annual Technical Conference*.
3. Cham, J. 2013. PhD Comics (June); http://www.phdcomics.com/comics.php?f=1592.
4. Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems* 31(3).
5. DeCandia, G., et al. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*: 205-220.
6. Gibson, G., Grider, G., Jacobson, A., Lloyd, W. 2013. PRObE: A thousand-node experimental cluster for computer systems research. Usenix *;login*: 38(3).
7. Gilbert, S., Lynch, N. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *ACM SIGACT News* 33(2): 51-59.
8. Lakshman, A., Malik, P. 2009. Cassandra—a decentralized structured storage system. In the 3rd ACM SIGOPS International Workshop on Large-scale Distributed Systems and Middleware.
9. Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7): 558-565.
10. Lipton, R. J., Sandberg, J. S. 1988. PRAM: a scalable shared memory. Technical Report TR-180-88. Princeton University, Department of Computer Science.
11. Lloyd, W., Freedman, M. J., Kaminsky, M., Andersen, D. G. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd Symposium on Operating Systems Principles*: 401-416.
12. Lloyd, W., Freedman, M. J., Kaminsky, M., Andersen, D.G. 2013. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th Usenix Conference on Networked Systems Design and Implementation*: 313-328.
13. Lloyd, W. 2013. Stronger consistency and semantics for low-latency geo-replicated storage. Ph.D. Dissertation, Princeton University.
14. Santora, M. 2013. In hours, thieves took $45 million in ATM scheme. *New York Times* (May 9).
15. Sovran, Y., Power, R., Aguilera, M. K., Li, J. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd Symposium on Operating Systems Principles*: 385-400.
16. Voldemort. 2013; http://project-voldemort.com.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

WYATT LLOYD is a postdoctoral researcher at Facebook and is to begin a position as an assistant professor at the University of Southern California in 2014. His research interests include the distributed systems and networking problems that underlie the architecture of large-scale Web sites, cloud computing, and big data. He received his Ph.D. from Princeton University in 2013 and his B.S. from Penn State University in 2007, both in computer science.

MICHAEL J. FREEDMAN is an associate professor of computer science at Princeton University, with a research focus on distributed systems, networking, and security. Recent honors include a Presidential Early Career Award, as well as early investigator awards through the National Science Foundation and Office of Naval Research, a Sloan Fellowship, and DARPA Computer Science Study Group membership.

MICHAEL KAMINSKY is a senior research scientist at Intel Labs and is an adjunct faculty member of the computer science department at Carnegie Mellon University. He is part of the Intel Science and Technology Center for Cloud Computing, based in Pittsburgh, Pennsylvania. His research interests include distributed systems, operating systems, and networking.

DAVID G. ANDERSEN is an associate professor of computer science at Carnegie Mellon University. He completed his S.M. and Ph.D. degrees at MIT, and holds B.S. degrees in biology and computer science from the University of Utah. In 1995, he cofounded an Internet service provider in Salt Lake City, Utah.

© 2014 ACM 1542-7730/14/0300 $10.00