WACCO and LOKO: Strong Consistency at Global Scale

Darrell Bethea
University of North Carolina at Chapel Hill
djb@cs.unc.edu

Michael K. Reiter University of North Carolina at Chapel Hill
reiter@cs.unc.edu

Feng Qian
AT&T Labs Research
fengqian@research.att.com

Qiang Xu
NEC Labs America
qiangxu@nec-labs.com

Z. Morley Mao
University of Michigan
zmao@umich.edu

December 18, 2014

Abstract

Motivated by a vision for future global-scale services supporting frequent updates and widespread concurrent reads, we propose a scalable object-sharing system called WACCO offering strong consistency semantics. WACCO propagates read responses on a tree-based topology to satisfy broad demand and migrates objects dynamically to place them close to that demand. To demonstrate WACCO, we use it to develop a service called LOKO that could roughly encompass the current duties of the DNS and simultaneously support granular status updates (e.g., currently preferred routes) in a future Internet. We evaluate LOKO, including the performance impact of updates, migration, and fault tolerance, using a trace of DNS queries served by Akamai.

1 Introduction

Today's Internet is served by infrastructures that, in general, scale remarkably well to the massive demands placed on them. Both the Domain Name System (DNS) and content-distribution networks (CDNs) are examples of dramatic feats of engineering that facilitate global and quick access to content. The power of these infrastructures, however, derives in part from the largely static nature of the data they serve. DNS scales through caching on the basis of time-to-live (TTL) values that are typically large enough to hide updates from parts of the network for minutes or hours. CDNs serve primarily static data or else data that, if updated, need not be viewed consistently by different parts of the network.

The viability of such approaches may be challenged, however, as the Internet evolves. Multiple visions for future Internet designs anticipate the need to support more dynamic information in the network (e.g., SCION's address and path servers [43], NIRA's NRLS [41], or rendezvous servers to support mobility in content-centric networking [24]), which may enable, e.g., mobile network location, dynamic route control, or diagnosis of network anomalies. Because this information can change quickly — in some cases at the granularity of seconds or less — there is a need for infrastructure services that support dynamic updates, strong consistency, and global scalability. Even for existing uses to direct clients to servers or to exercise route control, today's DNS has limited ability to provide fine-grained control (e.g., [33, 35]), and we expect this shortcoming to become more acute in the future.

In this paper we describe a system called Wide-Area Cluster-Consistent Objects (wacco). wacco manages access to stateful, deterministic *objects* that support invocations of arbitrary types, each of which is either an *update* that may modify object state or a *read* that does not. Objects are managed on a tree-based overlay network of *proxies* that is arranged with respect to geography; i.e., neighbors in the tree tend to be close geographically or, more to the point, enjoy low latency between them. Each client is assigned to a nearby proxy to which it connects to access objects, and object access is managed through a protocol that offers a novel type of consistency that we dub *cluster consistency*. Cluster consistency is strong: it ensures sequential consistency [3, 26] and also that *clusters* of concurrent reads see the most recent preceding update to the object on which the reads are performed. The resulting agreed-upon order and rapid visibility of updates facilitate a wide range of applications, e.g., network troubleshooting, trajectory tracking of mobile nodes, and content-oriented network applications.

wacco achieves scalability via two strategies. First, wacco uses the tree structure of the overlay to aggregate read demand, permitting the responses to some reads to answer others. As such, under high read concurrency, the vast majority of reads are not propagated to the location of the object; rather, most are *paused* awaiting others to complete, from which the return result can be "borrowed". Second, wacco uses migration to dynamically adjust each object's location, permitting the object to follow demand as it fluctuates, e.g., due to diurnal patterns.

To demonstrate and evaluate wacco, we use it to build a service called Low-Overhead Keyspace Objects (loko). loko permits clients to create, modify and query *keyspace* objects. A keyspace is identified by a public key *pk*, and the keyspace for *pk* stores (or generates) mappings, each from a query string *qstr* to a value *val* and bearing a digital signature that can be verified by *pk*. So, querying the keyspace for *pk* for the string `nytimes/publicKey`, e.g., might return the signed public key certificate that the owner of pk believes to be for `nytimes`. Similarly, the query `www/bestRoute` on the keyspace identified by pk' might return a signed mapping indicating the currently preferred route to reach the web server representing the owner of pk'. By iterating queries to a "chain" of keyspaces, each referring the client to the next keyspace in the chain, a client could securely resolve a multipart pathname, much as is done with DNSSEC [2]. In this respect, LOKO could encompass one of the main duties of today's DNS/DNSSEC, while supporting more dynamic mappings due to the consistency provided by WACCO.

In evaluating LOKO (and WACCO), we were handicapped in not having a global workload for such a service. So, we approximated a global workload using a trace of over 4.4 billion DNS requests served by Akamai servers over 36 hours to 83,448 clients in four geographic regions across Asia, North America, and Europe. We used this trace to drive 76-proxy emulations of LOKO with network delays induced to represent a LOKO deployment across these four regions. Our emulations show that LOKO provides good latency for operations, e.g., with up to 89% of reads completing in under 100ms. We also show that LOKO can sustain the full per-proxy query rate represented by the Akamai trace, while guaranteeing cluster consistency. We illustrate the effectiveness of the components of our design using measurements from these emulations.

We begin by presenting related work in Sec. 2. We discuss our design goals in Sec. 3 and present the design of WACCO in Sec. 4. Our evaluation (including a description of LOKO) is in Sec. 5, and we conclude in Sec. 6. In the appendix we define cluster consistency and prove that our protocol implements it.

2 Related Work

The use of a tree-based topology in WACCO for object access is reminiscent of hierarchical caching, which has been studied and deployed extensively for wide-area systems such as the World-Wide Web (e.g., [7, 30, 37]). In some respects, WACCO can be viewed as using *polling-every-time* cache validation [6], in which the authoritative object copy is consulted before returning a cached answer in order to enforce strong consistency. WACCO uses two strategies to reduce the overheads and response latencies induced by such polling: (i) it leverages the tree structure to aggregate polling by many concurrent reads into few messages along the tree. This aggregation also allows WACCO to reduce polling latency by using ongoing polling requests to accelerate others; this strategy has implications for the consistency offered by WACCO, which we characterize precisely; (ii) it migrates the authoritative object copy closer to where demand is largest, an option available to WACCO because it manages the authoritative copy of each object itself, in contrast to web caches that do not.

Many wide-area caching, edge service, and storage designs are also related to our work; space limitations preclude a comparison to all of them. That said, if a replication (or caching) scheme is to prevent conflicting object versions and to make updates available to reads immediately, it must apply reads and updates at a quorum of replicas that intersects the quorum used in another update [13, 18]. Different designs use different quorum systems; e.g., in read-one-update-all quorum systems, every proxy (the update quorum) must be contacted on the critical path of an update. CASCADE [9] is such a system (when configured to use its strongest consistency level). WACCO uses a quorum per object consisting of a single authoritative copy, uses a tree-based overlay to reach this copy, is optimized toward widespread concurrent read load and moderate concurrent update load, and, to our knowledge, offers a new type of consistency achieved by a novel combination of aggregation and migration.

Some designs offer stronger consistency than WACCO. For example, Scatter [15] supports linearizability [21]. However, partly due to its use of distributed hash tables, it does not offer the same benefits of request aggregation and geographic proximity that WACCO achieves through its tree structure and migration. Spanner [10] also implements linearizability, though it does so in part by relying on synchronized real-time clocks, which WACCO does not, and again does not leverage request aggregation. Other systems offer weaker consistency to improve partition-tolerance: e.g., COPS [27] implements causal consistency [1]. Here, we strive for stronger consistency and necessarily[1] presume that partitions in future Internet architectures will be negligibly rare (e.g., due to redundant routing paths [40, 38, 31]).

Our implementation of LOKO as a demonstration of WACCO is motivated by shortcomings of the current DNS for future Internet architectures or even for serving more dynamic data in support of today's mobility and content management (e.g., see [33, 35]). These shortcomings have led to numerous attempts to modify DNS usage (e.g., [39]), to enhance DNS operation (e.g., [8]), to replace it outright with alternative designs (e.g., [23, 11, 35]), and to understand the tradeoffs between new designs and the current DNS (e.g., [34]). CoDoNS [35] is a noteworthy design that, like LOKO, decouples namespace (or keyspace) management from the location and ownership of name servers (in our parlance, proxies) and accelerates the propagation of updates to clients. It provides fast read response via a dynamic replication technique that ensures that a large percentage of requests can be answered immediately by the first proxy to receive the request. However, as in the discussion of quorum systems above, consistency then requires that all of these replicas be updated (or invalidated) when an update occurs, making updates more costly. LOKO is a different point in the design space that anticipates more frequent updates and so strikes a different balance between read and update cost — one that still favors reads particularly when read load is high but that lessens the number of proxies that updates must alter.

[1]Gilbert and Lynch [14] proved that linearizability is impossible to achieve if all operations must return even when partitions occur. The proof applies equally to cluster consistency, the property that WACCO provides.

3 Design Considerations and Goals

We anticipate an object access workload that is generally read-dominated — maybe by orders of magnitude — but that may nevertheless involve frequent and even concurrent updates per object. Updates to an object may be frequent due to the transient nature of the information used to update an object (e.g., the current performance characteristics of a network link), and object updates may be concurrent due to contributions from many parties (e.g., one per link, for an object that calculates preferred routes based on current characteristics of many links). Such workloads temper our willingness to trade update performance for read performance arbitrarily, e.g., as in a typical read-one-update-all system (see Sec. 2). Rather, WACCO takes a more balanced approach that favors read performance but that still limits updates to a single authoritative object copy.

The consistency implemented in WACCO implies sequential consistency [3, 26] (and more, see below). Sequential consistency is a “strong” consistency model: it implies that clients observe update operations to objects in the same total order (cf., [3, Ch. 9]). Sequential consistency implies *causal consistency* [1]: updates related by potential causality [25] (e.g., a client reads an update and then performs another) will be observed by any client in order of their potential causality. But unlike causal consistency, sequential consistency also implies that all clients will observe all updates that are *not* related by potential causality in the same order.

Despite its strength, sequential consistency does not guarantee rapid propagation of updates: in the limit, a client of a sequentially consistent (only) object store is permitted to read the same value for an object for an arbitrarily long period, even if other clients update that object. As such, our goal is to enforce rapid propagation of updates, i.e., updates “take effect” (nearly) immediately. Linearizability [21] strengthens sequential consistency by mandating that an update be observed by any operation on the same object that begins after (in real time) the update operation returns to its caller. However, linearizability comes at substantial performance cost [3, Ch. 9], and so we adopt a weaker requirement that nevertheless strengthens sequential consistency so that updates take effect quickly.

The middleground we adopt allows read operations on the same object to be partitioned into *clusters* of concurrent reads,[2] so that all reads in each cluster return results based on the latest update preceding the cluster in real time (or a more recent update, i.e., one concurrent with the cluster). The resulting consistency property, which we term *cluster consistency*, is weaker than linearizability in that a read returns results based only on updates that preceded the cluster containing it, rather than all updates that precede the individual read. (Updates to the same object are still ordered according to their real-time order, however.) In exchange for this weaker property, we show that cluster consistency can be implemented scalably in wide-area settings by permitting a read to

[2]More specifically, in each cluster, the union of real-time intervals beginning with each read invocation and ending with its return, is contiguous. See App. .2 for details.

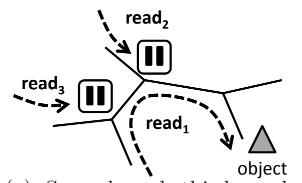
carry responses to other reads in its cluster, thereby accelerating the responses of those reads and reducing load on the authoritative copy.

Cluster consistency still permits a read operation to observe a stale value, but only if the operation is in a cluster of other reads that began before the latest update completed. In this respect, cluster consistency might be viewed as a weakening of linearizability that is analogous to how *obstruction freedom* [20] weakens *wait freedom* [19]: Wait freedom ensures that a client can drive its operation to completion in finitely many steps, whereas obstruction freedom only ensures this property if all other clients are inactive for sufficiently long. Analogously, cluster consistency ensures that a read returns the most recent value if other clients are inactive during this read. We note, however, that WACCO's implementation of cluster consistency limits the other read operations with which a read can be clustered to one per proxy, and so even when other reads are active, the value it returns can be only as stale as that returned by the earliest of these other reads.

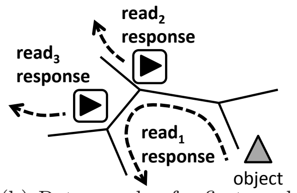
Beyond applications to future Internet designs (see Sec. 1), we also see cluster consistency as potentially useful in nearer-term applications of WACCO, e.g.:

- **Network troubleshooting** Updates from network sensors that publish to WACCO will appear in the same order, enabling consistent diagnosis and actuation of the network by distributed analysis engines. For example, routing anomalies caused by MED oscillation [16] and BGP policy divergence [17] in today's Internet require distributed monitoring to quickly detect and react to an anomaly, e.g., by modifying local routing policies to eliminate the divergent behavior and so to minimize its impact on traffic. A cluster-consistent view of routing updates published to WACCO will make it simpler for distributed monitors to concur on the anomaly and effect changes in policy at multiple locations to rectify the problem. Another example is real-time response to routing pollution, e.g., prefix hijacking [44]. Rapid update propagation and consistent event ordering (e.g., which networks are polluted first) could help reveal the source of pollution and enable a faster reaction to the propagation of polluted routes.
- **Trajectory tracking of mobile nodes** Predicting the future location of a mobile endpoint (e.g., a train) for use in routing (e.g., [32]) would be greatly simplified with a cluster-consistent view of the endpoint's trajectory. For example, if each network appends its name to a WACCO object representing the endpoint's trajectory when the endpoint attaches to the network, cluster consistency implies that the trajectory will be accurate. A weaker property like causal consistency might yield incomplete and even conflicting trajectories, since appends would not be causally related (in the sense of Lamport [25]).
- **Online gaming applications** To keep online games fair to all players, it can be at least as important for users see the same content as it is that the content they see is the most up-to-date [12, 28]. Such applications can be simplified if built on objects that appear to all clients to be modified in the same order.

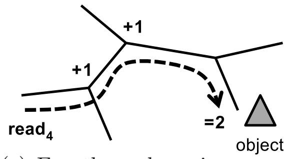
As suggested in Sec. 2 and detailed in Sec. 4, WACCO implements cluster



(a) Second and third reads are paused on the first.



(b) Return value for first read used to resume (respond to) second and third reads.



(c) Fourth read carries aggregated count of recently paused read invocations toward object.

Figure 1: Example of pausing some reads and resuming them later

consistency using a protocol in which each read cluster collectively polls an authoritative object copy before returning responses for the reads it contains. Prior work has generally found polling costlier than cache invalidation (e.g., [6]), and in ongoing work we are investigating scalable designs for implementing cluster consistency using cache invalidations. That said, polling serves dual purposes in WACCO; in addition to consistency, polling messages carry load information to the proxy holding the authoritative object copy, which it uses to determine if the object should be migrated. Migration enables an object to be placed closer to the predominant sources of demand and, as we will show, can significantly reduce response times for operations.

4 wacco Design

The object-sharing protocol that underlies WACCO utilizes a logically tree-structured overlay network that spans a collection of *proxies*. This overlay network should be assembled in a "geographically aware" manner, i.e., so that geographically close (and so presumably well-connected) proxies are also close to one another in the tree. The manner in which a client is paired with a proxy can be decoupled from the rest of our system design; our present design simply leverages a few widely-known proxies to refer each new client to a proxy near it. We assume that each client interacts with only a single proxy at a time, awaiting the completion of any operations it issued to one proxy before switching to another.

The proxies provide clients with access to a set of objects. A client sends a read or update invocation for an object to its proxy and awaits a response from that same proxy. Updates (potentially) modify the object state; reads do not. Our protocol description and proof presume that a read simply returns the current object state, though a proxy can instead return to the client a customized result derived from that state. Sec. 5.1 gives an example of this behavior in the context of LOKO.

4.1 Basic Protocol

WACCO maintains a single authoritative copy of each object. At any point in time, the proxy at which this copy of the object resides is said to *host* the object and, synonymously, to be the *location* of the object. Proxies implement a protocol to route client invocations toward the current location of the object over tree edges (see [36]). Once performed on the object, an operation's response is routed back over the tree to the client that invoked it.

While all update invocations are always routed to the object itself, a read invocation will be *paused* in the tree if the invocation, while en route to the object, encounters a proxy that already forwarded a read request for the same object and has not yet received a response. The paused read will not be forwarded further in the tree; rather, it will be held by the proxy until the response to the invocation on which it paused is returned. When that response arrives, it can serve as the response for any read invocation on the same object that was paused awaiting it and that meets certain conditions described below. In this way, a single read invocation that reaches the object may, in fact, end up serving numerous read requests that are paused on it elsewhere in the tree. This effect is shown in Fig. 1, where the second and third reads are paused waiting on the first (Fig. 1(a)) and then adopt the response to the first read as their own (Fig. 1(b)).

Pausing read requests in this way offers at least two benefits. First, it reduces overall latency in comparison to forwarding each request all the way to the object, since the read request on which another is paused is farther along the path to the object (and so should solicit a response sooner) than the paused read is. I.e., in Fig. 1(a), the first read is at least as close to the object as the second or third read is when each is paused, and a response may even already be traversing the path back. Second, in comparison to forwarding every read request to the object and returning each read response individually, pausing reduces bandwidth use, routing costs to proxies, and computational load on the proxy hosting the object.

Pausing also presents some challenges. First, a paused read constitutes state that a proxy must store until the response for the read on which it is paused returns, possibly opening the door to resource exhaustion. That said, aside from read invocations submitted to a proxy directly by clients, the number of paused reads for an object that a proxy must maintain simultaneously is limited by the number of its neighbors. Reads submitted to a proxy directly by clients (and that are paused) still pose a denial-of-service risk, but it can be managed using any of several techniques (e.g., [22]), and moreover, dropping these read requests as needed can never interfere with other reads (since none are paused on these reads). Resource exhaustion will be discussed further in Sec. 4.4.

Second, pausing erodes the consistency of the protocol, and, indeed, to achieve cluster consistency — and specifically to achieve the sequential consistency that implies — we must restrict which read responses can be used to respond to paused reads. Intuitively, implementing cluster consistency requires that a paused read is not answered by an incoming response that is too outdated. Specifically, as we prove in the appendix, the following conditions suffice to implement cluster consistency: Each read request from a client carries the largest Lamport time [25] at which any update that the client has observed was applied, and each read response carries the Lamport time at which the response was emitted from the authoritative object. A read response that returns to a proxy can be used to satisfy a read request paused at that proxy only if the response's timestamp exceeds the request's timestamp. If this requirement leaves any reads paused at the proxy unsatisfied, then the proxy unpauses one and forwards it along toward the object.

4.2 Caching

Each object state has a *version number* (an integer, initially zero). Applying an update to the object increments that version number. WACCO uses these version numbers to optimize the protocol above as follows.

Each proxy maintains a cache holding at most one cached state per object. The proxy is free to delete states from this cache and manage it using policies independent of those of other proxies. Each read request is augmented to carry a version number. If upon receiving a read request with version number v (new read requests submitted by clients have $v = -1$), a proxy has a version $v' > v$ of the relevant object in cache, then the proxy can increase the read request's version number to v' when forwarding it. If it does so, the proxy is said to have *taken responsibility* for the request and is obligated to retain the cached object state until it has responded to this request. (Our current proxy implementation defaults to taking responsibility; others could do so more selectively.)

When responding to a read request, the proxy hosting the authoritative copy sends the object state (as in Sec. 4.1) if the current object version is larger than the version number in the read request, and sends `same` otherwise. On receiving a response to a read for which a proxy took responsibility, the proxy identifies the latest object version it now has — either the object state in the response or, if the response was `same`, the version in its cache — and responds to paused reads similarly (subject also to the constraints of Sec. 4.1 on Lamport timestamps). I.e., it returns `same` to paused reads bearing the version number of the proxy's latest object version, and it responds with the latest object state to the rest.

A proxy that forwards a read request but that does not take responsibility for it might receive a `same` response, at which point it may not have the latest object version and so would be unable to respond to any paused read bearing an older object version number. Thus, one of these reads is unpaused and forwarded toward the authoritative object, as discussed in Sec. 4.1. Forwarding any read request bearing an old object version number guarantees a response containing the object state, and so when the proxy selects one to unpause and forward, it prefers those with smaller object version numbers.

4.3 Migration

WACCO responds to demand by strategically moving objects among proxies, a process called migration. E.g., a proxy may migrate an object to a neighbor that is forwarding a majority of the invocations for that object, or a proxy that is becoming too heavily loaded may choose to migrate objects away. In this way, migration can be used to reduce load by moving objects closer to areas of greater interest and to otherwise reposition load as needed to deal with hotspots. The former use of migration is particularly beneficial for LOKO (see Sec. 5), since it can be used to position objects to best serve the time zones that are most active at a particular time of day. Moreover, many entities will be accessed with a clear geographic preference — e.g., Chinese websites will likely be accessed mostly from China — and so migration makes sense for positioning such an object near where it is accessed most.

WACCO is not closely tied to the mechanics of migration; it requires only the ability to migrate an object from a proxy to its neighbor between invocations. So while WACCO uses the migration mechanism in Quiver [36], other migration mechanisms would also work. That said, effective migration requires us to resolve two issues. First, we must determine from where an object is currently experiencing the most load; because of paused reads, no single proxy necessarily observes the entire load on an object. Then, we need to determine the specific conditions under which an object should be migrated.

The first issue is resolved in WACCO by appending to each message carrying an object invocation the number of read invocations for that same object that were *recently* paused along the path the message has traveled. If this invocation is paused, the proxy that does so accumulates the message's count into a per-object, per-neighbor counter (using the neighbor that sent the invocation) that the proxy maintains and then further increments this counter by one (for the newly paused invocation). Otherwise, the proxy adds all its counters for this object into the field on the invocation message and forwards the message along toward the object, subsequently zeroing each of these counters. Fig. 1(c) shows an example where the field of a fourth read invocation, initially with value `0`, is updated to `1` at the proxy where $\texttt{read}_3$ was formerly paused and then to `2` as it travels through the proxy at which $\texttt{read}_2$ was formerly paused. In this way, a count of paused reads trickles toward the object at all times, which the hosting proxy can similarly incorporate into per-object, per-neighbor counts of paused invocations. (Update invocations cannot be paused, but the hosting proxy incorporates them into this count, as well, so that updates too are reflected in the load calculations.)

As described so far, this approach for conveying the numbers of paused reads to the proxy holding the object does not adjust these counters for the passage of time, but intuitively such adjustment is necessary — reads paused ten minutes ago should have less bearing on whether to migrate the object than reads paused within the last few seconds. For this reason, each WACCO proxy *decays* its per-object, per-neighbor counters to account for the passage of time before incorporating them into invocation messages bound for the object or calculating whether to migrate an object. In our present implementation, the proxy decays these counters linearly as a function of the time that passed since last unpausing (and returning values for) reads for that object, i.e., the interval between the proxy seeing the last object response and the subsequent object invocation.

Finally, the a proxy holding the object must determine whether to migrate an object and if so, to which of its neighbors. In our implementation, the proxy hosting an object periodically sums its per-neighbor counters for that object and, if one such counter accounts for more than a fraction m of this sum (for a fixed threshold m), then the proxy asks the neighboring proxy corresponding to that counter to migrate the object to it. That neighboring proxy might not do so (e.g., because it is already hosting too many other objects), but otherwise it initiates the object migration. Note that the threshold m value can be different per object, though in our present implementation we use the same m for all objects.

4.4 Resilience

Fault tolerance In WACCO as described so far, a proxy failure would disconnect the tree until the proxy recovers. A generic approach to tolerate proxy failure is to locally replicate each proxy; e.g., in our implementation, each proxy can optionally have a backup to which it commits any meaningful change in internal state [5, §8.2.1] before acting on it. In WACCO, such changes include changes to an object (due to update operations) and changes to internal routing tables (e.g., due to migration). In a straightforward implementation, this primary-backup configuration would double hardware requirements. In practice, we expect clusters of proxies to reside in datacenters in major metropolitan areas, in which case these proxies can provide backup service for others in the same datacenter.

Denial-of-service defense The most acute threat of denial-of-service attacks is interfering with proxy-to-proxy communication. Multi-path routing (e.g., [40, 38, 31]), using private leased lines, or other suitable defenses (e.g., [42]) can mitigate the threat of link overload. Each proxy should also ensure that it reserves adequate resources to retain communication with its neighbor proxies, e.g., by using two network interfaces, one dedicated to proxy-to-proxy communication and the other for serving clients that contact it directly. Moreover, proxies can prioritize tasks for managing inter-proxy activities ahead of those responding to clients and can terminate (or refuse) client requests in favor of retaining proxy-to-proxy communication.

Migration creates the risk of a degradation of service if a flood of read requests can migrate an object (see Sec. 4.3) far from the legitimate demand. If the region of legitimate demand is known in advance, this risk can be mitigated by each object expressing to WACCO its preferences or requirements for where it can be hosted. (This mechanism is also useful to enforce regulatory constraints on where data can reside.) Otherwise, allowing only *authorized* reads (see Sec. 5.1) to influence migration can mitigate this risk.

5 wacco Evaluation

We have implemented WACCO in Java. Our implementation consists of roughly 11,500 physical source lines of code. To evaluate WACCO, we used it to construct a service called LOKO, which we describe in Sec. 5.1. We then describe the traces that we use to induce a realistic workload on this service in Sec. 5.2. We describe our experimental setup in Sec. 5.3 and our results in Sec. 5.4.

5.1 loko

As discussed in Sec. 1, we have used WACCO to implement a service called LOKO that hosts *keyspace* objects. A keyspace is identified by a public key *pk* and stores (or generates) mappings, each from a query string *qstr* to a value *val*. When responding to a query, the keyspace sends the mapping $qstr \rightarrow val$, digitally signed so that it can be verified by *pk*. The signature could be inserted into the keyspace through an update invocation, or the keyspace could produce the signature itself using a private key it holds. The latter strategy might be appropriate for keyspaces that generate responses dynamically.

Generating dynamic responses is useful, e.g., to support CDNs by customizing the content-server address returned in response to a read query. I.e., a keyspace for *pk*, when queried for `nytimes/www/address`, could select the answer *val* from a set of candidate addresses based on load conditions and the address of the client. (This selection would be performed by the proxy directly returning the response to the client.) The cluster consistency offered by LOKO would improve the responsiveness of this mapping to changing conditions over that provided by DNS today (cf., [33]). Of course, keyspaces can also be used to store static mappings, e.g., to addresses or public keys, and keyspaces can be queried iteratively to resolve hierarchical names, analogous to DNS/DNSSEC today.

Any LOKO object can enforce its own access control by checking a signature for each invocation — possibly the same one that it will store and return in response to read invocations later. But by virtue of it having a public key, a keyspace enables the enforcement of coarse access-control policy at the first proxy to receive a request for it, even if that proxy does not host the object. That is, we could extend LOKO so that a proxy, upon receiving a read request for the keyspace identified by *pk* from a client, confirms that the request is accompanied by a delegation credential signed by the owner of *pk* and that authorizes the read. The proxy would do so prior to acting on the read request, dropping it if the check fails. This defense would hinder attempts to migrate the keyspace away from legitimate demand by submitting unauthorized read requests in order to degrade service (see Sec. 4.4). We have not yet implemented this extension, however.

5.2 Traces

The data we used in our evaluation of LOKO (and hence WACCO) are traces of DNS queries received by Akamai, collected for 36 hours beginning 6am, March 9, 2011. In addition to serving DNS queries for domain names of its own, Akamai also serves queries for the domain names of a number of customers. The dataset includes queries of both types and reportedly includes all queries Akamai received during that period by 357 of these (globally distributed) servers.

We stress that the goal of using Akamai data was *not* to evaluate LOKO as a DNS replacement per se but to subject LOKO to a global workload with diurnal patterns and regional object affinities. Thus, when populating objects and generating a workload for our evaluation (see below), we strived primarily to preserve the object-access and client distributions.

5.3 Experimental Setup

Hardware Our experiments consisted of emulations run using 4 servers, each with 64 2.3 GHz cores and 128 GB of RAM. Most emulations used 76 proxies spread across the servers — an average of between 3 and 4 CPUs each. In our fault-tolerance experiments, though, each proxy was given a backup, resulting in 152 proxies on the same hardware.

Proxy placement Recall that the number of servers (with consistency falling short of LOKO) that Akamai dedicates for the load that our traces represent is 357, and so we needed to scale down the Akamai trace to permit a realistic evaluation for 76 proxies. To do this, we selected 4 geographic regions that accounted for $72/357 = 20.2\%$ of all queries in the original trace and allocated 72 proxies to those regions proportionally to the number of requests originating there.[3] (The remaining 4 proxies in our experiments are described below.) The 20.2% of the original trace that we used included 4,460,838,100 queries spanning 1,009,689 domain names and 83,448 clients. Clients at each region were then assigned to that region's proxies to yield a roughly balanced number of queries at each proxy (but while ignoring the contents of those queries). There was one region in Asia, one in Europe, and two in North America, and so we believe this methodology produced a reasonable approximation to a global workload. While client requests drive our experiments, we do not instantiate (or measure) clients themselves, so latency between a client and its proxy is not represented in our measurements, nor are client computational costs. See App. .1 for more statistics about the data and how those statistics informed our experimental setup.

Network latencies To generate the tree topology for our experiments, we added an additional *head proxy* per region and built a minimum spanning tree covering the head proxies using geographical distance as our distance measure.

[3]More precisely, we first geolocated the clients in the Akamai traces using the database from IP2Location (`http://ip2location.com`) and truncated each one's latitude and longitude to an integral value, yielding its "region". We allocated a number of proxies to each selected region proportional to its queries; e.g., if one region originated 10% of the 20.2% of queries selected from the original trace, then it was allocated $10\% \times 72 = 7$ proxies.

Each region's other proxies were then organized in a balanced ternary tree underneath the region's head. So, the total proxies in each experiment was 72+4 = 76, of which only the 72 non-head proxies accepted requests from clients directly. Once the tree was fixed, we estimated latencies between neighboring proxies as a linear function of the geographical distance between them, where this function was calculated using linear regression on real distance/latency pairs.[4] We emulated proxy-to-proxy latencies at user level, using the method implemented in the EmuSockets toolkit [4].[5] We did not limit the bandwidth between proxies, because we do not expect LOKO to even remotely tax the capacity of future networks (or even today's).

Keyspace objects We used the queries selected as described above to populate keyspace objects as follows. Every DNS query indicates a DNS zone, the requested name in that zone, and a query type (e.g., IPv4 host (`A`) record, name server (`NS`) record). We created a keyspace object per zone and initialized it with a field for each name within that zone for which an `A` record was requested (e.g., "`www/A`"), since `A` records were by far the most common form of query. We assigned a random 16-byte value to each such field. We made no effort to represent resource records in keyspaces more explicitly, remembering that the goal of using the Akamai traces is to induce a realistic global workload on LOKO, *not* to make LOKO mimic DNS faithfully. Rather than signing each mapping individually, we compute a Merkle tree [29] over the mappings, signed by the private key corresponding to the keyspace's public key. The Merkle tree is transient; i.e., only the signed root is sent when the keyspace is copied (to support a read) or migrated; the interior nodes are recomputed on demand.

Prior to each measurement run of LOKO, we determined the starting location of each object by executing a warmup. The warmup migrated each keyspace object to its *dominant proxy*, i.e., the proxy that will make the most requests of it during the run. The warmup thus implements an optimal *static* placement of keyspace objects for the run.

Update operations As the Akamai traces include no updates, we introduce them artificially: For a parameter $u \in [0, 1]$, each read operation for a keyspace submitted to its dominant proxy was converted to an update operation with probability u.

If a query was chosen to become an update, an update was generated in its place for the relevant keyspace object, consisting of the relevant query name and query-type string (e.g., "`www/CNAME`"), a 16-byte value, and a 128-byte digital signature on the root of that keyspace's new Merkle tree (i.e., the previous

[4]We took round-trip latencies (ms) from AT&T (see `http://ipnetwork.bgtmo.ip.att.net/pws/current_network_performance.shtml`) on 9 Oct 2011 from Kansas City to 24 other cities in the continental US, as well as from San Francisco to Hong Kong, New York to London, and Washington to Frankfurt. We then obtained distance estimates (miles) for these city pairs. Using simple linear regression, the best fit line to these distance/latency points was $y = 0.019732193x + 8.712212072$ with an R^2 of 0.96820894, indicating a strong goodness of fit. We believe our use of distance-based latencies from within a single provider's network is reasonable, since our service may well be implemented by a major global provider.

[5]This design is an artifact of our trying out several different platforms for our emulations, including some where we were restricted to user-level modifications only.

Merkle tree updated to reflect the newly added or modified field). The accepting proxy verified the signature using the keyspace's public key. Since client costs are not included in our measurements, signature generation for update operations or signature verification after a read were omitted.

Time scaling Though the Akamai trace was 36 hours in length, it would have been impractical to allocate a full 36 hours for each experiment we planned to run. Simply truncating the trace would hide important features, notably any diurnal pattern. As such, we "compacted" the trace as follows, while retaining its features. Each experiment was parameterized by a *sampling rate* $s \in (0, 1]$ and an *acceleration* $a \geq 1$. Each query in the trace was replayed in the experiment independently with probability s, and the trace was accelerated by a factor of a. So, in a period in which the rate of requests in the original trace was q requests per second, sampling reduced this rate to sq requests per second in expectation, and acceleration increased this to sq requests per $1/a$ second in expectation. This method shortens the trace replay to $1/a$ times the original, thereby expediting our tests; in our tests we fixed $a = 48$ so that each test required 45 minutes. However, we sometimes varied the sampling rate s between experiments. It is convenient to describe an experiment in terms of the product sa, which we will call its *load factor*. For example, an experiment with load factor $sa = 0.1$ has an expected request rate of 10% of the original Akamai trace's rate.

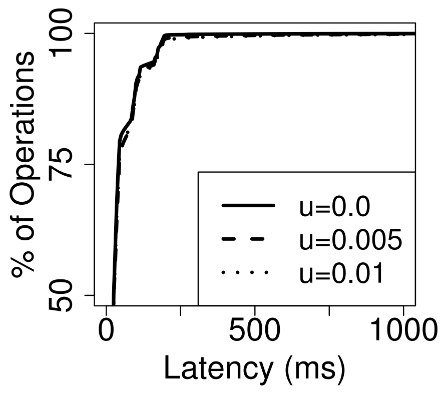
5.4 Experimental Results

All performance numbers in this section were produced using Java SE 7 Server. Unless otherwise noted, we use $m = 0.75$ and load factor 0.1.

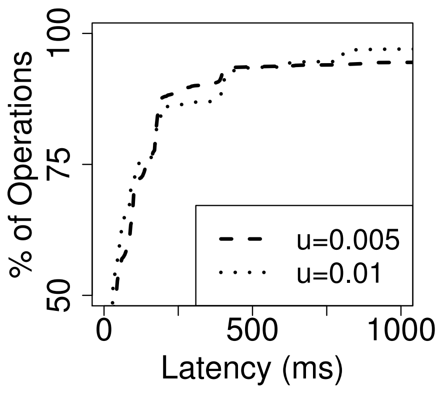
Updates We first explore the impact of varying the fraction of updates in an execution on request latencies. Fig. 2 shows CDFs of operation latencies in experiments for update probabilities $u \in \{0.0, 0.005, 0.01\}$, where $u = 0.0$ implies no updates. Fig. 2(a), shows that as updates increase, read latency increases somewhat, because updates invalidate caches, creating the need for more network traffic. Such cache invalidations also tend apply to larger and popular objects (see App. .1), amplifying the effect. Despite these effects, read latency stays low, with 89.5%, 86.7% and 84.7% of reads completing in under 100ms for $u = 0.0$, 0.005, and 0.01, respectively.

Latencies for the updates themselves appear in Fig. 2(b). These too perform well, with 67.7% and 66.0% completing in under 100ms for $u = 0.005$ and 0.01, respectively. This low latency is partly due to our warmup method, which places objects at the proxy which requests them most, making many updates local (unless the object has been migrated away). Note that this behavior is part of our design — migration moves objects toward the proxies requesting them most.

Migration We show the impact of object migration on operation latency in Fig. 3. Recall that m represents the fraction of the total load for which a neighbor must account in order for migration in the direction of that neighbor to begin. Thus, $m > 1$ is impossible to satisfy and allows no migration at all.



(a) Reads



(b) Updates

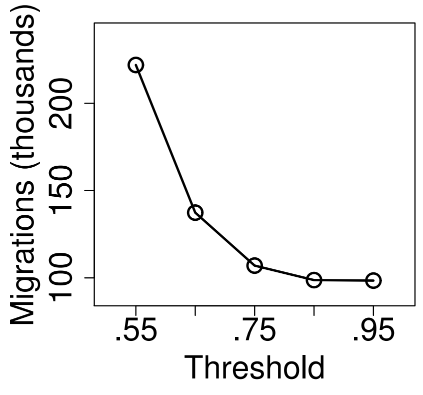
Figure 2: CDFs of latencies (ms) as u varies.

We ran experiments with various migration thresholds: $m = 0.55$ to 0.95 in increments of 0.1, as well as $m > 1$.

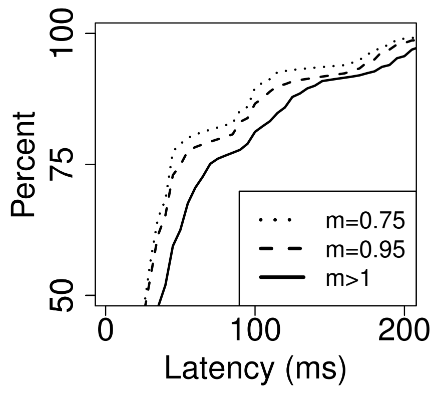
Fig. 3(a) shows the total number of migrations for each value of m, and Fig. 3(b) shows the impact of these migrations on operation latencies. Without migration, 85% of operations finished in less than 120ms. But even with migration enabled at a very conservative threshold ($m = 0.95$), that figure was reduced by 17% to 100ms. Migration at that level also reduced the total number of proxy-to-proxy messages by 19%. Objects migrated within the tree over 110,000 times, resulting in faster response times as well as fewer and smaller network messages.

Reducing m further increases performance. At a very liberal threshold, $m = 0.55$, 85% of operations finished in less than 95ms. In general, the performance differences resulting from different migration thresholds (e.g., $m = 0.55$ vs. $m = 0.95$) are much smaller than the differences between runs with migration and those without it (e.g., $m = 0.95$ vs. $m > 1$), because even a high migration threshold allows objects to move quite close to their areas of demand. If an object is far (in the tree) from the part of the tree where demand for the object is high, then the proxy hosting that object will see that nearly 100% of the load for that object is coming to it from whatever neighbor is in the direction of the load. The host will thus try to migrate the object to that neighbor (see Sec. 4.3). Thus, any migration threshold will allow migration of sufficiently out-of-place objects toward the parts of the tree where they are in demand. The exact value of m only becomes relevant once the object is near enough to its demand that significant fractions of demand for it come from different neighbors. But by that point, objects are already fairly close to the demand, and performance has already improved substantially.

Fault tolerance We measured the effect of fault tolerance on operation laten-



(a) Migration count



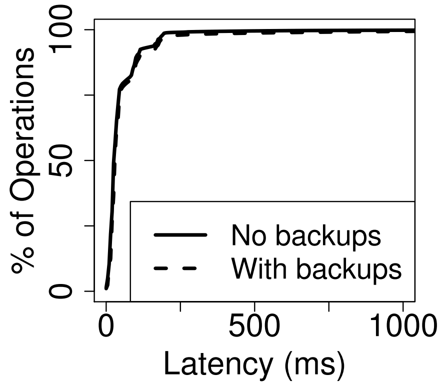
(b) Latency

Figure 3: Impact of varying m, with $u = 0.0$. Lines for some values of m are omitted from Fig. 3(b) for clarity.

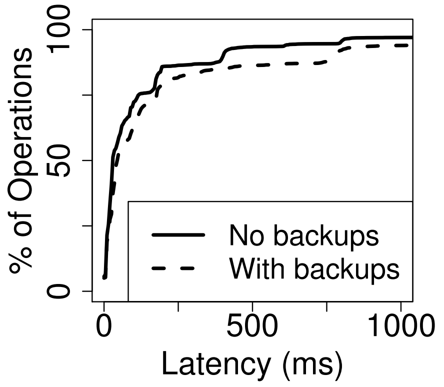
cies when using a backup per proxy (see Sec. 4.4) and $u = 0.01$. Fig. 4 shows the results. As expected, the overhead of fault tolerance is much more evident for update operations, since communication with the backup is on the critical path of each update operation. A possible cause of the added read latency is that we allocated no new hardware to host backups, nor did we reduce the number of primary proxies to make room for their backups. Instead, the primaries and their backups shared the same resources that, in other experiments, were available exclusively to the primaries. Despite the more thinly spread resources and the synchronization costs of the primary-backup protocol, operation latencies with backups were still reasonably close to those without.

Throughput We next present experiments that offer insights into the achievable throughput of our system. In these tests, we increased the sampling rate s and so the load factor, up to a load factor of 1.0, i.e., the same query rate per proxy as Akamai supported in the original trace. Fig. 5(a) shows the achieved throughput in operations per second with $u = 0.01$. This figure shows that our LOKO implementation absorbs the full per-proxy query rate of the Akamai trace. Fig. 5(b) illustrates one reason behind this throughput, namely that as the operation rate increases, the efficacy of read pausing also increases, since more reads are concurrent. This increase in read pausing then results in a reduced number of messages needed per operation, on average (Fig. 5(b)). Finally, Fig. 5(c) shows that the average number of proxy-to-proxy hops a read request travels before it is paused or reaches the object is stable, even as the load factor increases. At load factor 1.0, each read request travels less than 1.1 hops on average.

Consistency To better show the performance gain from cluster consistency, we built a linearizable version of LOKO that we subject to the same load as the cluster-consistent version. Since cluster consistency itself represents a very spe-

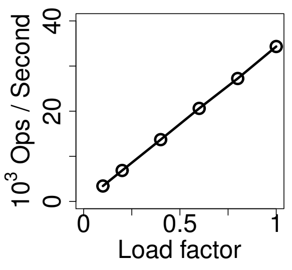


(a) Reads

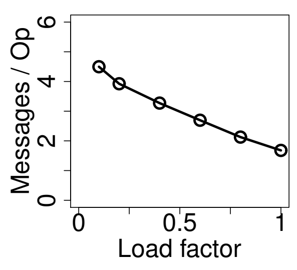


(b) Updates

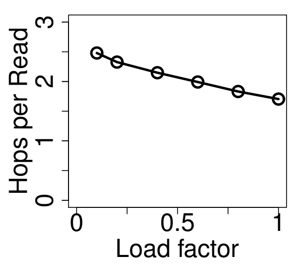
Figure 4: CDFs of latencies (ms) when using backups, with $u = 0.01$.



(a) Throughput

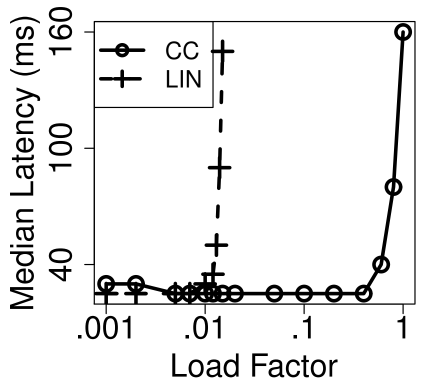


(b) Message cost

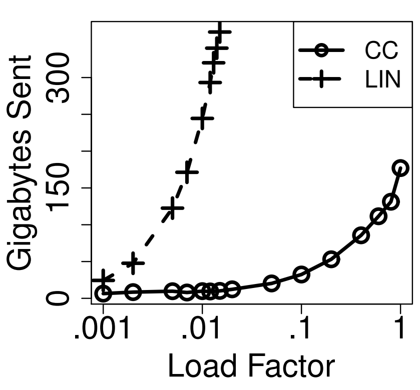


(c) Read hops

Figure 5: Throughput and messaging overhead as load factor varies, with $u = 0.01$.



(a) Median latency



(b) Bytes sent

Figure 6: Impact of varying load factor on median latency and total bytes sent in both the cluster consistent (CC) and linearizable (LIN) versions of LOKO.

cific weakening of linearizability, we can revert to a linearizable version of LOKO with few changes. For clarity, in this section we will refer to the linearizable version of LOKO as LIN-LOKO and the standard, cluster consistent version of LOKO as CC-LOKO.

The main change is that LIN-LOKO lacks read clusters. Instead, each read request passes through the tree all the way to the object (as is done in Quiver [36]), even if other requests are concurrent for the same object. Since read clusters contribute significantly to LOKO's scalability, we expect LIN-LOKO to succumb to heavy loads far sooner than CC-LOKO does.

Lacking clusters, LIN-LOKO can make a minor optimization. Instead of forwarding responses back through the tree, the hosting proxy can reply directly to the original proxy. Recall from Sec. 4.1 that the main reason responses returned through the tree was so that they could answer paused reads along the way, which does not apply to LIN-LOKO.

A final LIN-LOKO change is that, since no responses are sent through the tree, proxies forwarding requests no longer update their object version numbers — it is pointless for a proxy along the request path to take responsibility (see Sec. 4.2) for a request if the response never actually reaches that proxy.

Fig. 6 gives the results of our comparison between CC-LOKO and LIN-LOKO. Fig. 6(a) shows the median request latency vs. load factor, with each point representing a single run at the given configuration. The graph shows that the two versions perform essentially equally at load factors of .012 and below. As the load factor increases to .015, LIN-LOKO quickly climbs to a median latency of 150 ms. We were unable to complete LIN-LOKO runs with load factors higher than .015 due to the massive computational burden placed in the proxies. In contrast, the median latency for CC-LOKO remains relatively constant until a load factor of .6. Then it begins to rise, only reaching a median latency of 160 ms at load factor 1 — equivalent to the full Akamai load. In both versions, increasing load factor eventually causes a spike in median latency, but the CC-LOKO spike occurs at load factors almost two orders of magnitude higher than in the LIN-LOKO case.

Fig. 6(b) shows that the total amount of data sent in the two cases. The graph clearly shows that LIN-LOKO sends far more traffic than CC-LOKO, due to the lack of read clustering. For example, at load factor 1, CC-LOKO sent 177 GB. In contrast, LIN-LOKO sent 171 GB at only .007, a difference of 2–3 orders of magnitude. Together, these results show that the relaxation of consistency — bounded though it is in our case — can indeed lead to dramatic improvements in scalability.

5.5 Limitations

The Akamai data we used in our experiments is the best data we have found for a realistic, global workload. That said, it is important to recognize that this dataset has limitations for the purposes it is used here. First, Akamai customers tend to be large organizations for which domain-name query activity might be heavier and more widespread than most domain names not served by Akamai or than other objects that one might envision in a future application (e.g., a mobile device's location). This tendency might yield an overly optimistic evaluation of LOKO, since it makes more opportunities to aggregate (i.e., pause) reads in the tree, but it also might yield an overly conservative evaluation, since global demand reduces the ability to improve access latencies through migration. Second, as already noted, the Akamai dataset contains no update operations, and so it was necessary to fabricate them.

6 Conclusion

This paper describes the design and evaluation of WACCO, a system for implementing object-based services that need to support both frequent updates and widespread, massive read demand with strong consistency. A contribution of our work is a novel type of strong consistency dubbed *cluster consistency*, which implies both sequential consistency and rapid update propagation and, we argue, can be useful in a range of future networked applications. We used WACCO to implement a service called LOKO that supports keyspace objects and, in one style of usage, could roughly encompass the current duties of DNSSEC. Our evaluation using an emulated global topology and trace of DNS queries to Akamai shows that LOKO provides good responsiveness and can scale to large demand. Through our evaluation, we also documented the importance of object migration and read pausing (and hence cluster consistency) to the performance LOKO achieves.

References

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

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS security introduction and requirements. RFC 4033, March 2005.

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., second edition, 2004.

[4] M. Avvenuti and A. Vecchio. Application-level network emulation: The EmuSocket toolkit. *J. Netw. Comp. Appl.*, 29(4), 2006.

[5] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems, 2nd edition*, pages 199–216. Addison-Wesley, 1993.

[6] P. Cao and C. Liu. Maintaining strong cache consistency in the World Wide Web. *IEEE Trans. Computers*, 47(4), 1998.

[7] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *USENIX ATC*, 1996.

[8] X. Chen, H. Wang, S. Ren, and X. Zhang. Maintaining strong cache consistency for the Domain Name System. *IEEE Trans. Knowledge and Data Engineering*, 19(8), 2007.

[9] G. Chockler, R. Friedman, and R. Vitenberg. Consistency conditions for a corba caching service. DISC '00, 2000.

[10] 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 *10th USENIX OSDI*, 2012.

[11] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a peer-to-peer lookup service. In *1st Intern. Wkshp. Peer-to-Peer Syst.*, 2002.

[12] E. Cronin, B. Filstrup, A. B. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *1st Wkshp. Netw. Syst. Support for Games*, 2002.

[13] D. K. Gifford. Weighted voting for replicated data. In *7th ACM SOSP*, 1979.

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

[15] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *23rd ACM SOSP*, 2011.

[16] T. Griffin and G. Wilfong. Analysis of the MED oscillation problem in BGP. In *IEEE ICNP*, 2002.

[17] T. G. Griffin and G. Wilfong. An analysis of BGP convergence properties. In *ACM SIGCOMM*, 2009.

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

[19] M. P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1), 1991.

[20] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd ICDCS*, 2003.

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

[22] A. Juels and J. Brainard. Client puzzle: A cryptographic defense against connection depletion attacks. In *5th ISOC NDSS*, 1999.

[23] J. Kangasharju and K. W. Ross. A replicated architecture for the Domain Name System. In *19th IEEE INFOCOM*, 2000.

[24] D. Kim, J. Kim, Y. Kim, H. Yoon, and I. Yeom. Mobility support in content centric networks. In *2nd Wkshp. Inform.-Centric Netw.*, 2012.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21, 1978.

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

[27] 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 *23rd ACM SOSP*, 2011.

[28] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: Providing consistency for replicated continuous applications. *IEEE Trans. Multimedia*, 6(1), 2004.

[29] R. C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Department of Electrical Engineering, Stanford University, 1979.

[30] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching: Towards a new global caching architecture. *Comp. Netw. and ISDN Syst.*, 30, 1998.

[31] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path splicing. In *ACM SIGCOMM*, 2008.

[32] J. Paek, K. Kim, J. P. Singh, and R. Govindan. Energy-efficient positioning for smartphone applications using cell-ID sequence matching. In *9th MobiSys*, 2011.

[33] J. Pang, A. Akella, A. Shaikhy, B. Krishnamurthyz, and S. Seshan. On the responsiveness of DNS-based network control. In *Internet Measurement Conf.*, 2004.

[34] V. Pappas, D. Massey, A. Terzis, and L. Zhang. A comparative study of the DNS design with DHT-based alternatives. In *25th IEEE INFOCOM*, 2006.

[35] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the Internet. In *ACM SIGCOMM*, 2004.

[36] M. K. Reiter and A. Samar. Quiver: Consistent object sharing for edge services. *IEEE TPDS*, 19(7), 2008.

[37] P. Rodriguez, C. Spanner, and E. W. Biersack. Analysis of web caching architectures: Hierarchical and distributed caching. *IEEE/ACM Trans. Networking*, 9(4), 2001.

[38] X. Wang and D. Wetherall. Source selectable path diversity via routing deflections. In *ACM SIGCOMM*, 2006.

[39] Y. Wu, J. Tuononen, and M. Latvala. Performance analysis of DNS with TTL value 0 as location repository in mobile Internet. In *IEEE Wireless Comm. and Netw. Conf.*, 2007.

[40] W. Xu and J. Rexford. MIRO: Multi-path Interdomain ROuting. In *ACM SIGCOMM*, 2006.

[41] X. Yang, D. Clark, and A. W. Berger. NIRA: A new inter-domain routing architecture. *IEEE/ACM Trans. Netw.*, 15(4), 2007.

[42] X. Yang, D. Wetherall, and T. Anderson. TVA: A DoS-limiting network architecture. *IEEE/ACM Trans. Networking*, 16(6), 2008.

[43] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. Andersen. SCION: Scalability, control, and isolation on next-generation networks. In *IEEE Symp. Security & Privacy*, 2011.

[44] Zheng Zhang, Ying Zhang, Y. Charlie Hu, and Z. Morley Mao. iSPY: Detecting IP prefix hijacking on my own. In *ACM SIGCOMM*, 2008.

.1 Namespace statistics

In general, most namespaces in the Akamai dataset are small (i.e., having few keys) and represent only a small fraction of the total number of queries. However, Fig. 7(a) shows that there are a few keyspaces which represent a significant portion of the total requests. The most frequently queried keyspace object comprised over 14% of the total, and the 5 most frequently queried keyspace objects comprised over one third of all requests. The distribution of keyspace sizes was also far from uniform, as shown in Fig. 7(b). While over 88% of all keyspaces contained less than 10 keys, some contained over one million.

In Sec. 5.3, we explained that the warmup phase of our experiments places each keyspace object at its dominant proxy. Fig. 7(c) shows that the request rate by the dominant proxy for a keyspace is strongly correlated with the request rate by other, non-dominant proxies for that keyspace, implying that operation workloads will be dominated by nonlocal operations in any static placement of keyspaces — that is, our placement of keyspaces at their dominant proxies at the start of each experiment does not significantly reduce their demand by remote proxies.

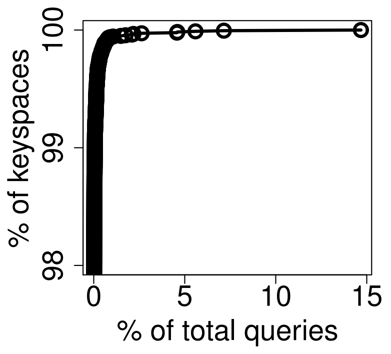
The wide range of query counts per keyspace also means that update operations were not uniformly spread across keyspace objects but instead were concentrated in those that were also read most often, including read most often from non-dominant proxies (again, see Fig. 7(c)). So, update operations caused many caches to become invalid and thus many copies of objects to be sent, and, because the keyspaces accessed the most often tended to be larger (Fig. 7(d)), these sent objects also tended to be large.

.2 Definition of Cluster Consistency

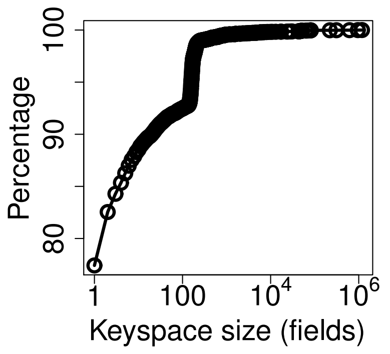
.2.1 Definition

Here we define *cluster consistency*. An *object* consists of state and a set of methods that can be *invoked*. Each invocation returns a *response*, and an invocation/response pair is called an *operation*. Correct behavior of the object is defined by its *sequential specification*, which specifies the return results of operations invoked sequentially on the object.

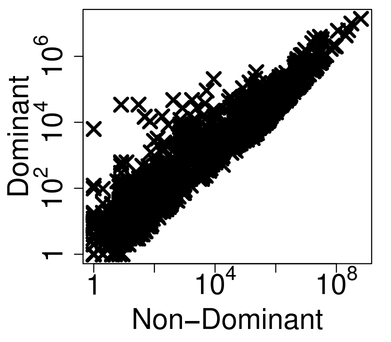
We use op to denote any operation, and $r\text{-}op$ or $u\text{-}op$ denote a read or update operation, respectively. The invocation and response for any op occur at distinct real times $op.\mathsf{inv}$ and $op.\mathsf{res}$, respectively, with $op.\mathsf{inv} < op.\mathsf{res}$ and $[op.\mathsf{inv}, op.\mathsf{res}]$ denoted as $op.\mathsf{interval}$. A *history* H is a set of operations and an induced partial order $\prec_H$ defined as $op_1 \prec_H op_2 \iff op_1.\mathsf{res} < op_2.\mathsf{inv}$. If $\prec_H$ is a total order, H is *sequential*. For an object obj, the set $H|obj$ includes only those operations in H that are invoked on obj, and for a client c, the set $H|c$ includes only those operations in H that are invoked by c. By convention, we assume that $H|c$ is sequential for each client c. (In practice, each "client" is a client *thread*.) A *serialization* S of H is the set H totally ordered by a relation $\prec_S$.



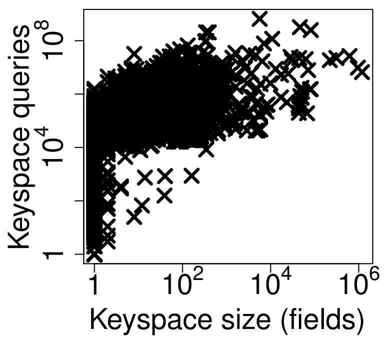
(a) CDF of queries per keyspace. A few keyspaces comprise a large portion of requests.



(b) CDF of keyspace size. Note the x-axis is log-scale. Most keyspaces are small, but some are quite large.



(c) Dominant vs. non-dominant proxy queries (one $\times$ per keyspace). These query types are strongly correlated.



(d) Keyspace size versus queries to that keyspace (one $\times$ per keyspace). The larger keyspaces tend to be accessed more frequently.

Figure 7: Keyspace query and size distributions

Definition 1 (Sequential consistency [3, 26])**.** *A history H is sequentially consistent if there exists a serialization S of H such that the following properties hold: (i) Legality: For each object obj, $S|obj$ is legal (i.e., is in the sequential specification of obj). (ii) Local-Order: If op_1 and op_2 are executed by the same client and $op_1 \prec_H op_2$, then $op_1 \prec_S op_2$.*

The consistency implemented in WACCO, called *cluster consistency*, implies sequential consistency. As such, there is a well-defined order in which updates are applied to each object, and each update operation produces a new version of the object on which it operates. The version number of the new object instance is one greater than that of the object instance to which the update was applied. Let $u\text{-}op.\mathsf{ver}$ be the version number of the object instance produced by $u\text{-}op$.

Definition 2 (Read cluster)**.** *A read cluster C is a nonempty set of read operations (i) that return the same object version, and (ii) for which $\bigcup_{op \in C} op.\mathsf{interval}$ is a contiguous interval of time. For a read cluster C, we define $C.\mathsf{inv} = \min_{op \in C} op.\mathsf{inv}$ and $C.\mathsf{res} = \min_{op \in C} op.\mathsf{res}$. Let $C.\mathsf{ver}$ be the version of the object when it was read by C.*

We also represent each $u\text{-}op$ as its own *update cluster* $C = \{u\text{-}op\}$, with $C.\mathsf{inv} = u\text{-}op.\mathsf{inv}$, $C.\mathsf{res} = u\text{-}op.\mathsf{res}$, and $C.\mathsf{ver} = u\text{-}op.\mathsf{ver}$. We then use $C_1 \prec_H C_2$ (where C_1 and C_2 are read or update clusters) to mean $C_1.\mathsf{res} < C_2.\mathsf{inv}$.

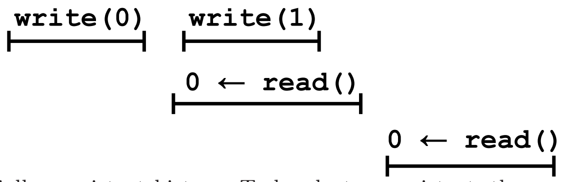
Definition 3 (Cluster consistency)**.** *A set of operations is cluster-consistent if it is sequentially consistent and satisfies Cluster-Order: There exists a partition of the operations into clusters so that if C_1, C_2 are performed on the same object and $C_1.\mathsf{res} < C_2.\mathsf{inv}$, then $C_1.\mathsf{ver} \le C_2.\mathsf{ver}$.*

Fig. 8(a) shows an execution that is sequentially consistent but not cluster-consistent, and so cluster consistency is strictly stronger. However, cluster consistency is weaker than linearizability [21], which requires that for any op_1 and op_2, if $op_1.\mathsf{res} < op_2.\mathsf{inv}$ then $op_1.\mathsf{ver} \le op_2.\mathsf{ver}$; i.e., history precedence must hold at the operation level, not only the cluster level. Fig. 8(b) shows a cluster-consistent execution may not be linearizable.

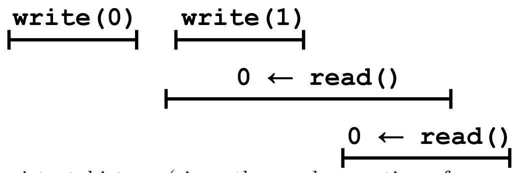
.2.2 Proof of Cluster Consistency

We now prove that the protocol described in Sec. 4.1 implements cluster consistency, ignoring caching (Sec. 4.2), migration (Sec. 4.3), and proxy backups (Sec. 4.4), as these do not alter the semantics of the protocol. Given a history H, consider a directed graph $\mathcal{G}_H$ with operations in H as vertices and three types of edges:

- **Client order** ($\xrightarrow{\text{c}}$): if op_1 and op_2 are performed by the same client and if $op_1 \prec_H op_2$, then $op_1 \xrightarrow{\text{c}} op_2$.
- **Reads-from order** ($\xrightarrow{\text{rf}}$): if $u\text{-}op$ results in an object state on which op is applied, then $u\text{-}op \xrightarrow{\text{rf}} op$.
- **Version order** ($\xrightarrow{\text{v}}$): Let $u\text{-}op_1$ and $u\text{-}op_2$ denote distinct update operations on the same object, and let op denote any other operation on that object



(a) A sequentially consistent history. To be cluster-consistent, the second read must return 1 since its read cluster (itself only) occurs after the write of 1.



(b) A cluster-consistent history (since the read operations form a cluster). To be linearizable, the second read must return 1 since it occurs after the write of 1.

Figure 8: Execution histories for a single object. Time increases left-to-right. Each row denotes one client.

such that $u\text{-}op_1 \xrightarrow{\text{rf}} op$. If $u\text{-}op_1.\mathsf{ver} > u\text{-}op_2.\mathsf{ver}$, then $u\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_1$ and otherwise $op \xrightarrow{\text{v}} u\text{-}op_2$.

We use natural shorthands such as $\xrightarrow{\text{c,rf}} \;=\; \xrightarrow{\text{c}} \cup \xrightarrow{\text{rf}}$. We also use $\xrightarrow{\text{c}}_+$ to denote the irreflexive transitive closure of $\xrightarrow{\text{c}}$, and similarly for other orders.

$\xrightarrow{\text{c}}$ and $\xrightarrow{\text{rf}}$ naturally capture the temporal and data-flow relationships relevant when serializing H, whereas $\xrightarrow{\text{v}}$ constrains any serialization to respect the object versions observed by operations. To prove the sequential consistency of H, we first show that $\mathcal{G}_H$ is acyclic (Lemma 6) and then that this implies that there is a serialization of H respecting *Legality* and *Local-Order* (Corollary 1). We then argue in Lemma 7 that there must be such a serialization that also satisfies *Cluster-Order*.

Below we prove several lemmas which we then use to prove that $\mathcal{G}_H$ is acyclic. Our proofs below involve the following additional notation. To each operation op is associated a logical (Lamport) time [25] $op.\mathsf{linv}$ at which the client invoked it and another logical time $op.\mathsf{lres}$ at which it returned its result at that client. In addition, each $u\text{-}op$ has a *(logical) effective time* of $u\text{-}op.\mathsf{leff}$, which is the Lamport clock value assigned to the event applying $u\text{-}op$ to the object at the proxy hosting the object. For reads, $r\text{-}op.\mathsf{leff}$ is the logical time at which a response $r\text{-}op$ was issued, either by the last proxy to pause $r\text{-}op$ or (if $r\text{-}op$ reached it) by the proxy hosting the object. Note that $op.\mathsf{linv} < op.\mathsf{leff} < op.\mathsf{lres}$ for all operations.

Lemma 1. *The subgraph of $\mathcal{G}_H$ consisting of only edges in $\xrightarrow{\text{c,rf}}$ is acyclic.*

Proof. Since $op_1 \xrightarrow{\text{c}} op_2$ implies $op_1.\mathsf{lres} < op_2.\mathsf{linv}$, it also implies $op_1.\mathsf{leff} < op_2.\mathsf{leff}$. Similarly, it must be that $op_1 \xrightarrow{\text{rf}} op_2$ implies $op_1.\mathsf{leff} < op_2.\mathsf{leff}$, since an update must have been written before it can be read from. Therefore, each edge in $\xrightarrow{\text{c,rf}}$ represents an increase in $op.\mathsf{leff}$, meaning $op_1 \xrightarrow{\text{c,rf}}_+ op_2$ implies $op_1.\mathsf{leff} < op_2.\mathsf{leff}$.

If there is a cycle consisting only of edges in $\xrightarrow{\text{c,rf}}$, then we have $op \xrightarrow{\text{c,rf}}_+ op$, implying $op.\mathsf{leff} < op.\mathsf{leff}$, a contradiction. □

Lemma 2. *If there is a cycle in $\mathcal{G}_H$, then there is a cycle in $\mathcal{G}_H$ in which every $\xrightarrow{v}$ edge appears in an edge sequence of the form $u\text{-}op_1 \xrightarrow{c,rf}_+ r\text{-}op_2 \xrightarrow{v} u\text{-}op_3$.*

Proof. We prove the result by first showing that for any cycle in $\mathcal{G}_H$, any $\xrightarrow{\text{v}}$ edge not already in an edge sequence of the form $op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ can be replaced by edges not in $\xrightarrow{\text{v}}$ to produce a new cycle in $\mathcal{G}_H$. Since $\xrightarrow{\text{v}}$ edges must point to an update, we must consider $\xrightarrow{\text{v}}$ edges of only the forms $r\text{-}op \xrightarrow{\text{v}} u\text{-}op$ and $u\text{-}op' \xrightarrow{\text{v}} u\text{-}op$. In the first case, since $op \xrightarrow{\text{v}} r\text{-}op$ is impossible (again, $\xrightarrow{\text{v}}$ edges point to updates), an edge $r\text{-}op \xrightarrow{\text{v}} u\text{-}op$ already occurs within an edge sequence of the form $op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ on the cycle. In the second case, because updates on each object are applied sequentially, $u\text{-}op'$ is applied before $u\text{-}op$, and so there is a chain of updates to the object such that $u\text{-}op' \xrightarrow{\text{rf}}_+ u\text{-}op$. Replacing the edge $u\text{-}op' \xrightarrow{\text{v}} u\text{-}op$ with this chain produces a cycle not containing $u\text{-}op' \xrightarrow{\text{v}} u\text{-}op$.

To complete the proof, we now must argue that for any edge sequence of the form $op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ on the cycle, there is a corresponding edge sequence $u\text{-}op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ on the cycle. If op_1 is an update, then setting $u\text{-}op_1 = op_1$ completes the argument. Otherwise, consider walking the cycle backward along $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{c}}$ edges from op_1, terminating at a $\xrightarrow{\text{v}}$ edge. Since this $\xrightarrow{\text{v}}$ edge must point to an update, this update suffices for $u\text{-}op_1$. □

If there is a cycle in $\mathcal{G}_H$, then Lemma 2 guarantees the existence of a cycle in which all $\xrightarrow{\text{v}}$ edges occur within edge sequences of a certain form. Below we refer to such a cycle as *constrained*.

Lemma 3. *If there is a cycle in $\mathcal{G}_H$, then within a constrained cycle, at least one edge sequence $u\text{-}op_1 \xrightarrow{c,rf}_+ r\text{-}op_2 \xrightarrow{v} u\text{-}op_3$ has $u\text{-}op_3.\mathsf{leff} \le u\text{-}op_1.\mathsf{leff}$.*

Proof. Consider an alternative graph $\mathcal{G}'_H$ that includes all of the edges of $\mathcal{G}_H$ and additionally the edge $u\text{-}op_1 \xrightarrow{\text{s}} u\text{-}op_3$ whenever $u\text{-}op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$. From any constrained cycle in $\mathcal{G}_H$ we can construct a cycle $op \xrightarrow{\text{c,rf,s}}_+ op$ in $\mathcal{G}'_H$ by replacing edge sequences $u\text{-}op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ on the constrained cycle with the edge $u\text{-}op_1 \xrightarrow{\text{s}} u\text{-}op_3$. Recall from the proof of Lemma 1 that $op' \xrightarrow{\text{c,rf}}_+ op$ implies $op'.\mathsf{leff} < op.\mathsf{leff}$. Moreover, if Lemma 3 were false, then $u\text{-}op_1.\mathsf{leff} < u\text{-}op_3.\mathsf{leff}$ for every edge $u\text{-}op_1 \xrightarrow{\text{s}} u\text{-}op_3$ used in the cycle in $\mathcal{G}'_H$. So, from the cycle $op \xrightarrow{\text{c,rf,s}}_+ op$ we could infer $op.\mathsf{leff} < op.\mathsf{leff}$, a contradiction. □

Each read cluster C has exactly one read operation that reads from the authoritative object itself. We call this "representative" read operation $C.\mathsf{rep}$.

Lemma 4. *If there is an edge sequence $u\text{-}op' \xrightarrow{c,rf}_+ r\text{-}op \xrightarrow{v} u\text{-}op$ in $\mathcal{G}_H$ where $u\text{-}op.\mathsf{leff} \leq u\text{-}op'.\mathsf{leff}$, then $r\text{-}op'.\mathsf{leff} \leq u\text{-}op'.\mathsf{leff}$ where C is the read cluster containing $r\text{-}op$ and $r\text{-}op' = C.\mathsf{rep}$.*

Proof. If $u\text{-}op'.\mathsf{leff} < r\text{-}op'.\mathsf{leff}$, we have $u\text{-}op.\mathsf{leff} \leq u\text{-}op'.\mathsf{leff} < r\text{-}op'.\mathsf{leff}$. Since $r\text{-}op'$ read from the object itself and $u\text{-}op.\mathsf{leff} < r\text{-}op'.\mathsf{leff}$, $r\text{-}op'$ (and thus $r\text{-}op$) must have read from $u\text{-}op$ (or a later update), giving $u\text{-}op \xrightarrow{rf}_+ r\text{-}op$, contradicting $r\text{-}op \xrightarrow{v} u\text{-}op$. □

Lemmas 1–4 show that for $\mathcal{G}_H$ to have a cycle, a necessary condition is an edge sequence of the form $u\text{-}op' \xrightarrow{c,rf}_+ r\text{-}op \xrightarrow{v} u\text{-}op$ where $r\text{-}op$ is contained in a read cluster C whose representative $r\text{-}op' = C.\mathsf{rep}$ is too outdated, i.e., $r\text{-}op'.\mathsf{leff} \leq u\text{-}op'.\mathsf{leff}$. It is for this reason that WACCO is designed to prevent this possibility. Specifically, each returning response to a read operation $r\text{-}op$ carries with it the effective time of representative $r\text{-}op'$ of the cluster containing $r\text{-}op$ and the effective time of the update from which $r\text{-}op'$ and thus $r\text{-}op$ are reading, called $r\text{-}op.\mathsf{lueff}$. That is, if $u\text{-}op \xrightarrow{rf} r\text{-}op$, then $r\text{-}op.\mathsf{lueff} = u\text{-}op.\mathsf{leff}$. Responses to updates can also carry the effective time back to the requester, so that $u\text{-}op.\mathsf{lueff} = u\text{-}op.\mathsf{leff}$.

Each client c tracks the largest $op.\mathsf{lueff}$ for all operations op it has issued, denoted $c.\mathsf{after}$; i.e., $c.\mathsf{after} = \max_{op}\{op.\mathsf{lueff}\}$ where the maximum is taken over all operations issued by c. The outbound request for each $r\text{-}op$ carries with it the current value of $c.\mathsf{after}$, called $r\text{-}op.\mathsf{after}$. When a read response arrives at a proxy, it carries with it the effective time of the read operation $r\text{-}op'$ that reached the authoritative object to elicit that response. The proxy will use this read response to answer a paused read operation $r\text{-}op$ only if $r\text{-}op'.\mathsf{leff} > r\text{-}op.\mathsf{after}$; in this case, $r\text{-}op$ is added to the cluster for which $r\text{-}op'$ serves as the representative and so $r\text{-}op.\mathsf{lueff}$ is set to $r\text{-}op'.\mathsf{lueff}$. Any reads $r\text{-}op''$ that were not answered by $r\text{-}op'$ (i.e., because $r\text{-}op'.\mathsf{leff} \leq r\text{-}op''.\mathsf{after}$) must still be addressed, and now no response is expected inbound. Therefore, the proxy chooses any remaining $r\text{-}op''$ to forward along to elicit another response.

Lemma 5. *There is no edge sequence $u\text{-}op' \xrightarrow{c,rf}_+ r\text{-}op \xrightarrow{v} u\text{-}op$ in $\mathcal{G}_H$ such that $u\text{-}op.\mathsf{leff} \leq u\text{-}op'.\mathsf{leff}$.*

Proof. By Lemma 4, the existence of edge sequence $u\text{-}op' \xrightarrow{c,rf}_+ r\text{-}op \xrightarrow{v} u\text{-}op$ in $\mathcal{G}_H$ such that $u\text{-}op.\mathsf{leff} \leq u\text{-}op'.\mathsf{leff}$ implies that $r\text{-}op'.\mathsf{leff} \leq u\text{-}op'.\mathsf{leff}$ where C is the read cluster containing $r\text{-}op$ and $r\text{-}op' = C.\mathsf{rep}$. By construction, $r\text{-}op$ can be answered by a read response only if the effective time of the read operation $r\text{-}op'$ that reached the authoritative object to elicit the response satisfies $r\text{-}op'.\mathsf{leff} > r\text{-}op.\mathsf{after}$. So, to prove the lemma, it suffices to show that $u\text{-}op'.\mathsf{leff} \leq r\text{-}op.\mathsf{after}$.

Given the edge sequence $u\text{-}op' \xrightarrow{c,rf}_+ r\text{-}op \xrightarrow{v} u\text{-}op$, let $u\text{-}op''$ be the update operation on this sequence that precedes and is closest to $r\text{-}op$; i.e., there is no update operation between $u\text{-}op''$ and $r\text{-}op$ along this edge sequence. Let c be the client that issued $r\text{-}op$. Either $u\text{-}op'' = u\text{-}op'$ and so $u\text{-}op'.\mathsf{leff} = u\text{-}op''.\mathsf{leff}$, or $u\text{-}op' \xrightarrow{c,rf}_+ u\text{-}op''$ and so $u\text{-}op'.\mathsf{leff} < u\text{-}op''.\mathsf{leff}$. It thus suffices to prove that $u\text{-}op''.\mathsf{leff} \leq r\text{-}op.\mathsf{after}$. If the chain $u\text{-}op'' \xrightarrow{\text{c,rf}}_{+} r\text{-}op$ includes no $\xrightarrow{\text{rf}}$ edges, then c also issued $u\text{-}op''$, and so $u\text{-}op''.\mathsf{leff} = u\text{-}op''.\mathsf{lueff} \leq r\text{-}op.\mathsf{after}$ because $r\text{-}op.\mathsf{after}$ is defined as the maximum $op.\mathsf{lueff}$ for all operations op that c has issued so far (including $u\text{-}op''$ itself). If the chain $u\text{-}op'' \xrightarrow{\text{c,rf}}_{+} r\text{-}op$ includes one $\xrightarrow{\text{rf}}$ edge, it must be the first edge, giving $u\text{-}op'' \xrightarrow{\text{rf}} r\text{-}op'' \xrightarrow{\text{c}}_{+} r\text{-}op \xrightarrow{\text{v}} u\text{-}op$. Then, c also issued $r\text{-}op''$, and so $u\text{-}op''.\mathsf{leff} = r\text{-}op''.\mathsf{lueff} \leq r\text{-}op.\mathsf{after}$, again due to the construction of $r\text{-}op.\mathsf{after}$. □

Lemma 6. *$\mathcal{G}_H$ is acyclic.*

Proof. If there is a cycle in $\mathcal{G}_H$, a sequence $u\text{-}op_1 \xrightarrow{\text{c,rf}}_{+} r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ such that $u\text{-}op_3.\mathsf{leff} \leq u\text{-}op_1.\mathsf{leff}$ must appear in the cycle (by Lemma 3), which is not possible (by Lemma 5), giving a contradiction. □

Corollary 1. *The protocol of Sec. 4.1 is sequentially consistent.*

Proof. Consider any topological sort of $\mathcal{G}_H$. Due to the $\xrightarrow{\text{c}}$ edges, it satisfies *Local-Order*. Moreover, every read and update operation appears in this serialization after the update producing the object state to which it is applied (due to $\xrightarrow{\text{rf}}$ edges) and before any subsequent update (due to $\xrightarrow{\text{v}}$ edges). Consequently, *Legality* is satisfied. □

Lemma 7. *The protocol of Sec. 4.1 satisfies Cluster-Order.*

Proof. Consider two clusters $C_1, C_2 \subseteq H|obj$ as defined above, such that $C_1 \prec_H C_2$. Therefore, $C_1.\mathsf{rep}$ was applied to the authoritative object before $C_2.\mathsf{rep}$ (in real time), and so $C_1.\mathsf{ver} \leq C_2.\mathsf{ver}$. □