Depot: Cloud Storage with Minimal Trust

PRINCE MAHAJAN, SRINATH SETTY, SANGMIN LEE, ALLEN CLEMENT,
LORENZO ALVISI, MIKE DAHLIN, and MICHAEL WALFISH, The University of Texas at Austin

This article describes the design, implementation, and evaluation of Depot, a cloud storage system that minimizes trust assumptions. Depot tolerates buggy or malicious behavior by *any number* of clients or servers, yet it provides safety and liveness guarantees to correct clients. Depot provides these guarantees using a two-layer architecture. First, Depot ensures that the updates observed by correct nodes are consistently ordered under Fork-Join-Causal consistency (FJC). FJC is a slight weakening of causal consistency that can be both safe and live despite faulty nodes. Second, Depot implements protocols that use this consistent ordering of updates to provide other desirable consistency, staleness, durability, and recovery properties. Our evaluation suggests that the costs of these guarantees are modest and that Depot can tolerate faults and maintain good availability, latency, overhead, and staleness even when significant faults occur.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/server; distributed systems*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms: Design, Algorithms, Reliability, Experimentation, Security

Additional Key Words and Phrases: Cloud storage, Byzantine fault tolerance, Fork-Join-Causal (FJC) consistency, fork consistency

ACM Reference Format:
Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. 2011. Depot: Cloud storage with minimal trust. ACM Trans. Comput. Syst. 29, 4, Article 12 (December 2011), 38 pages.
DOI = 10.1145/2063509.2063512 http://doi.acm.org/10.1145/2063509.2063512

1. INTRODUCTION

This article describes the design, implementation, and evaluation of Depot, a cloud storage system in the spirit of Amazon S3[1], Microsoft Azure[2], and Google Storage[3] but with a crucial difference: Depot clients do not have to *trust*, that is *assume*, that Depot servers operate correctly.

[1]http://aws.amazon.com/s3
[2]http://www.microsoft.com/windowsazure/windowsazure
[3]http://code.google.com/apis/storage/docs/overview.html

A. Clement is now at Max Planck Institute for Software Systems.
This article revises and extends a prior publication of the same name at the Symposium on Operating Systems Design and Implementation (OSDI) 2010.
This work was supported by ONR grant N00014-09-10757, AFOSR grant FA-9550-10-1-0073, NSF grant CNS-0720649, NSF grant CCF-1048269, and NSF grant CNS-1055057.
Authors' address: P. Mahajan (corresponding author), S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, Department of Computer Science, The University of Texas at Austin, 1616 Guadalupe, Suite 2.408, Austin, TX; email: fuss@cs.utexas.edu.

DOI 10.1145/2063509.2063512 http://doi.acm.org/10.1145/2063509.2063512

What motivates Depot is that cloud Storage Service Providers (SSPs), such as S3 and Azure, are fault-prone black boxes operated by a party other than the data owner. Indeed, clouds can experience correlated software bugs [Beckmann 2009; CNet News 2011], correlated manufacturing defects [Pinheiro et al. 2007], misconfigured servers and operator error [Amazon 2011; Oppenheimer et al. 2003], malicious insiders [US 2005], bankruptcy [News 2002] or change of business strategy [Yahoo 2011], undiagnosed problems [Calore 2009], Acts of God (e.g., fires [Cook 2009]) and Man [Miller 2009]. Thus, it seems prudent for clients to avoid strong assumptions about an SSP's design, implementation, operation, and status—and instead to rely on end-to-end checks of well-defined properties. In fact, removing such assumptions promises to help SSPs too: today, a significant barrier to adopting cloud services is precisely that many organizations hesitate to place trust in the cloud [CircleID 2009].

Given this motivation, Depot assumes less than any prior system about the correctness of participating hosts.

— *Depot eliminates trust for safety.* A client can ensure safety by assuming the correctness of only itself. Depot guarantees that any subset of correct clients observes sensible, well-defined semantics. This holds regardless of how many nodes fail and no matter whether they are clients or servers, whether these are failures of omission or commission, and whether these failures are accidental or malicious.
— *Depot minimizes trust for liveness and availability.* We wish we could say "trust only yourself" for liveness and availability. Depot does eliminate trust for updates: a client can always update any object for which it is authorized, and any subset of connected, correct clients can always share updates. However, for reads, there is a fundamental limit to what any storage system can guarantee: if no correct, reachable node has an object, that object may be unavailable. We cope with this fundamental limit by allowing reads to be served by any node (even other clients) while preserving the system's guarantees, and by configuring the replication policy to use several servers (which protects against failures of clients and subsets of servers) and at least one client (which protects against cloud failures, temporary [Amazon S3 Team 2008] and permanent [Calore 2009; News 2002]).

Though prior work has reduced trust assumptions in storage systems, it has not minimized trust with respect to safety, liveness, or both. For example, quorum and replicated state machine approaches [Castro and Liskov 2002; Clement et al. 2009; Guerraoui et al. 2010] tolerate failures by a fraction of servers. However, they sacrifice safety when faults exceed a threshold and liveness when too few servers are reachable. Fork-based systems [Cachin et al. 2007, 2009; Li and Mazières 2007; Li et al. 2004] remain safe without trusting a server, but they compromise liveness in two ways. First, if the server is unreachable, clients must block. Second, a faulty server can permanently partition correct clients, preventing them from ever observing each other's subsequent updates.

Indeed, it is challenging to guarantee safety and liveness while minimizing trust assumptions: without some assumptions about correct operation, providing even a weak guarantee like eventual consistency—the bare minimum of what a storage service should provide—seems difficult. For example, a faulty storage node receiving an update from a correct client might quietly fail to propagate that update, thereby hiding it from the rest of the system. Perhaps surprisingly, we find that eventual consistency *is* possible in this environment.

In fact, Depot meets a contract far stronger than eventual consistency even under assorted and abundant faults and failures. This set of well-defined guarantees under

weak assumptions is Depot's top-level contribution, and it derives from a novel synthesis of prior mechanisms and our own. Depot is built around three key ideas.

(1) *Reduce misbehavior to concurrency.* As in prior work [Cachin et al. 2007, 2009; Li and Mazières 2007; Li et al. 2004], the protocol requires that an update be signed and that it name both its antecedents and the system state seen by the updater. Then, misbehavior by clients or servers is limited to *forking*: showing divergent histories to different nodes. However, previous work detects but does not repair forks. In contrast, Depot allows correct clients to *join forks*, that is, to incorporate the divergence into a sensible history, which allows them to keep operating in the face of faults. Specifically, a correct node regards a fork as logically concurrent updates by two *virtual nodes*. At that point, correct nodes can handle forking by faulty nodes using the same techniques [Birrell et al. 1982; Demmer et al. 2008; Kistler and Satyanarayanan 1992; Reiher et al. 1994; Terry et al. 1995] that they need anyway to handle a better understood problem: logically concurrent updates during disconnected operation.
(2) *Enforce Fork-Join-Causal consistency.* To allow end-to-end checks on SSP behavior, we must specify a contract: When must an update be visible to a read? When is it okay for a read to "miss" a recent update? Depot guarantees that a correct client observes *Fork-Join-Causal consistency* (FJC) no matter how many other nodes are faulty. FJC is a slight weakening of causal consistency [Ahamad et al. 1995; Lamport 1978; Petersen et al. 1997]. Depot defines FJC as its consistency contract because it is weak enough to enforce despite faulty nodes and without hurting availability. At the same time, FJC is strong enough to be useful: nodes see each other's updates in an order that reflects dependencies among both correct and faulty nodes' writes. This ordering is useful not only for end users of Depot but also internally, within Depot.
(3) *Layer other storage properties over FJC.* Depot has a layered architecture. It builds on the ordering guarantees provided by FJC to provide other desirable properties: eventual consistency, bounded staleness, durability, high availability, integrity (ensuring that only authorized nodes can update an object), snapshotting of versions (to guard against spurious updates from faulty clients), garbage collection, and eviction of faulty nodes.[4] For all of these properties, the challenge is to precisely define the strongest guarantee that Depot can provide with minimal assumptions about correct operation. Once each property is defined, implementation is straightforward because we can build on FJC, which lets us reason about the order in which updates propagate through the system.

The price of providing these guarantees is tolerable, as demonstrated by an experimental evaluation of a prototype implementation of Depot. Depot adds a few hundred bytes of metadata to each update and each stored object, and it requires a client to sign and store each of its updates. We demonstrate that Depot can tolerate faults and maintain good availability, latency, overhead, and staleness even when significant faults occur. Additionally, because Depot makes minimal assumptions about servers, we can implement *Teapot*, a variation of Depot that provides many of Depot's guarantees using an unmodified SSP, such as Amazon's S3. The difference between Depot and Teapot suggests several modest extensions to SSPs' interfaces that would strengthen their guarantees.

[4]We are not explicitly addressing confidentiality and privacy, but, as discussed in Section 3.1, existing approaches can be layered on Depot.

2. WHY UNTRUSTED STORAGE?

When we say that a component is untrusted, we are not adopting a "tinfoil hat" stance that the component is operated by a malicious actor, nor are we challenging the honesty of Storage Service Providers (SSPs). What we mean is that the system provides guarantees, usually achieved by end-to-end checks, even if the given component is incorrect. Since components could be incorrect for many reasons (as stated in the Introduction), we believe that designing to tolerate incorrectness is prudence, not paranoia. We now answer some natural questions.

SSPs are operated by large, reputable companies, so why not trust them? That is like asking, "Banks are large, reputable repositories of money, so why do we need bank statements?" For many reasons, customers as well as banks want customers to be able to check the bank's view of their account activity. Likewise, our approach might appeal not only to customers but also to SSPs: by requiring less trust, a service might attract more business.

How likely are faults in the SSP? We do not know the precise probability. However, we know that providers do fail (as mentioned in the Introduction). More broadly, SSPs are subject to nonnegligible risks. First, they are opaque (by nature). Second, they are complex distributed systems. Indeed, it is difficult to cope with known hardware failure modes in file systems that are local [Prabhakaran et al. 2005]; in the case of cloud storage, this difficulty can only grow. Given the opacity and complexity, it seems prudent not to assume the unfailing correctness of an SSP's internals.

Even if we do not assume that SSPs are perfect, the most likely failure is the occasional corrupted or lost block, which can be addressed with checksums and replication. Do you really need mechanisms to handle other cases (that all of the nodes are faulty, that a fork happens, that old or out-of-order data is returned, etc.)? Replication and checksums are helpful, and they are part of Depot. However, they are not sufficient. First, failures are often correlated: as Vogels notes, uncorrelated failures are "absolutely unrealistic . . . as [failures] are often triggered by external or environmental events" [Vogels 2006]. These events include the litany in the Introduction. Second, other types of failures are possible. For example, a machine that loses power after failing to commit its output [Nightingale et al. 2008; Yang et al. 2006] may lose recent updates, leading to forks in history. Or, a network failure might delay propagation of an update from one SSP node to another, causing some clients to read stale data. In general, our position is that rather than try to handle every possible failure individually, it is preferable to define an end-to-end contract and then design a system that always meets that contract.

The preceding events seem unlikely. Is tolerating them worth the cost? One of our purposes in this article is to report for the first time what that cost is. Whether to pay for the guarantees is up to the application, but as the price is modest, we anticipate, with hope, that many applications will find it attractive.

What about clients? We also minimize trust of clients (since they are, of course, also vulnerable to faults).

3. ARCHITECTURE, SCOPE, AND USE

Figure 1 depicts Depot's high-level architecture. A set of clients stores key-value pairs on a set of servers. Keys and values are uninterpreted bytes. The servers are operated by one or more Storage Service Providers (SSPs) that are distinct from the data owner that operates the clients. Depot runs as a library on both clients and servers, and a Depot client exposes an interface of GET and PUT to its application users.

Depot's target scenarios—and the cases where it is engineered to incur modest cost—are when the values are at least a few KB, and each key-value pair is shared

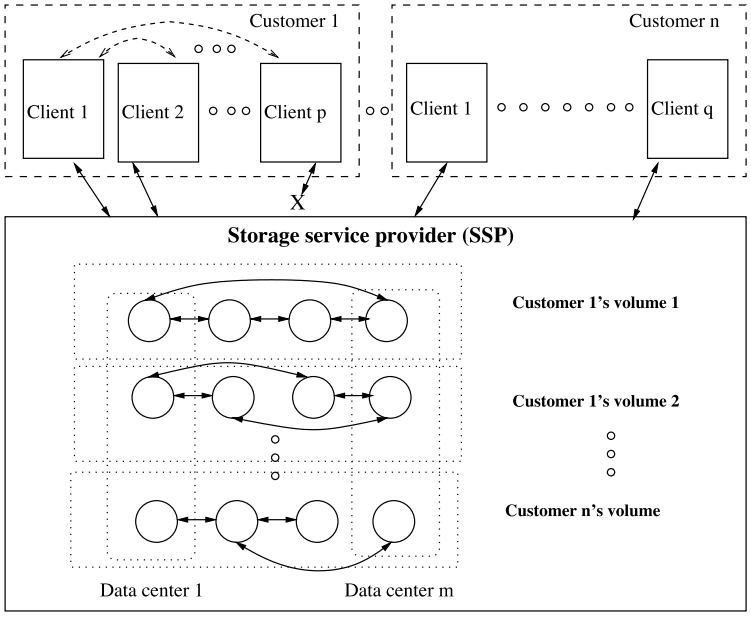

Fig. 1. Architecture of Depot. The arrows between servers represent replication and exchange. Clients can exchange data directly (as represented by the dotted arrows between clients) and continue functioning in the event of a catastrophic SSP failure. Each volume is replicated over a set of servers that can be connected using any topology.

by at most a few hundreds of clients. For example, the Depot clients could be running on the Web servers of a small- or medium-scale Web service that uses an SSP for back-end storage.

For scalability, we slice the system into groups of servers, with each group responsible for one or more *volumes*. Each volume corresponds to a range of one customer's keys, and a server independently runs the protocol for each volume assigned to it. Many strategies for partitioning keys are possible [DeCandia et al. 2007; Karger et al. 1997; Nath et al. 2006], and we leave the assignment of keys to volumes to the layers above Depot. The servers for each volume may be geographically distributed, and servers may replicate updates using any topology (chain, mesh, star, etc.). Depot clients can access data through any server; for maximal availability, Depot, like Dynamo [DeCandia et al. 2007], does not require overlapping read and write quorums.

Depot supports many strategies for coping with the failure of an SSP. In a *single-SSP* deployment, clients are configured such that each client stores a copy of the data that it authors. Then, if the SSP fails, clients can ensure availability by exchanging metadata with each other directly (as indicated by the dotted arrows in Figure 1) and by using the data stored at the authoring clients. If the SSP later recovers, clients can continue using the SSP (after sending the missed updates to the SSP servers). This raises a question: why have the SSP at all? We point to the usual benefits of cloud services: cost, scalability, geographic replication, management, and performance.

Indeed, SSPs are useful enough that a customer might use a *multiple-SSP* deployment, with servers for the same volume hosted by different SSPs. This configuration drastically reduces the risk of correlated failures across replicas [Abu-Libdeh et al. 2010; Kotla et al. 2007]. Thus, clients in this configuration need not exchange with each other or stored authored data locally. For simplicity, the rest of this article describes and evaluates the single-SSP configuration.

Most mechanisms in Depot treat clients and servers indistinguishably; in describing these mechanisms, we use the term *node* to refer to both clients and servers. Certain subprotocols, however, treat clients and servers differently; in those cases we explicitly use the terms *client* and *server*.

3.1. Issues Addressed

One of our aims in this work is to push the envelope in the trade-offs between trust assumptions and system guarantees. Specifically, for a set of standard properties that one might desire in a storage system, we ask: what is the minimum assumption that we need to provide useful guarantees, and what are those guarantees? The issues that we examine are as follows.

— *Consistency* (Section 5–Section 6.2) and *bounded staleness* (Section 6.4). Once a write occurs, the update should be visible to reads "soon". Consistency limits the extent to which the storage system can reorder or omit updates; bounded staleness limits the extent to which updates can be delayed undetectably.
— *Availability and durability* (Section 6.3). Our availability goal is to maximize the fraction of time that a client succeeds in reading or writing an object. Durability means that the system does not lose data permanently.
— *Integrity and authorization* (Section 6.5). Only clients authorized to update an object should be able to create valid updates that affect reads on that object.
— *Data recovery* (Section 6.6). Data owners care about end-to-end reliability. Consistency, durability, and integrity are not enough when the layers above Depot—faulty clients, applications, or users—can issue authorized writes that replace good data with bad. Depot does not try to distinguish good updates from bad ones, nor does it innovate on the abstractions used to defend data from higher-layer failures. We do, however, explore how Depot can support standard techniques such as *snapshots* to recover earlier versions of data.
— *Evicting faulty nodes* (Section 6.7). If a faulty node provably deviates from the protocol, we wish to evict it from the system so that it will not continue to disrupt operation. However, we must never evict correct nodes.

Depot provides these properties with a layered approach. Its core protocol (Section 5) addresses consistency. Specifically, the protocol enforces Fork-Join-Causal consistency (FJC), which is the same as causal consistency [Ahamad et al. 1995; Lamport 1978; Petersen et al. 1997] in benign runs. This protocol is the essential building block for the other properties listed (availability and durability, integrity and authorization, etc.). In Section 6, we define these properties precisely and discuss how Depot provides them.

Note that we explicitly do not try to solve the confidentiality/privacy problem within Depot. Instead, like commercial storage systems (S3, Azure, Google Storage), Depot enforces integrity and authorization (via client signatures) but leaves it to higher layers to use appropriate techniques for the privacy requirements of each application (e.g., allow global access, encrypt values, encrypt both keys and values, introduce artificial requests to thwart traffic analysis, etc.).

We also do not claim that the aforesaid list of issues is exhaustive. For example, it may be useful to audit SSPs with black-box tests to verify that they are storing data as promised [Kotla et al. 2007; Shah et al. 2007], but we do not examine that issue. Still, we believe that the properties are sufficient to make the resulting system useful.

3.2. Depot in Use: Applications and Conflicts

Depot's key-value store is a low-level building block over which many applications can be built. For example, hundreds of widely used applications—including backup,

point of sale software, file transfer, investment analytics, cross-company collaboration, and telemedicine—use the S3 key-value store [S3 App Catalog 2011], and Depot can serve all of them: it provides a similar interface to S3, and it provides strictly stronger guarantees.

An issue in systems that are causally consistent and weaker—a set that includes not just Depot and S3 but also CVS, SVN, Git, Bayou [Petersen et al. 1997], Coda [Kistler and Satyanarayanan 1992], and others—is handling concurrent writes to the same object. Such conflicts are unfortunate but unavoidable: they are provably the price of high availability [Gilbert and Lynch 2002].

Many approaches to resolving conflicting updates have been proposed [Kistler and Satyanarayanan 1992; Reiher et al. 1994; Terry et al. 1995], and Depot does not claim to extend the state-of-the-art on this front. In fact, Depot is less ambitious than some past efforts: rather than try to resolve conflicts internally (e.g., by picking a winner, merging concurrent updates, or rolling back and reexecuting transactions [Terry et al. 1995]), Depot simply exposes concurrency when it occurs: a read of key k returns the *set* of updates to k that have not been superseded by any logically later update of k.[5]

This approach is similar to that of S3's replication substrate, Dynamo [DeCandia et al. 2007], and it supports a range of application-level policies. For example, applications using Depot may resolve conflicts by *filtering* (e.g., reads return the update by the highest-numbered node, reads return an application-specific merge of all updates, or reads return all updates) or by *replacing* (e.g., the application reads the multiple concurrent values, performs some computation on them, and then writes a new value that thus appears logically after and thereby supersedes the conflicting writes).

3.3. System and Threat Model

We now briefly state our technical assumptions. First, nodes are subject to standard cryptographic hardness assumptions, and each node has a public key known to all nodes. Second, *any number* of nodes can fail in arbitrary (Byzantine [Lamport et al. 1982]) ways: they can crash, corrupt data, lose data, process some updates but not others, process messages incorrectly, collude, etc. Third, we assume that any pair of timely, connected, and correct nodes can eventually exchange any finite number of messages. That is, a faulty node cannot forever prevent two correct nodes from communicating (but we make no assumptions about how long "eventually" is).

Fourth, we have just used the term *correct node*. This term refers to a node that never deviates from the protocol nor becomes permanently unavailable. A node that obeys the protocol for a time but later deviates is not counted as correct. Conversely, a node that crashes and recovers with committed state intact is equivalent to a correct node that is slow. Fifth, we assume that unresponsive clients are eventually repaired or replaced. To satisfy this assumption, an administrator can install an unresponsive client's keys and configuration on new hardware [Castro and Liskov 2002]. Sixth, we assume that the clocks at clients are synchronized, that is, that there is a bound on worst-case skew.[6]

[5]Note that Depot neither creates concurrency nor makes the problem worse. If an application cannot deal with conflicts, it can still use Depot but must restrict its use (e.g., by adding locks and sending all operations through a single SSP node), and it must sacrifice the ability to tolerate faults (such as forks) that appear as concurrency.

[6]This assumption is required only for the bounded staleness property (Section 6.4); a deployment that does not need the property can avoid making the assumption: the bounded staleness subprotocol is modular and can be separately turned off.

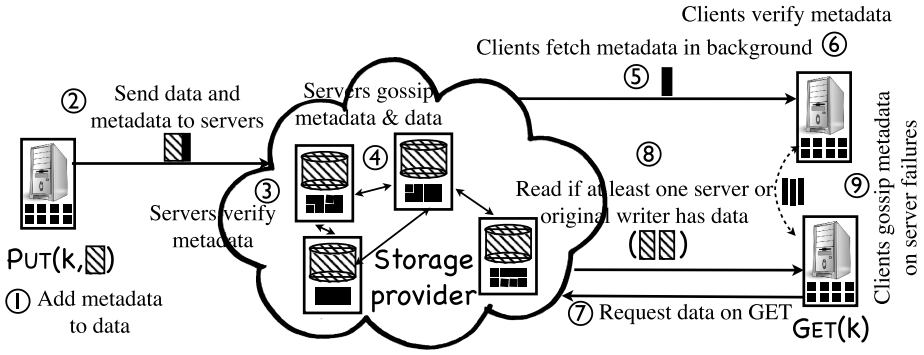

Fig. 2. Overview of Depot's operation in a single-SSP deployment. Striped boxes denote data whereas shaded boxes denote metadata (updates). Depot runs as a library at clients and servers. ① On receiving a PUT request from the application, a client generates a signed update that summarizes the prior updates seen by this client and contains a hash of the data provided in the PUT request. Clients store all updates that they receive or create but store data only for updates that they create. ② The client pushes this update to the servers when connectivity permits. ③ Each server verifies all incoming updates and data. ④ All verified updates are stored and propagated to other servers through background gossip. ⑤ Clients periodically fetch new updates from the server in the background. ⑥ Like servers, clients also verify updates before accepting them. ⑦ On receiving a GET request, a client uses its stored updates to identify the hash of data that must be accessed and attempts to fetch this data from the servers or, if the servers are unavailable, from the client that created this update. ⑧ A GET completes when a client receives data that matches the most recent update(s) stored at a client. ⑨ If a catastrophic cloud failure occurs, clients gossip updates with each other.

4. OVERVIEW

This section gives a high-level operational description of Depot while the sections ahead go bottom-up.

Figure 2 depicts Depot's handling of PUTs and GETs in a single-SSP deployment. On receiving a PUT request (step 1), a client creates an *update* (as described in Section 5) and a collision-resistant hash of the data (the value in the PUT request). Each update completely overwrites the previous value of that object. The update is signed and contains information necessary to enforce the consistency, integrity, and other safety properties that Depot guarantees. While a PUT request can be completed by simply creating the appropriate update and storing it locally, a client tries to push the update (step 2) to its preferred server (typically the nearest one) to ensure that other clients can observe its PUT in a timely manner (Section 7).

Each correct node (client and server) verifies incoming updates to ensure Depot's safety and liveness properties (step 3 and step 6). These update checks are described in Section 5, Section 6.3, Section 6.5, and Section 6.7. All correct nodes store all the updates until they are garbage collected by unanimous consent (Section 6.6).

Servers gossip updates and data with each other to ensure quick convergence (step 4). In the background, clients periodically fetch new updates from their preferred servers (step 5, detailed in Section 7) to minimize the metadata (update) transfer on a GET. Clients, like servers, verify received updates before accepting them (step 6).

On a GET to a key k, the client uses its local update store to identify the hash of the most recent PUT(s) to k. The client then tries to fetch appropriate data from the server (step 7, detailed in Section 7). The server responds by sending the appropriate data if it has that or newer data for the requested key k (step 8, detailed in Section 7). The client fails over to other servers and eventually to client-to-client mode (step 9, detailed in Section 7) if no server has the appropriate data or if all the servers are unavailable. In client-to-client mode, clients continue exchanging updates with each

other until the servers recover, in which case they first notify the servers of any new updates generated since the last failure.

The next three sections detail Depot's design and implementation. Section 5 describes the core replication protocol that all nodes (clients and servers) use to replicate updates. Section 6 describes the subprotocols that are layered over the replication protocol, along with the properties that are provided by Depot. Last, Section 7 puts the pieces together, explaining how the various subprotocols are composed.

5. CORE PROTOCOL

In Depot, clients' reads and updates to shared objects should always appear in an order that reflects the logic of higher layers. For example, an update that removes one's parents from a friend list and an update that posts spring break photos should appear in that order, not the other way around [Cooper et al. 2008]. However, Depot has two challenges. First, it aims for maximum availability, which fundamentally conflicts with the strictest orderings [Gilbert and Lynch 2002]. Second, it aims to provide its ordering guarantees despite arbitrary misbehavior from any subset of nodes. In this section, we describe how the protocol at Depot's core achieves a sensible and robust order of updates while optimizing for availability and tolerating arbitrary misbehavior.

As mentioned earlier, this protocol is run by both clients and servers (and we will describe it in terms of nodes). This symmetry simplifies the design; for example, it allows client-to-client and client-to-server data exchange to look similar.

5.1. Basic Protocol

This subsection describes the basic protocol to propagate updates, ignoring the problems raised by faulty nodes. The protocol is essentially a standard log exchange protocol [Belaramani et al. 2006; Petersen et al. 1997]; we describe it here for background and to define terms.

The core message in Depot is an *update* that changes the *value* associated with a *key*. It has the following form.

$$\text{dVV, \{key, H(value), logicalClock@nodeID, H(history)\}}_{\sigma_{\text{nodeID}}}$$

Updates are associated with logical times. A node assigns each update an *accept stamp* of the form *logicalClock@nodeID* [Petersen et al. 1997]. A node N increments its logical clock on each local write. Also, when N receives an update u from another node, N advances its logical clock to exceed u's. Thus, an update's accept stamp exceeds the accept stamp of any update on which it depends [Lamport 1978]. The remaining fields, *dVV* and *H(history)*, and the writer's signature, σ_{nodeID}, defend against faults and are discussed in Sections 5.2 and 5.3.

Each node maintains two local data structures: a *log* of the updates that it has seen and a *checkpoint* reflecting the current state of the system. For efficiency, Depot separates data from metadata [Belaramani et al. 2006], so the log and checkpoint contain collision-resistant hashes of values. (A node fetches the actual values on demand, which it can do by using the hash as an index [Fu et al. 2002]; this happens in modules layered over the core replication protocol.) Each node sorts the updates in its log by accept stamp, sorting first by *logicalClock* and breaking ties with $nodeID$. Thus, each new write issued by a node appears at the end of its own log and (assuming no faulty nodes) the log reflects a causally consistent [Ahamad et al. 1995; Lamport 1978; Petersen et al. 1997] ordering of all writes.

Information about updates propagates through the system when nodes exchange tails of their logs. Each node N maintains a *version vector* VV with an entry for each node M in the system: $N.VV[M]$ is the highest logical clock N has observed for any

update by M [Parker et al. 1983]. To transmit updates from node M to node N, M sends to N the updates from its log that N has not seen.

Two updates are *logically concurrent* if neither appears in the other's history. Concurrent writes may *conflict* if they update the same object; conflicts are handled as described in Section 3.2.

5.2. Consistency Despite Faults

There are three fields in an update that defend the protocol against faulty nodes. The first is a *history hash*, *H(history)*, that encodes the history on which the update depends using a collision-resistant hash that covers the most recent update by each node known to the writer when it issued the update. By recursion, this hash covers all updates included by the writer's current version vector. Second, each update is sent with a dependency version vector, *dVV*, that indicates the version vector that the history hash covers. (We discuss the scalability of this mechanism in Section 7. Briefly, *dVV* logically represents a full version vector, but when node N creates an update u, u's *dVV* actually contains only the entries that have changed since the last write by N.) Third, a node signs its updates with its private key.

A correct node C accepts an update u only if it meets five conditions. First, u must be properly signed. Second, except as described in the next subsection, u must be newer than any updates from the signing node that C has already received. This check prevents C from accepting updates that modify the history of another node's writes. Third, C's version vector must include u's *dVV*. Fourth, u's *history hash* must match a hash computed by C across every node's last update at time *dVV*. The third and fourth checks ensure that before receiving update u, C has received all of the updates on which u depends. Fifth, u's accept stamp must be at most a constant times C's current wall-clock time (e.g., $u.acceptStamp < 1000 * currentTimeMillis()$). This check defends against exhaustion of the 64-bit logical time space. Note that C can always issue new updates (e.g., at a rate of at least 1 update per microsecond), and C can communicate new updates to other correct nodes after a worst-case delay determined by the difference in their clocks.

Given these checks, attempts by a faulty node to fabricate u and pass it as coming from a correct node, to omit updates on which u depends, or to reorder updates on which u depends will result in C rejecting u. To compromise causal consistency, a faulty node has one remaining option: to *fork*, that is, to show different sequences of updates to different communication partners [Li et al. 2004]. Such behavior certainly damages consistency. However, the damage is limited, as we now illustrate. Then, in Section 5.3 we describe how Depot *recovers* from forks.

Example: The history hash in action. A faulty node M can create two updates $u_{1@M}$ and $u'_{1@M}$ such that neither update's history includes the other's. M can then send $u_{1@M}$ and the updates on which it depends to one node, $N1$, and $u'_{1@M}$ and its preceding updates to another node, $N2$. $N1$ can then issue new updates (like $u_{2@N1}$) that depend on updates from *one* of M's forked updates (here, $u_{1@M}$) and send these new updates to $N2$. At this point, absent the history hash, $N2$ would receive $N1$'s new updates without receiving the updates by M on which they depend: $N2$ already received $u'_{1@M}$, so its version vector appears to already include the prior updates. Then, if $N2$ applies just $N1$'s writes to its log and checkpoint, multiple consistency violations could occur. First, the system may never achieve eventual consistency because $N2$ may never see write $u_{1@M}$. Further, the system may violate causality because $N2$ has updates from $N1$ but not some earlier updates (e.g., $u_{1@M}$) on which they depend.

The preceding confusion is prevented by the history hash. If $N1$ tries to send its new updates to $N2$, $N2$ will be unable to match the new updates' history hashes to the updates that $N2$ actually observed, and $N2$ will reject $N1$'s updates (and vice versa). As a result, $N1$ and $N2$ will be unable to exchange any updates after the *fork junction* introduced by M after $u_{0@M}$.

Discussion. At this point, we have composed mechanisms from Bayou [Petersen et al. 1997] and PRACTI [Belaramani et al. 2006] (update exchange), SUNDR [Li et al. 2004] (signed version vectors), and BFT2F [Li and Mazières 2007] (history hashes, here used by *clients* and modified to apply to history trees instead of linear histories) to provide *Fork-Causal Consistency* (FCC) under arbitrary faults. We define FCC precisely in Appendix A. Informally, it means that each node sees a causally consistent subset of the system's updates even though the whole system may no longer be causally consistent. Thus, although the global history has branched, as each node peers backward from its branch to the beginning of time, it sees causal events the entire way.

Unfortunately, enforcing even this weakening of causal consistency would prohibit eventual consistency, crippling the system: FCC requires that once two nodes have been forked, they can never observe one another's updates after the fork junction [Li et al. 2004]. In many environments, partitioning nodes this way is unacceptable. In those cases, it would be far preferable to further weaken consistency to ensure an availability property: *connected, correct nodes can always share updates*. We now describe how Depot achieves this property, using a new mechanism: *joining forks* in the system's history.

5.3. Protecting Availability: Joining Forks

To *join forks*, nodes use a simple coping strategy: they convert concurrent updates by a single faulty node into concurrent updates by a pair of virtual nodes. A node that receives these updates handles them as it would "normal" concurrency: it applies both sets of updates to its state and, if both branches modify the same object, it returns both conflicting updates on reads (Section 3.2). We now fill in some details.

Version-and-hash vectors. Each node N's locally maintained $N.VV[M]$ contains not only the highest logical clock that N has observed for M but also a hash of M's update at that logical clock. As a result, if a faulty node creates logically different updates with the same accept stamp, other nodes can detect the discrepancy through update exchange.

Identifying a fork. First consider a two-way fork. A fork junction comprises exactly three updates where a faulty node M has created two updates (e.g., $u_{1@M}$ and $u'_{1@M}$) such that: (i) neither update includes the other in its history and (ii) each update's history hash links it to the same previous update by that writer (e.g., $u_{0@M}$). If a node $N2$ receives from a node $N1$ an update whose history is incompatible with the updates it has already received, and if neither node has yet identified the fork junction, $N1$ and $N2$ identify the three forking updates as follows. First, $N1$ and $N2$ perform a search on the updates included in the nodes' version vectors to identify a version vector, VV_c, encompassing a common history (the search proceeds with $N2$ requesting exponentially older updates until it identifies an encompassing VV_c). Then, $N1$ sends its log of updates beginning from VV_c. Finally, at some point, $N2$ receives the first update by M (e.g., $u_{1@M}$) that is incompatible with the updates by M that $N2$ has already received (e.g., $u_{0@M}$ and $u'_{1@M}$).

Tracking forked histories. After a node identifies the three updates in the fork junction, it expands its version vector to include three entries for the node that issued the

forking updates. The first is the prefork entry, whose index is the index (e.g., M) before the fork and whose contents will not advance past the logical clock of the last update before the fork (e.g., $u_{0@M}$). The other two are the postfork entries, whose indices consist of the index before the fork augmented with the history hash of the respective first update after the fork (e.g., the indices are $M \parallel H(u_{1@M})$ and $M \parallel H(u'_{1@M})$). Each of these entries initially holds the logical clock of the first update after the fork; these values advance as the node receives new updates after the fork junction.

Note that this approach works without modification if a faulty node creates a j-way fork, creating updates $u^1_{1@M}$, $u^2_{1@M}$, ..., $u^j_{1@M}$ that link to the same prior update (e.g., $u_{0@M}$). The reason is that, regardless of the order in which nodes detect fork junctions, the branches receive identical names (because branches are named by the first update in the branch). A faulty node that is responsible for multiple dependent forks does not stymie this construction either. After i dependent forks, a virtual node's index in the version vector is well-defined: it is $M \parallel H(u_{fork_1}) \parallel H(u_{fork_2}) \parallel \ldots \parallel H(u_{fork_i})$ [Petersen et al. 1997].

Log exchange revisited. The expanded version vector allows a node to identify which updates to send to a peer. In the basic protocol, when a node $N2$ wants to receive updates from $N1$, it sends its current version vector to $N1$ to identify which updates it needs. After $N2$ detects a fork and splits one version vector entry into three, it simply includes all three entries when asking $N1$ for updates. Note that $N1$ may not be aware of the fork, but the history hashes that are part of the indices of $N2$'s expanded version vector (as per the virtual node construction) tell $N2$ to which branch $N1$'s updates should be applied and tell $N1$ which updates to actually send. Conversely, if the sender $N1$ has received updates that belong to neither branch, then $N1$ and $N2$ identify the new fork junction as described earlier.

Bounding forks. The overhead of this coping strategy is the extra space, bandwidth, and computation to deal with larger version vectors and with conflict detection. However, this overhead is negligible (as shown in Section 8.3). Moreover, the *number* of forks is curtailed by three mechanisms that together bound the number of forks correct nodes will accept before the faulty node is evicted from the system. First, once a correct node learns that a faulty node has created a fork, it has a proof of misbehavior from that node, and it will not communicate with, and thus not accept any new updates directly from, that node. Second, once a correct node is aware of a fork, it accepts a new update by the forking node only if some other node signs an *i-vouch-for-this certificate* that accepts responsibility for having received the update before having learned of the fork. Third, we bound the number of i-vouch-for-this certificates. We give the details in Section 6.7.

6. PROPERTIES AND GUARANTEES

This section defines the properties that Depot enforces and describes how Depot provides them under minimal trust assumptions. The key idea is that the replication protocol enforces *Fork-Join-Causal consistency* (FJC). Given FJC, we can constrain and reason about the order that updates propagate, and use those constraints to help enforce the remaining properties. Figure 3 summarizes these properties and lists the required assumptions.

6.1. Fork-Join-Causal Consistency

Clients expect a storage service to provide consistent access to stored data. Depot guarantees a new consistency semantic for all reads and updates (to a volume) that

Dimension	Safety/ Liveness	Property	Correct nodes required
Consistency	Safety	Fork-Join Causal	Any subset
	Safety	Bounded staleness	Any subset
	Safety	Eventual consistency (s)	Any subset
Availability	Liveness	Eventual consistency (l)	Any subset
	Liveness	Always write	Any subset
	Liveness	Always exchange	Any subset
	Liveness	Write propagation	Any subset
	Liveness	Read availability / durability	A correct node has object
Integrity	Safety	Only auth. updates	Clients
Recoverability	Safety	Valid discard	Any subset
Eviction	Safety	Valid eviction	Any subset
	Safety	Bounded forks	Any subset

Fig. 3. Summary of properties provided by Depot.

are observed by any correct node: *Fork-Join-Causal consistency* (FJC). A formal description of FJC appears in Appendix A. Here we describe its core property.

— *Dependency preservation.* If update u_1 by a correct node depends on an update u_0 by any node, then u_0 becomes *observable* before u_1 at any correct node. (An update u of an object o is *observable* at a node if a read of o would return a version at least as new as u [Frigo and Luchangco 1998].)

To explain FJC, we contrast it with Causal Consistency (CC) in fail-stop systems [Ahamad et al. 1995; Lamport 1978; Petersen et al. 1997]. CC is based on a dependency preservation property that is identical to the one just given, except that it omits the "correct nodes" qualification. Thus, to applications and users, FJC appears almost identical to causal consistency with two exceptions. First, under FJC, a faulty node can issue *forking writes* w and w' such that one correct node observes w without first observing w' while another observes w' without first observing w. Second, under FJC, faulty nodes can issue updates whose stated histories do not include all updates on which they actually depend. For example, when creating the forking updates w and w' just described, the faulty node might have first read updates u_{C1} and u_{C2} from nodes $C1$ and $C2$, then created w that claimed to depend on u_{C1} but not u_{C2}, and finally created update w' that claimed to depend on u_{C2} but not u_{C1}. Note, however, that once a correct node observes w (or w'), it will include w (or w') in its subsequent writes' histories. Thus, as correct nodes observe each others' writes, they will also observe both w and w' and their respective dependencies in a consistent way. Specifically, w and w' will appear as causally concurrent writes by two virtual nodes (Section 5.3).

Though FJC is weaker than linearizability [Herlihy and Wing 1990], sequential consistency [Lamport 1979], or causal consistency, it still provides properties that are critical to programmers. First, FJC implies a number of useful *session guarantees* [Terry et al. 1994] for programs at correct nodes, including monotonic reads, monotonic writes, read-your-writes, and writes-follow-reads. Second, as we describe in the subsections that follow, FJC is the foundation for eventual consistency, for bounded staleness, and for further properties beyond consistency.

Stronger consistency during benign runs. Depot guarantees FJC even if an arbitrary number of nodes fail in arbitrary ways. However, it provides a stronger guarantee—causal consistency—during runs with only omission failures. Of course, causal

consistency itself is weaker than sequential consistency or linearizability. We accept this weakening because it allows Depot to remain available to reads and writes during partitions [DeCandia et al. 2007; Gilbert and Lynch 2002].

6.2. Eventual Consistency

The term *eventual consistency* is often used informally, and, as the name suggests, it is usually associated with both liveness ("eventual") and safety ("consistency"). For precision, we define eventual consistency as follows.

— *Eventual consistency (safety).* Successful reads of an object at correct nodes that observe the same set of updates return the same values.
— *Eventual consistency (liveness).* Any update issued or observed by a correct node is eventually observable by all correct nodes.

The safety property is directly implied by FJC. The liveness property follows from the availability properties in the next subsection.

6.3. Availability and Durability

In this subsection, we consider availability of writes, of update propagation, and of reads. We also consider durability.

The write availability properties described next follow from the replication protocol (Section 5), which entangles updates to prevent selective transmission, and by communication heuristics (Section 7), under which a node that is unable to communicate with a given server then communicates with any other server or client.

— *Always write.* An authorized node can always update any object.
— *Always exchange.* Any subset of correct nodes can exchange any updates that they have observed (assuming that they can communicate, as per the model in Section 3.3).
— *Write propagation.* If a correct node issues a write, eventually all correct nodes observe that write (again assuming the model in Section 3.3, namely that any message sent between correct nodes is eventually delivered).

Unfortunately, there is a limit to what any storage system can guarantee for *reads*: if no correct node has an object, then the object may not be durable, and if no correct, reachable node has an object, then the object may not be available. Nevertheless, we could, at least in principle, still have each node rely only on itself for read availability and durability: nodes could propagate updates and values, and all servers and all clients could store all values. However, fully replicating all data is not appealing for many cloud storage applications.

Depot copes with these limits in two ways. First, Depot provides guarantees on read availability and durability that minimize the required number of correct nodes. Second, Depot makes it likely that this number of correct nodes actually exists. The guarantees are as follows (note that durability—roughly, "the system does not permanently lose data"—manifests as a liveness property).

— *Read availability.* If some correct node N stores a given object's value, then reads to this object at any correct node that can communicate with N in a timely manner will succeed.
— *Durability.* If any correct *hoarding node*, as defined shortly, has an object's value, then a read of that object will eventually succeed. That is, an update is durable once its value reaches a correct node that will not prematurely discard it.

A *hoarding node* is a node that stores the value of a version of an object until that version is garbage collected (Section 6.6). In contrast, a *caching node* may discard a value at any time.

To make it likely that the premise of the guarantees holds—namely, that a correct node has the data—Depot does three things. First, its *configuration* replicates data to survive important failure scenarios. In the single-SSP configuration, all servers usually store values for all updates they receive: except as discussed in the remainder of this subsection, when a client sends an update to a server and when servers transmit updates to other servers, the associated value is included with the update. Additionally, the client that issues an update also stores the associated value, so even if all servers become unavailable, clients can fetch the value from the original writer. Such replication—where an object's hoarding nodes are all servers and the authoring client—allows the system to handle not only the *routine failure* case where a subset of servers and clients fail and lose data but also the *client disaster* and *cloud disaster* cases where all clients *or* all servers fail [Calore 2009; News 2002] or become unavailable [Amazon S3 Team 2008].

Second, *receipts* allow a node to avoid accepting an insufficiently replicated update. When a server processes an update and stores the update's value, it signs a receipt and sends the receipt to the other servers. Then, we extend the basic protocol to require that an update carry either: (a) a *receipt set* indicating that at least k servers have stored the value or (b) the value, itself. In normal operation, servers receive and store updates with values, and clients receive and store updates with receipt sets. However, if over some interval, fewer than k servers are available, clients will instead receive, store, and propagate both updates and values for updates created during this interval. And although servers normally receive updates and values together, there are corner cases where—to avoid violating the *always exchange* property—they must accept an update with only a receipt set. Even in the worst case, therefore, Depot guarantees that if a value is not stored locally, then it is replicated by the client that created it and by at least k servers.

Third, if a client has an outstanding read for version v, it withholds assent to garbage collect v (Section 6.6) until the read completes with either v or a newer version.

6.4. Bounded Staleness

A client expects that soon after it updates an object, other clients that read the object see the update. Depot is therefore designed to allow a client either to know that it has seen all recent updates or else to *suspect* that it has not. The following guarantee codifies what clients can expect.

— *Bounded staleness.* If correct clients $C1$ and $C2$ have clocks that remain within Δ of a true clock and $C1$ updates an object at time t_0, then by no later than $t_0 + 2T_{\text{ann}} + T_{\text{prop}} + \Delta$, either: (1) the update is observable to $C2$ or (2) $C2$ *suspects* that it has missed an update from $C1$.

T_{ann} and T_{prop} are configuration parameters indicating how often a node announces its liveness and how long propagating such announcements is expected to take; reasonable values for T_{ann} and T_{prop} are a few tens of seconds.

Depot relies on FJC consistency to provide the above guarantee, as follows. Every T_{ann} preceding, each client updates a per-client *beacon object* [Li et al. 2004] in each volume with its current physical time. When $C2$ sees that $C1$'s beacon object indicates time t, then $C2$ is guaranteed—by FJC consistency—to see all updates issued by $C1$ before time t. On the other hand, if $C1$'s beacon object does not show a recent time, $C2$ *suspects* that it may not have seen other recent updates by $C1$. At that point, $C2$

can switch to receiving updates from a different server. If that does not resolve the problem, $C2$ can contact $C1$ directly to fetch any missed updates and the updates on which those missed updates depend.

Applications use the preceding mechanism as follows. If a node *suspects* missing updates, then an application that calls GET has two options. First, GET can return a warning that the result might be stale. This option is our default; it provides the *bounded staleness* guarantee. Alternatively, an application that prefers to trade worse availability for better consistency [Gilbert and Lynch 2002] can retry with different servers and clients, blocking until the local client has received all recent beacons.

Note that a faulty client might fail to update its beacon, making all clients *suspicious* all the time. What, then, are the benefits of this bounded staleness guarantee? First, although Depot is prepared for the worst failures, we expect that it often operates in benign conditions. When clients, servers, and the network operate properly, clients are given an explicit guarantee that they are reading fresh data. Second, when some servers or network paths are faulty, *suspicion* causes clients to fail-over to other communication paths to get recent updates. Additionally, stale reads can be reported to the client's administrator who may attempt to diagnose the problem (Is $C1$ down?, Is my network down?, Is my ISP down?, Is the SSP down?, etc.) and repair it.

Bounded staleness vs. FJC. Bounded staleness and FJC consistency are complementary properties in Depot. Without bounded staleness, a faulty server could serve a client an arbitrarily old snapshot of the system's state—and be correct according to FJC. Conversely, bounding staleness without a consistency guarantee (assuming that is even possible; we bound staleness by relying on consistency) is not enough. For engineering reasons, our staleness guarantees are tens of seconds; absent consistency guarantees, applications would get confused because there could be significant periods of time when some updates are visible, but related ones are not.

6.5. Integrity and Authorization

Under Depot, no matter how many nodes are faulty, only authorized clients can update a key-value pair in a way that affects correct clients' reads: the protocol requires nodes to sign their updates, and correct nodes reject unauthorized updates.

A natural question is: how does the system know which nodes are authorized to update which objects? Our prototype takes a simple approach. Volumes are statically configured to associate ranges of lookup keys with specific nodes' public keys. This lets specific clients write specific subsets of the system's objects, and it prevents servers from modifying clients' objects. Implementing more sophisticated approaches to key management [Mazières et al. 1999; Wobber et al. 2010] is future work. We speculate that FJC will make it relatively easy to ensure a sensible ordering of policy updates and access control decisions [Feldman et al. 2010; Wobber et al. 2010].

6.6. Data Recovery

Data owners care about end-to-end reliability. Thus, even if a storage system retains a consistent, fresh view of the data written to it, we have to consider the significant risk from the applications and users above the storage system. More specifically, we have to consider what happens when these applications or users corrupt or destroy data, whether accidentally or intentionally.

Depot does not try to distinguish "good" and "bad" updates or advance the state-of-the-art in protecting storage systems from bad updates. Depot's FJC consistency does, however, provide a basis for applying many standard defenses. For example, Depot can keep all versions of the objects in a volume, or it can provide a basic backup ladder

(all versions of an object kept for a day, daily versions kept for a week, weekly versions kept for a month, and monthly versions kept for a year).

Given FJC consistency, implementing laddered backups is straightforward. Initially, servers retain every update and value that they receive, and clients retain the update and value for every update that they create. Then, servers and clients discard the nonladdered versions by *unanimous consent of clients*. This process works as follows, at a high level (tedious details are omitted). Every day, clients garbage collect a prefix of the system's logs by producing a checkpoint of the system's state, using techniques adopted from Bayou [Petersen et al. 1997]. The checkpoint includes information needed to protect the system's consistency and a *Candidate Discard List* (CDL) that states which prior checkpoints and which versions of which objects may be discarded. The job of proposing the checkpoint rotates over the clients each day.

The keys to correctness here are: (a) a correct client will not sign a CDL that would delete a checkpoint prematurely and (b) a correct node discards a checkpoint or version if and only if it is listed in a CDL signed by *all* clients. These checks ensure the following property.

— *Valid discard.* If at least one client is correct, a correct node will never discard a checkpoint or a version of an object required by the backup ladder.

Note that a faulty client cannot cause the system to discard data that it needs: the preceding approach provides the same read availability and durability guarantees (Section 6.3) for backup versions as for the current version. A faulty client can, however, delay garbage collection, though if a checkpoint fails to garner unanimous consent, clients notify an administrator, who troubleshoots the faulty client or, if all else fails, replaces it with a new machine. Thus, faulty clients can cause the system to consume extra storage, but only temporarily, assuming that unresponsive clients are eventually repaired or replaced (Section 3.3).

6.7. Evicting Faulty Nodes and Bounding Forks

Recall that a faulty node can weaken consistency by issuing a pair of illegal *forking updates* such that neither update depends on the other. We now explain how Depot evicts provably faulty nodes and how it bounds the number of forks that faulty nodes introduce.

Eviction of a provably faulty node is triggered when a correct node first observes a pair of updates from different forks, thus providing a *Proof Of Misbehavior* (POM) against the faulty node. At that point, the correct node refuses to accept new updates from the faulty node directly; moreover, since the POMs propagate, all correct nodes will eventually refuse to communicate with the provably faulty node. This mechanism ensures that, if all nodes other than N are correct, then N can introduce at most $n - 1$ forks in an n-node system before all nodes stop communicating with it. However, absent any more checks, a faulty node could *launder* infinitely many forks through another node that has not issued forking updates yet. Unfortunately, laundering a forking update is not itself evidence of misbehavior, as a node propagating one of the updates might simply be unaware of the fork.

To mitigate laundering while allowing for correct nodes that unwittingly launder, Depot does the following. Before a correct node accepts updates created by a provably faulty node, the correct node requires i-vouch-for-this certificates from nodes not known to be faulty. Such a certificate is created by a correct node when it first learns of a fork; the certificate contains a single entry version-and-hash vector that covers all previously received updates issued by the faulty node. Then, the i-vouch-for-this certificate propagates with the forking updates via log exchange, allowing each node

to maintain the invariant that, for all updates that it has received by the faulty node after the fork junction, it has received i-vouch-for-this certificates covering those updates. Moreover, a correct node issues at most one such certificate per faulty node, which, as we will see shortly, helps to bound the number of forks.

Unfortunately, we are still not done because a faulty node $N2$ could create inconsistent i-vouch-for-this certificates and thereby introduce additional forks by N into the system. To handle this case, a correct node treats conflicting i-vouch-for-this certificates roughly as it treats forking updates: it views the conflicting certificates as a proof of misbehavior by $N2$, stops communicating directly with $N2$, creates an i-vouch-for-this certificate for the updates that it has already received from $N2$, supplies $N2$'s conflicting i-vouch-for-this certificates to peers during log exchange, and demands i-vouch-for-this certificates for any new updates by $N2$ that it receives.

Next, we show that, as a result of the POMs and i-vouch-for-this certificates, k colluding faulty nodes in a group of n nodes cannot cause correct nodes to evict faulty nodes or observe more than $(n - k) \cdot (2^k - 1)$ forks (this number might seem high, but we expect POMs to circulate and cut off faulty nodes much more quickly than that). Thus, Depot ensures the following.

— *Valid eviction.* No correct node is ever evicted.
— *Bounded forks.* In a n-node system with k faulty nodes, no correct node will ever observe more than $(n - k) \cdot (2^k - 1)$ forks introduced by faulty nodes.

We next explain the intuition for these two properties based on our eviction and update checks.

Valid eviction. Eviction occurs only if nodes sign messages constituting a cryptographic proof of misbehavior. If a faulty node is merely unresponsive, that is handled exactly as SLA violations are today.

Bounded forks. Toward an upper bound, we first consider the conditions under which faulty nodes can produce the maximum number of forks. To do so, faulty nodes must expose different sets of forks to different correct nodes. Moreover, these correct nodes should not exchange i-vouch-for-this certificates or POMs until the very end of execution, when faulty nodes have revealed as many forks as possible to each correct node. (If a correct node received a POM or i-vouch-for-this certificate earlier, a faulty node could be flagged as such, causing correct nodes to reject forks, lowering the number of exposed forks.) Based on the foregoing, the number of forks observed by a set of correct nodes will be a product of the number of correct nodes and the maximum number of forks that each correct node can observe.

We now argue that the maximum number of forks observed by a correct node in the presence of k colluding faulty nodes is $2^k - 1$. We induct over k. For the base case, $k = 1$, a correct node can observe at most one fork if there is one faulty node. For the induction step, we assume the claim holds for k.

Now, consider an execution with 1 correct node and $k + 1$ colluding faulty nodes ($k \geq 1$) that exposes the maximum number of forks to the correct node C. Note that in an execution that exposes the maximum number of forks to correct nodes, each faulty node will attempt to expose multiple forks to correct nodes; otherwise, in an execution with multiple faulty nodes, we can increase the number of forks exposed to correct nodes by laundering one faulty node's forks through other faulty nodes. Now, because of our eviction checks, multiple forks from a node must be accompanied by a corresponding i-vouch-for-this certificate by a node not known to be faulty. Hence, in an execution that contains multiple faulty nodes and in which correct nodes observe the maximum number of forks, each correct node will ultimately receive i-vouch-for-this

certificate(s) against at least some faulty node. Now consider the last such i-vouch-for-this certificate that a correct node C accepts; since all nodes other than C are faulty, this i-vouch-for-this has been created by a faulty node N. Since C *accepts* this i-vouch-for-this certificate, C must not have observed an i-vouch-for-this certificate against N yet.

Hence, the execution until C's acceptance of N's last i-vouch-for-this certificate must look like a correct execution with 2 correct nodes and k faulty nodes. By the induction hypothesis and by our worst-case assumption that correct nodes are observing independent forks, the correct node C and the seemingly correct node N can together accept a maximum of $2 \cdot (2^k - 1) = 2^{k+1} - 2$ forks in this execution prefix. However, in addition to these forks that are created by the other k faulty nodes, the correct node C has also accepted updates (in the form of i-vouch-for-this certificates) from the seemingly correct, but actually faulty, node N. These updates constitute yet another fork that has been observed by C (even though C may only later realize that the node N is faulty). Therefore, C can observe a maximum of $2^{k+1} - 2 + 1 = 2^{k+1} - 1$ forks. Hence, the claim holds for $k + 1$.

Combining the bound on the maximum number of forks that each correct node can observe in the presence of k colluding faulty nodes with the claim that each correct node could observe independent distinct forks (in the worst case), we get our desired bound that k colluding faulty nodes can force $n-k$ correct nodes to observe a maximum of $(n - k) \cdot (2^k - 1)$ forks.

7. IMPLEMENTATION

Figure 4 depicts Depot's software architecture. The architecture comprises modules that enforce the properties described in Section 6. The *core replication* module, which enforces FJC consistency on updates, is at the core of the system. The other modules are layered on top of this module, in one or both of two ways. First, the *bounded staleness* and *availability and durability* modules are invoked when the client GETs and PUTs. Second, the *eviction*, *valid discard*, and *integrity* modules, and the receipt management logic in the availability and durability module are filters that screen incoming updates en route to the core replication module (as indicated by the horizontal block arrows in Figure 4) and optionally issue read/write requests to the core replication module (as indicated by the thin arrows in Figure 4).

Two design choices simplify the filter implementation in Depot. First, Depot nodes exchange collections of updates called *bundles* that are processed atomically by all the modules: either all the updates in a bundle are accepted or all are rejected. Second, these screening modules are implemented using a two-phase approach. In the first phase, each module performs its checks and votes for accepting or rejecting the bundle without committing any changes. In the second phase, the changes are made persistent if no module voted to reject the bundle in the first phase. Hence, if an update bundle is rejected by a module that appears later on the update path, then changes in earlier modules that are influenced by that update bundle are also rolled back, which avoids inconsistencies in module state.

In the remainder of this section, we detail the implementation of GETs and PUTs, describe optimizations that save bandwidth and storage, and say a few words about our prototype.

Implementation of GET *and* PUT. The main complexity in implementing GETs and PUTs in Depot results from the separation of data and metadata. Recall that an *update* is only the metadata (Section 5.1) and that the core replication module replicates updates across a configurable topology and at a configurable frequency. In our prototype, servers exchange updates with each other, whereas each client node chooses a (usually

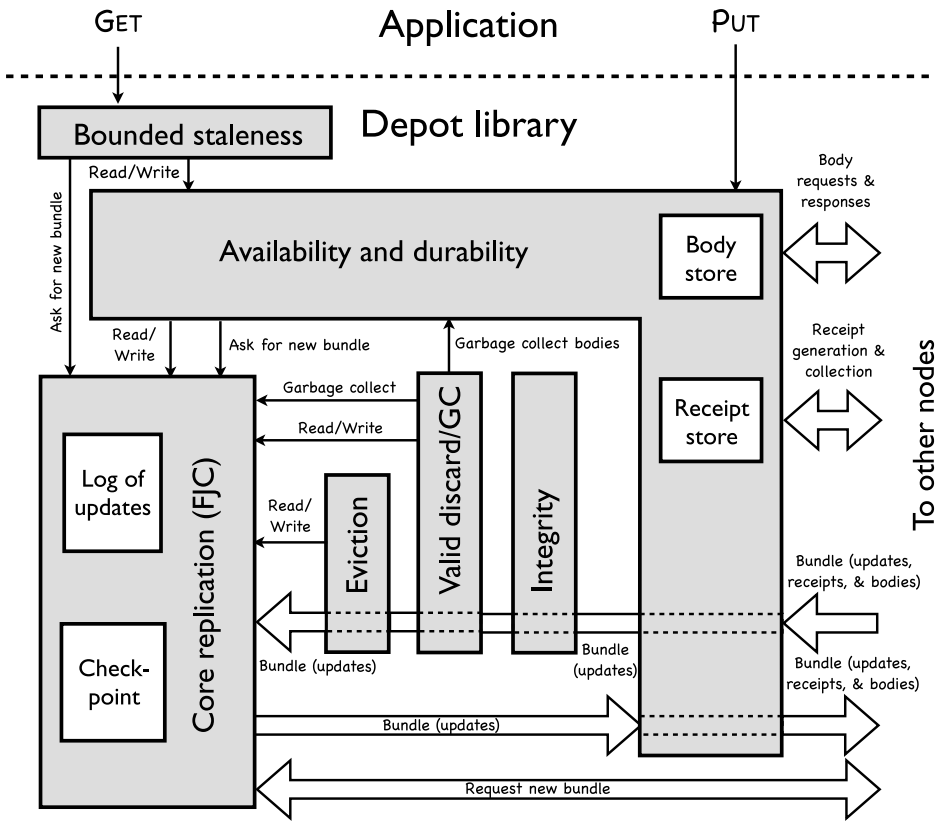

Fig. 4. Overview of Depot's software architecture. Block arrows denote messages, and thin arrows denote function calls. The core replication module replicates updates and ensures fork-join-causally consistent access to these updates. These updates are exchanged as collections called *bundles*. A bundle is processed atomically by all modules in that all of the updates in a bundle are accepted or all are rejected. The availability module separates data from metadata by handing hashes of values to the core replication module. The availability module also fetches values on GETs and ensures that insufficiently replicated updates are accompanied by their corresponding values, as described in Section 6.3. Likewise, the bounded staleness, eviction, valid discard, and integrity modules implement the checks described in Section 6. These modules are implemented as filters: Incoming bundles that do not satisfy the checks are rejected.

nearby) *primary server* and fetches updates from that server periodically. The availability and durability module selectively replicates values depending on the receipt logic and depending on whether a node is configured as a client or as a server; this module also serves GETs and PUTs.

On a PUT, a client creates an update and stores the value and update locally. Next, the client attempts to transfer the update and value to the client's primary server for ensuring the availability and durability properties (Section 6.3) and for receipt generation. As an optimization, rather than initiate the log exchange protocol, a client sends only the update and value of the PUT to its primary server. If the update passes all consistency checks and if the value matches the hash in the update, the server adds these items to its log and checkpoint. Otherwise, the client and server fall back on log-exchange.

On receiving a new update and value, the primary server forwards the update and associated value to the other servers for receipt generation. Each server verifies the update and value and, if the verification succeeds, responds with a receipt. The

primary server gathers these receipt responses into a receipt set and sends this receipt set to the client. The client's availability module stores these receipts and attaches them to the updates that are propagated via log exchange.

On a GET, a client first checks the beacons needed to enforce the bounded staleness property (Section 6.4). If the beacon checks succeed, the client requests the value for the specified key k, by sending k along with a *value-digest* to the client's primary server. The value-digest is a set of two-byte digests, one per logically latest update of k that the client has received via background gossip; note that unless there are concurrent updates to k, the value-digest contains just one element.

If the value-digest matches the latest updates known to the server, the server responds with the corresponding values. The client then checks that these values correspond to the $H(value)$ entries in the previously received updates. If so, the client returns the values, completing the GET. If the server rejects the value-digest or if the values do not match, then the client initiates a value and update transfer by sending to its primary server: (a) its version vector and (b) k. The server replies with: (a) the missing updates, which the client verifies (Section 5.2), and (b) the most recent set of values for k.

If a client cannot reach its primary server, it randomly selects another server (and does likewise if it cannot reach that server). If no servers are available, the client enters client-to-client mode for a configurable length of time, during which it gossips with the other clients. In this mode, on a PUT, the client responds to the application as soon as the data reaches the local store. On a GET, the client fetches the values from the clients that created the latest known updates of the desired key.

Optimizing bandwidth and storage for scalability. Bandwidth and storage are the most expensive resources that Depot uses. To reduce its consumption of these resources, Depot reduces the average size of an update and its associated *dVV* to about 300 bytes. It does so via several optimizations.

— The dependency version vector, *dVV*, included with each update (Section 5.2) logically specifies a full version vector. However, an update by node N includes only the entries that have changed since node N's last write. Since updates are received in FIFO order from each node, a receiver can always construct the full version vector from the *dVV* [Schiper et al. 1991].
— We also use a similar technique to optimize the *gossip request* message sent when a node asks another for recent updates. Though the requester logically sends its entire version vector VV, we optimize for the case of repeated fetches from the same node by sending a diff since the last request and resending the full version vector only if the nodes lose synchronization.
— Lastly, when sending an update, we further optimize the *dVV* encoding by exploiting the temporal locality of various logical times present in an update's *dVV* and its *accept stamp*. We expect that in a Depot deployment, these logical times will be close to one another. Therefore, when sending *dVV*, we transmit only the delta between the logical time in the *dVV* entries and the *accept stamp*. Temporal locality makes this delta small, and smaller integers take fewer bytes to transmit (our implementation uses a custom serialization), reducing the overall bandwidth consumed by *dVV*.

These optimizations ensure that version vectors create very little bandwidth and storage overhead. For example, with 8 nodes, the first and third optimizations make the size of an encoded *dVV* only a few bytes. Similarly, in the common case, a gossip request message is only a few bytes long.

Depot adds modest latency relative to a baseline system. Depot's additional GET latency is comparable to checksumming data with SHA-256. For PUTs, 99-percentile latency for 10KB objects increases from 14.0 ms to 23.3 ms.	§8.1
Depot's main resource overheads are client-side storage and client- and server-side CPU use.	§8.1
Depot imposes little additional cost for read-mostly workloads. For example, Depot's weighted dollar cost of 10KB GETs and PUTs are 2% and 50% higher than the baseline.	§8.2
When failures occur, Depot continues operating correctly, with little impact on latency or resource consumption.	§8.3

Fig. 5. Summary of main evaluation results.

Baseline	Clients trust the server to handle their PUTs and GETs correctly. Clients neither maintain local state nor perform checks on returned data.
B+Hash	Clients attach SHA-256 hashes to the values that they PUT and verify these hashes on GETs.
B+H+Sig	Clients sign the values that they PUT and verify these signatures on GETs.
B+H+S+Store	The same checks as B+H+Sig, plus clients locally store the values that they PUT, for durability and availability despite server failures.

Fig. 6. Baseline variants whose costs we compare to Depot's. Two of the main differences between B+H+S+Store (the strongest baseline) and Depot are that Depot includes history hashes and that Depot uses local storage at each client.

Prototype. Our prototype is implemented in Java. It keeps every version written so does not implement laddered backups or garbage collection (Section 6.6); implementing these features is future work. The implementation is otherwise complete (but not optimized). It uses Berkeley DB (BDB) for local storage and does so synchronously: after writing to BDB, Depot calls *commit* before returning to the caller, and we configure BDB to call `fsync` on every commit.[7]

8. EXPERIMENTAL EVALUATION

In evaluating Depot, our principal question is: what is the "price of distrust?" That is, how much do Depot's guarantees cost, relative to a baseline storage system? We measure latency, network traffic, storage at both clients and servers, and CPU cycles consumed at both clients and servers (Section 8.1). We then convert the resource overheads into a common currency [Gray and Shenoy 2000] using a cost model loosely based on the prices charged by today's storage and compute services (Section 8.2). We then move from "stick" to "carrot", illustrating Depot's end-to-end guarantees under faults (Section 8.3). Figure 5 summarizes our results.

Method and environment. Most of our experiments compare our Depot implementation to a set of *baseline* storage systems, described in Figure 6. All of them replicate key-value pairs to a set of servers, using version vectors to detect precedence, but omit

[7]This approach aids, but does not quite guarantee, persistence of committed data: "Synchronous" disk writes in today's systems do not always push data all the way to the disk's platter [Nightingale et al. 2008]. Note that if a node commits data and subsequently loses it because of an ill-timed crash, Depot handles that case as it does any other faulty node.

some of Depot's safeguards. In none of the variants do clients check version vectors or maintain history hashes. These baselines use the same code base as Depot, so they are not heavily optimized. For example, as in Depot, the baselines separate data from metadata, causing writes to two Berkeley DB tables on every PUT, which may be inefficient compared to a production system. Such inefficiencies may lead to our underestimating Depot's overhead.

Our default configuration is as follows. There are 8 clients and 4 servers with the servers connected in a mesh and two clients connecting to each server. This configuration models the example scenario in which the Depot clients are running on the Web servers of a medium-sized Web service (Section 3). Servers gossip with each other once per second; a client gossips with its primary server every 5 seconds. RSA moduli are 1024 bits, and our hash function is SHA-256. We experiment with a slightly older implementation that runs without receipts (Section 6.3) and beaconing (Section 6.4). Since receipts require signature checks, our current evaluation slightly understates overhead.

Our default workload is as follows. Clients issue a sequence of PUTs and GETs against a volume preloaded with 1000 key-value pairs. We partition the write key set into several nonoverlapping ranges, one for each client. As a result, a GET returns a single value, never a set. A client chooses write keys randomly from its write key range and read keys randomly from the entire volume. We fix the key size at 32 bytes. In each run, each client issues 600 requests at roughly one request per second. We examine three different value sizes (3 bytes, 10KB, and 1MB) and the following read-write percentages: 0/100, 10/90, 50/50, 90/10, and 100/0. (We do not report the 10/90 and 90/10 results; their results are consistent with, and can be predicted by, those from the other workloads.)

We use a local Emulab [White et al. 2002]. All hosts run Linux FC 8 (version 2.6.25.14-69) and are Dell PowerEdge r200 servers, each with a quad-core Intel Xeon X3220 2.40 GHz processor, 8GB of RAM, two 7200RPM local disks, and one Gigabit Ethernet port.

8.1. Overhead of Depot

Microbenchmarks. To put our results in perspective, we begin by measuring the costs of low-level cryptographic and storage operations that Depot and the baseline systems compose to execute a single PUT or GET. The cryptographic operations are SHA-256, RSA-Sign, and RSA-Verify and use the Sun Java security library. We issued 1000 operations, measuring CPU time and latency. Similarly for local BDB storage operations, we issue 1500 local "DB get" or "DB put" operations on a dummy table on randomly chosen keys from a set of 1000 keys, measuring latency and CPU utilization for various object sizes. We set the BDB cache to 100MB. The statistics from these runs are reported in Figure 7. The BDB latencies have significant variance, which we speculate comes from variation in disk access times.

Latency. To evaluate latencies in Depot and the baseline systems, we measure from the point of view of the application, from when it invokes GET or PUT at the local library until that call returns. Note that for a PUT, the client commits the PUT locally (if it is a Depot or B+H+S+Store client) and only then contacts the server, which replies only after committing the PUT. We report means, standard deviations, and 99th percentiles, from the GET (i.e., 100/0) and PUT (i.e., 0/100) workloads.

Figure 8 depicts the results. For the GET runs, the difference in means between Baseline and B+Hash are 0.0, 0.2, and 14.9 ms for 3B, 10KB, and 1MB, respectively, which are explained by our measurements (Figure 7) of mean SHA-256 latencies in

Operation	Size	Latency (ms) ($\mu \pm \sigma$)	CPU (ms/req)
SHA-256	3B	0.1±0.2	0.0
SHA-256	10KB	0.2±0.4	0.0
SHA-256	1MB	15.7±0.5	14.2
RSA-Sign	300B	4.2±0.7	3.2
RSA-Verify	300B	0.3±0.5	0.0
BDB local get	3B	0.2±0.5	1.0
BDB local get	10KB	0.3±0.9	1.2
BDB local get	1MB	7.6±8.6	10.1
BDB local put	3B	1.3±1.9	1.0
BDB local put	10KB	2.6±2.4	1.2
BDB local put	1MB	19.3±12.4	9.4

Fig. 7. Statistics for the costly low-level operations that Depot uses, which contribute to end-to-end costs.

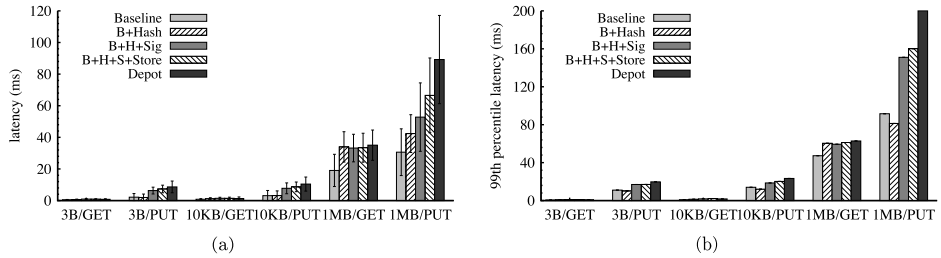

Fig. 8. Latencies ((a) mean and standard deviation and (b) 99th percentile) for GETs and PUTs for various object sizes in Depot and the four baseline variants. For small- and medium-sized requests, Depot introduces negligible GET latency and sizeable latency on PUTs, the extra overhead coming from signing, synchronously storing a local copy, and Depot's additional checks.

the cryptographic library that Depot uses:[8] 0.1, 0.2, and 15.7ms for those object sizes. Similarly, the means of RSA-Verify operations explain the difference between B+Hash and B+H+Sign. The GET latency for Depot itself, with the two smaller object sizes, is, surprisingly, lower than that of the strongest two baselines because Depot clients verify signatures in the background (as part of exchanging updates), whereas the baselines do so on the critical path. With 1MB objects, Depot's GET latency is comparable to that of B+Hash and all stronger baselines. Thus, for GETs, Depot does not introduce much latency beyond applying a collision-resistant hash to data stored in an SSP, which prudent applications likely do anyway.

For PUTs, the latency is higher. Each step from B+Hash to B+H+Sign to B+H+S+Store to Depot adds significantly to mean latency, and for large requests, going from Baseline to B+Hash does as well. For example, the mean latency for 10KB PUTs ascends 3.1ms, 3.1ms, 7.8ms, 8.7ms, 10.4ms as we step through the systems; 99th percentile latency goes 14.0ms, 12.0ms, 18.6ms, 20.2ms, 23.3ms.

We can explain the observed Depot PUT latency with a model based on our preceding measurements of the main steps in the protocol (see Figure 7). For example, for 10KB PUTs, the client hashes the value (mean measured time: 0.2ms), hashes history ($\approx$ 0.1ms), signs the update (4.2ms), stores the body and update (2.6ms, with the DB cache enabled), and transfers the update and body over the 1Gbps network ($\approx$ 0.1ms); the server verifies the signature (0.3ms), hashes the value (0.2ms), hashes history ($\approx$ 0.1ms), and stores the body and update (2.6ms). The sum of the means (10.4ms)

[8]FlexiProvider: `www.flexiprovider.de`.

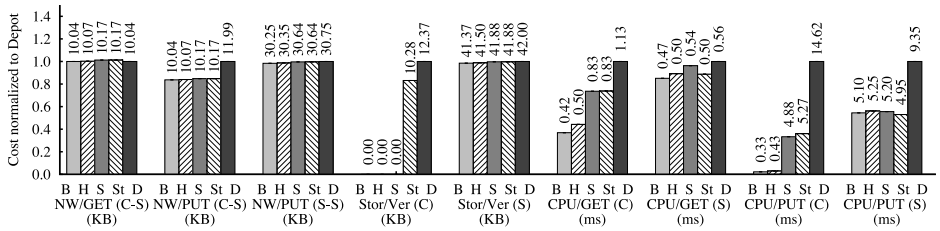

Fig. 9. Per-request average resource use of Baseline (B), B+Hash (H), B+H+Sig (S), B+H+S+Store (St), and Depot (D) in the 100/0 (GET) and 0/100 (PUT) workloads with 10KB objects. The bar heights represent resource use normalized to Depot. The labels indicate the absolute per-request averages. (C) and (S) indicate resource use at clients and servers, respectively. (C-S) and (C-S) are client-server and server-server network use, respectively. For storage costs, we report the cost of storing a version of an object.

matches the observed latency (10.4ms). The model is similarly accurate for the 3B and 1MB experiments.

These PUT latencies could be reduced. For example, we have not exploited obvious pipelining opportunities. Also, we experiment on a 1Gbit/s LAN; in many cloud storage deployments, WAN delays would dominate latencies, shrinking Depot's percentage overhead.

Resource use. Figure 9 depicts the average use of various resources in the latency experiments with 10KB objects. We measure CPU use at the end of a run, summing the user and system time from `/proc/<pid>/stat` on Linux and dividing by the number of requests. We measure network use as the number of bytes handed to TCP. We report the overhead of a PUT as including the cost of propagating the resulting update to all the nodes (clients and servers) and storing the update at these nodes.

Depot's overheads are small for network use, server storage, and server CPU on GETs. They are also small for client CPU on GETs, relative to the B+H+Sign baseline. The substantial client storage overheads result from clients' storing data for the PUTs that they create and metadata for all PUTs. The substantial PUT CPU overheads are due to additional Berkeley DB accesses (which cost CPU cycles, per our microbenchmarks) and cryptographic checks, which happen intensively during gossiping. Since the request rate is low relative to the gossip rate, each request pays for a lot of gossip work. With increased request rate or larger objects (as in the experiments summarized immediately below) this CPU overhead will be lower.

Throughput. Most of our evaluation is about Depot's underlying costs as opposed to the performance of the prototype, so we treat throughput only briefly. We ran separate measurements in which we saturated a single Depot server with requests from many clients. For 10KB GETs, a single Depot server can handle 11k requests per second, at which point network bandwidth is the bottleneck. For 10KB PUTs, peak throughput is 700 requests per second. This disappointing number is not surprising given the resource use just reported, but a well-tuned version ought to see sequential disk bandwidth with the bottleneck being signature checks (0.3ms per core).

8.2. Dollar Cost

Is Depot's added consumption of CPU cycles and client-side storage truly costly? To answer this question, we must weight Depot's resource use by the costs of the various resources. To do so, we convert the measured overheads from the prior subsection into

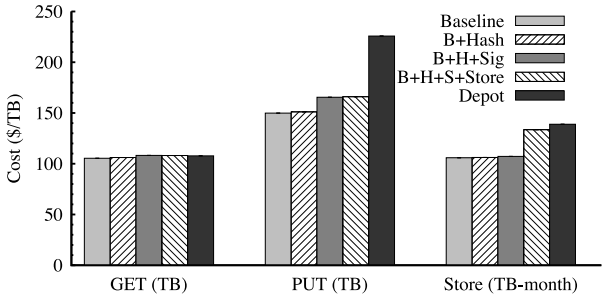

Fig. 10. Dollar cost to GET 1TB of data, PUT 1TB of data, or store 1TB of data for 1 month. Each object has a small key and a 10KB value. 1TB of PUTs or GETs corresponds to 10^8 operations, and 1TB of storage corresponds to 10^8 objects.

dollars (to pick a convenient currency). We use the following cost model, loosely based on what customers pay to use existing cloud storage and compute resources.

Client-server network bandwidth	$.10/GB
Server-server network bandwidth	$.01/GB
Disk storage (one client or server)	$.025/GB per month
CPU processing (client or server)	$.10 per hour

For intuition, note that 4ms of CPU time to sign a small message costs about the same as sending 1KB between a client and a server or storing 4KB at one node for a month.

Figure 10 shows the overheads from Figure 9 weighted by these costs. Depot's overheads are modest for read-mostly workloads. Depot's GET costs are only slightly higher than Baseline's: \$107.80 versus \$105.40 for 10^8 GET operations on 10KB objects. However, Depot's PUT costs are over 50% higher: \$225.80 versus \$149.90 for 10^8 operations on 10KB objects. Most of the extra cost is from gossiping, so the relative overheads would fall for larger objects or more frequent updates. Depot's storage costs are 31% higher than Baseline's: \$138.90 versus \$105.90 to store 10^8 10KB objects for a month. Most of the extra cost is from storing a copy of each object at the issuing client; the rest is from storing metadata.

8.3. Experiments with Faults

We now examine Depot's behavior when servers become unavailable and when clients create forking writes.

Server unavailability. In this experiment, 8 clients access 8 objects on 4 servers. The objects are 10KB, and the workload is 50/50 GET/PUT. Servers gossip with random servers every second, and clients gossip with their chosen partner (initially a server) every 5 seconds. At about 220 seconds into the experiment, we kill all servers, and 300 seconds after that, we start a set of replacement servers on the same physical nodes. The replacement servers do not retain the pre-failure state; they begin with fresh state.

By postprocessing logs, we measure the *staleness* of GET results, compared to instantaneous propagation of all updates: the staleness of a GET's result is the time since that result was overwritten by a later PUT. If the GET returns the most recent update, the staleness is 0. We also report the latency of GETs and PUTs issued by a client. Finally, we measure CPU consumption at each client; we create a time series by combining the second-on-second increase in user time and the second-on-second increase in system time, as obtained from `/proc/<pid>/stat`. This CPU consumption

roughly corresponds to the per-request CPU cost because, in these experiments, clients issue one request (PUT or GET) every second.

Figure 11(a) depicts the staleness observed at one client. Before the servers fail, GETs in both Depot and B+H+S+Store have low staleness. After the failure, B+H+S+Store blocks forever. Depot, however, switches to client-to-client mode, continuing to service requests. Later, when a new set of servers is started, these servers rebuild their state from the clients and the clients switch back to client-server mode. Staleness increases noticeably in client-to-client mode because: (1) disseminating updates takes more network hops and (2) the lower gossip frequency increases the delay between hops. However, on recovery, the staleness value falls back to its original value.

Figure 11(b) depicts the latency of GETs observed by the same client. Prior to the failure, Depot's GET latency is significantly higher than in the experiments in Section 8.1 because each object is often updated (because there are few objects in the workload), so the optimization described in Section 7 often fails, making the client and server perform a log exchange to complete the GET. When the servers fail, Depot continues to function, and GET latency actually improves: rather than requesting the "current value" from the server (which requires a log exchange to get the new metadata for validating the newest update), in client-to-client mode, a client fetches the version mentioned in the update it already has from the writer.

Figure 11(c) shows the PUT latency observed by the same client. PUT latency follows a pattern similar to the GET latency and improves in client-to-client mode: PUTs return as soon as the update and value are stored locally, with no round trip to a server. On recovery, PUT latency rises to its former value.

Figure 11(d) depicts the per-request CPU utilization observed by the same client. Like latency, the mean per-request CPU utilization drops slightly on failure because less work is happening in the system (no transfers on PUTs, fewer fetches on GETs). On recovery, the mean per-request CPU utilization rises again.

Client fork. In this experiment, 8 correct clients (8C0F), 6 correct clients and 2 faulty clients (6C2F), and 6 correct clients (6C0F) access 1000 objects on 4 servers. The objects are 10KB, and the workload is 50/50 GET/PUT. 300 seconds into the experiment, faulty clients begin to issue forking writes. When a correct client observes a fork, it publishes a Proof Of Misbehavior (POM) against the faulty client, and when servers or other clients receive the POM, they stop accepting new writes directly from the faulty client.

Figure 12(a) and Figure 12(b) depict the results for GETs and PUTs respectively. Forks introduced by faulty clients do not have an obvious effect on GET or PUT latency; note that the spikes in GET latency prior to $t = 300$ are unrelated to client failures.

Figure 12(c) shows CPU consumption over the trailing second (as described in the discussion of Figure 11(d)) at a correct client during this experiment. Any additional processing caused by the forking clients is small compared to the normal variation that we see across time for all configurations of this experiment.

9. TEAPOT FOR LEGACY SSPS

Depot runs on both clients and SSP nodes, but it would be desirable to provide Depot's guarantees using unmodified legacy SSPs such as S3, Azure Storage, or Google Storage. Intuitively, such an approach appears possible. In Depot, servers must: (1) propagate updates among clients and (2) provide update bodies (i.e., values) in response to GET requests. We should be able to use an SSP's abstract key-value map as a communication channel and as storage for update bodies. And because Depot clients verify everything that they receive from servers, we should still be able to provide most of the properties discussed in Section 6. In this section, we give a brief overview of *Teapot*,

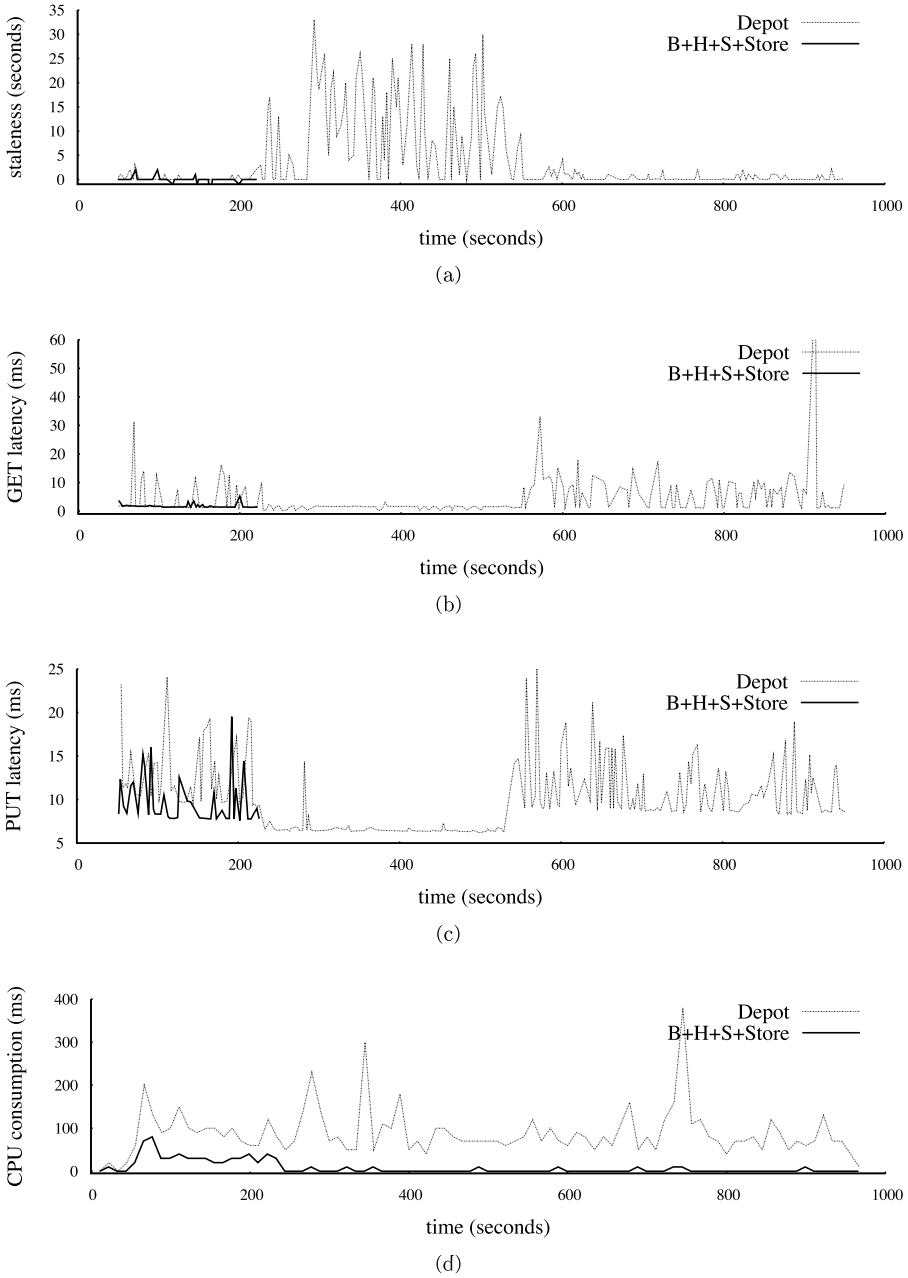

Fig. 11. The effect of total server failure (t = 220) and failover to a fresh set of servers (t = 520) on: (a) staleness; (b) GET latency; (c) PUT latency; (d) CPU consumption over the trailing second. The workload is 50/50 R/W and 10KB objects chosen from a volume containing 8 objects. Depot maintains availability through client-to-client transfers whereas the baseline system blocks, and GET latency, PUT latency, and CPU consumption actually improves (at the expense of staleness).

a variation of Depot that uses legacy SSPs. We then compare Teapot and Depot and discuss how legacy SSPs could be extended to support all of Depot's properties.

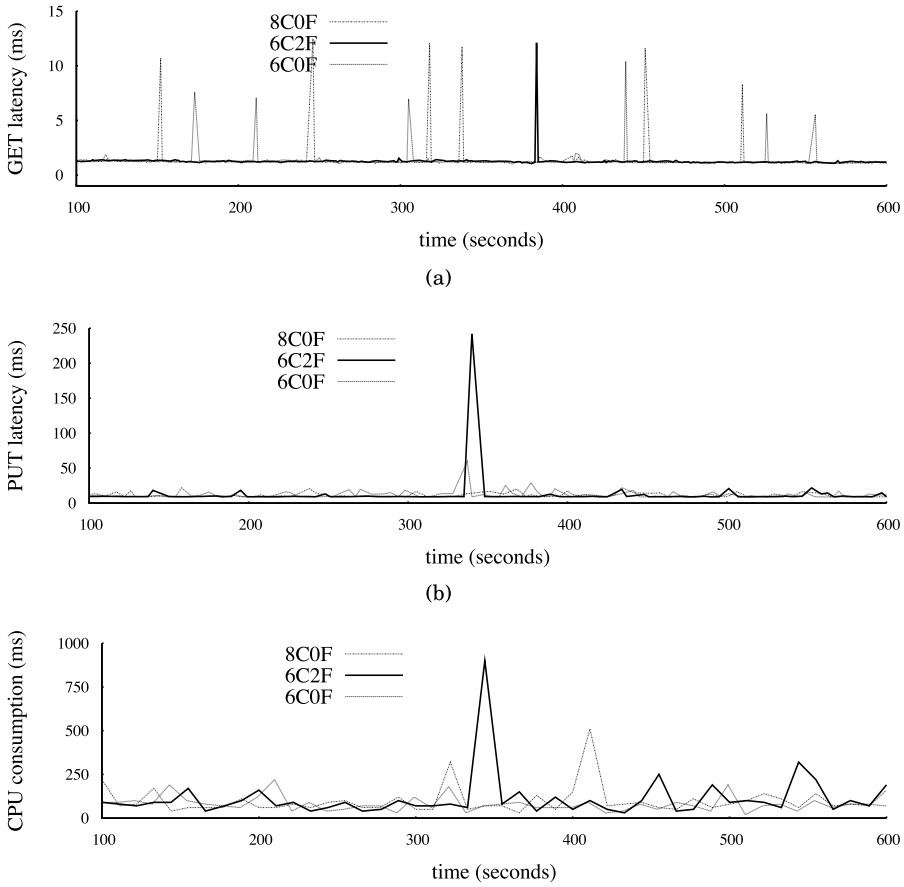

Fig. 12. PUT and GET latency and CPU consumption over the trailing second at a correct client with Depot in three runs: 8 correct clients (8C0F), 6 correct clients and 2 faulty clients (6C2F), and 6 correct clients (6C0F). Depot survives a fork introduced at 300 seconds into the execution without affecting the client-perceived latency and the CPU consumption.

Teapot assumes an API like that of S3 (Figure 13): LPUT(k, v, b) (associate v with k in a bucket b owned by a given client) and LGET(k, b) (return v). On a PUT, the Teapot client creates and locally stores the metadata u (a Depot update) and the data d (a Depot value). The client then stores both to the SSP by calling LPUT$(H(u), u, b_c)$ and LPUT$(H(d), d, b_c)$, where b_c is a bucket that only c can write. The client then identifies its latest update by storing it to a distinguished key, k_c^* (that is, the client executes LPUT(k_c^*, u, b_c)). In the background, the client periodically fetches the other clients' latest updates by reading their k_c^* entries and then fetching and validating the updates' dependencies. On a GET, the Teapot client uses LGET to retrieve the value(s) associated with the latest update(s) that it has received.

We have prototyped Teapot using S3 and a variation on the arrangement just sketched. As shown in Figure 13, assessing S3 through Teapot rather than through LPUT and LGET introduces little latency over S3; the baseline latencies to S3 are already scores of milliseconds, so the additional overheads are small. The resource costs are similar to those of Depot (Section 8.1).

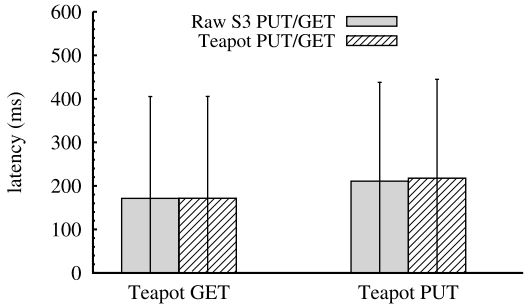

Fig. 13. Average latencies (with standard deviations) perceived by Teapot for GET and PUT operations with 10KB payload when using Amazon S3 for storage.

Discussion. Teapot has two key differences from Depot. First, if a client fails in particular ways, Teapot cannot guarantee *valid discard* (Section 6.6). A client can, for example, issue a PUT, allow the update to be observed by other clients, and then delete the associated value. This unilateral deletion may cause a correct SSP to discard the current value of a key. Depot, in contrast, meets *valid discard* as long as at least one client is correct. Second, Teapot servers cannot provide the durability receipts that Depot clients use to avoid depending on insufficiently replicated data (Section 6.3). Note that Teapot tolerates arbitrary SSP failures and many other client failures (crashes, forks, etc.), so Teapot's additional vulnerability over Depot is limited and may be justified by its deployability.

We now ask: what incremental extensions to SSPs would allow us to run code only on clients but recover Depot's full guarantees? We speculate that the following suffices. First, to prevent a correct client from having to depend on updates that a faulty client could delete, the SSP could implement LINK($K, b_c, b_{c'}$), UNLINK($k, b_c, b_{c'}$), and VERIFY(k, H, b_c). LINK causes every existing or new key-value pair in a key range K in one client's bucket (b_c) to be *linked to* another client's bucket ($b_{c'}$), where a key-value pair that is the target of a link by another bucket may not be modified or deleted. UNLINK removes such a link. VERIFY checks that the SSP stores a value with hash H for key k in bucket b_c. Then, if a client LINKs to other clients' buckets when it joins the system and VERIFIES an update's value before accepting the update into its history, we can restore unanimous consent, and hence *valid discard*, for garbage collecting versions (Section 6.6). Second, to assure clients that updates are sufficiently replicated, the SSP could return a receipt in response to LPUT that the clients could use like receipt sets (Section 6.3). These extensions seem plausible: Others have proposed receipts [Kotla et al. 2007; Popa et al. 2011; Shah et al. 2007; Yumerefendi and Chase 2007], and the proposed LINK and UNLINK calls have correlates on Unix file systems, suggesting utility beyond Teapot.

This discussion illustrates that clients can use an SSP-supplied key-value map as a black box to recover most of Depot's properties. To recover all of them, the SSP needs to be incrementally augmented not to delete prematurely.

10. RELATED WORK

We organize prior work in terms of trade-offs between availability and fault tolerance.

Restricted fault tolerance, high availability. A number of systems provide high availability but do not tolerate arbitrary faults. For example, key-value stores in clouds [Chang et al. 2006; Cooper et al. 2008; DeCandia et al. 2007] take a pragmatic approach, using system structure and relaxed semantics to provide high availability. Also, systems like

Bayou [Terry et al. 1995], Ficus [Reiher et al. 1994], PRACTI [Belaramani et al. 2006], and Cimbiosys [Ramasubramanian et al. 2009] can get high availability by replicating all data to all nodes. Unlike Depot, none of these systems tolerates arbitrary failures.

Medium fault tolerance, medium availability. Another class of systems provides safety even when only a subset (for example, 2/3 of the nodes) is correct. However, the price for this increased fault tolerance compared to the prior category is decreased liveness and availability: to complete, an operation must reach a quorum of nodes. Such systems include Byzantine Fault Tolerant (BFT) replicated state machines (see Castro and Liskov [2002], Clement et al. [2009], Guerraoui et al. [2010], and Hendricks et al. [2007]) and Byzantine Quorums [Malkhi and Reiter 1998]. Note that researchers are keenly interested in reducing trust: compared to classic BFT systems, the more recent A2M [Chun et al. 2007], TrInc [Levin et al. 2009], and BFT2F [Li and Mazières 2007] all tolerate more failures, the former two by assuming trusted hardware and the latter by weakening guarantees. However, unlike Depot, these systems still have fault thresholds, and none works disconnectedly. PeerReview [Haeberlen et al. 2007] requires a quorum of witnesses with complete information (hindering liveness), one of which must be correct (a trust requirement that Depot does not have).

High fault tolerance, low availability. In fork-based systems, such as SUNDR [Li et al. 2004] and FAUST [Cachin et al. 2009], the server is totally untrusted, yet even under faults provides a safety guarantee: fork-linearizability, fork-sequential consistency, etc. [Oprea and Reiter 2006]. However, these systems provide reduced liveness and availability compared to Depot. First, in benign runs, their admittedly stronger semantics (versus Depot's causal consistency during such runs) means that they cannot be available during a network partition or server failure. Second, after a fork, nodes are "stranded" and cannot talk to each other, effectively stopping the system. A related strand of work focuses on accountability and auditing (see Kotla et al. [2007], Shah et al. [2007], Yumerefendi and Chase [2007], and Popa et al. [2011]), providing proofs to participants if other participants misbehave. All of these systems *detect* misbehavior, whereas our aim is to *tolerate* and *recover* from it, which we view as a requirement for availability.

Systems with similar motivations. Venus [Shraer et al. 2010] allows clients not to trust a cloud storage service. While Venus provides consistency semantics stronger than Depot's (causal consistency for pending operations, linearizability for completed operations (roughly)), it makes stronger assumptions than Depot. Specifically, Venus relies on an untrusted verifier in the cloud, assumes that a core set of clients does not permanently go offline, and does not handle faulty clients, such as clients that split history. SPORC [Feldman et al. 2010] is designed for clients to use a single untrusted server to order their operations on a single shared document and provides causal consistency for pending operations (and stronger for committed operations). Unlike Depot, SPORC does not consider faulty clients, allow clients to talk to any server, or support arbitrary failover patterns. Furthermore, SPORC is not designed for environments with large numbers of updates, as it does not permit garbage collection of updates; SPORC requires that all the updates be preserved to be able to detect and recover from forks. However, SPORC provides innate support for confidentiality and access control, whereas Depot layers those on top of the core mechanism.

A number of other systems have sought to minimize trust for safety and liveness. However, they have not given a correctness guarantee under arbitrary faults. For example, Zeno [Singh et al. 2009] does not operate with maximum liveness or minimal trust assumptions: it assumes $f + 1$ available servers per partition, where f is the

number of faulty servers. TimeWeave [Maniatis 2003] ensures that correct nodes can blame any malactivity on culprit nodes. However, unlike Depot, TimeWeave does not *detect* or *repair* forks. S2D2 [Kang 2004] tracks dependences using tamper-evident history summaries, which are similar to Depot's history hashes. However, unlike Depot, S2D2 does not *detect* forks. And neither S2D2 nor TimeWeave targets cloud storage (which requires addressing staleness, durability, and recoverability). Other systems target scenarios similar to cloud storage but do not protect consistency [Goh et al. 2003; Kallahalla et al. 2003; Strunk et al. 2000].

Some systems have, like Depot, been designed to resist large-scale correlated failures. Glacier [Haeberlen et al. 2005] can tolerate a high threshold, but still no more than this threshold, of faulty nodes, and it stores only immutable objects. OceanStore [Kubiatowicz et al. 2000] is designed to minimize trust for durability but does not tolerate nodes that fail perniciously.

Distributed revision control. Distributed repositories like Git[9], Mercurial[10], and Pastwatch [Yip et al. 2006] have a data model similar to Depot's and could be augmented to resist faulty nodes (e.g., forcing clients to sign updates in Git would prevent servers from undetectably altering history). However, all of these systems are geared toward replicating a source-code repository. Our context brings concerns that these systems do not address, including how to avoid clients' storing all data, how to perform update exchange in this scenario, how to provide freshness, how to evict faulty nodes, how to garbage collect, etc.

11. CONCLUSION

Depot began with an attempt to explore a radical point in the design space for cloud storage: *trust no one*. Ultimately we fell short of that goal: unless all nodes store a full copy of the data, then nodes must rely on one another for durability and availability. Nonetheless, we believe that Depot significantly expands the boundary of the possible by demonstrating how to build a storage system that eliminates trust assumptions for safety and minimizes trust assumptions for liveness.

APPENDIX

A. FORK-JOIN-CAUSAL CONSISTENCY

We express *Fork Join Causal (FJC) consistency* semantics in terms of a set of conditions that must hold for the *observer graph* that we associate with each execution of a system. The observer graph of an execution captures how information flows during the execution: the graph's vertices represent the read and write operations executed by the nodes, and the edges encode dependencies among these operations. The graph is not an actual data structure that our protocol maintains, but it is useful for presentation purposes.

Definition A.1. An *observer graph* is an *execution* and an *edge assignment*.

Definition A.2. An *execution* is a set of read and write vertices, with one vertex for each read or write operation by any node.

(1) *Write vertices* are tuples of the form (n, s, oId, val), where n is the node issuing the write operation, s is a per-node sequence number that monotonically increases with every operation issued by n, oId is the identifier of the object being written, and val is the value written to object oId.

[9] `http://git-scm.com/`
[10] `http://mercurial.selenic.com/`

(2) *Read vertices* are tuples of the form (n, s, oId, wl) where n, oId, and s define the node issuing the read, the object read, and the sequence number of the operation and where wl denotes the list of write vertices whose values are returned by the read. We say a read r *reads from* a write w if $r.wl$ includes w.

Definition A.3. An *edge assignment* for an execution is a set of directed edges connecting vertices of an execution.

An edge assignment is an abstract representation of the data flow in an execution. Notice that the definition does not specify how the edge assignment is produced.

A given consistency semantic is defined by a *consistency check* that determines the set of allowed observer graphs. And showing that an *execution is consistent* under some semantics requires showing that an oracle can produce an edge assignment, and hence an observer graph, that passes the consistency check. On the other hand, showing that a *system enforces some consistency semantics* requires presenting an algorithm that, for every possible execution of the system, constructs an edge assignment that passes the consistency check.

Definition A.4. A *consistency check* for a consistency semantics C is a set of conditions that an observer graph must satisfy to be called consistent with respect to C.

Definition A.5. An execution α is C-consistent iff there exists an edge assignment for α such that the resulting observer graph satisfies C's consistency checks.

A final bit of housekeeping.

Definition A.6. We say that vertex u *precedes* vertex v in observer graph G (denoted as $u \prec_G v$) if there is a directed path from u to v in G. By extension, we say that the operation corresponding to u precedes the one corresponding to v. If $u \prec_G v$, then v *depends on* u. If $u \not\prec_G v$ and $v \not\prec_G u$, then we say that u and v are concurrent.

Definition A.7. An operation u is said to be *observed* by a correct node p in G if either p executes u or if p executes an operation v such that $u \prec_G v$.

We now define the set of executions admitted by FJC consistency semantics in terms of its consistency checks.

Fork-Join-Causal Consistency. An execution α is said to be *Fork-Join-Causal (FJC) consistent* if there exists an edge assignment for α that produces an observer graph G that satisfies the following consistency check.

(1) *Serial ordering at each correct node.* The ordering of operations by any correct node is reflected in the observer graph. Specifically, if p is a correct node and v and v' are vertices corresponding to operations by p, then $v.s < v'.s \Leftrightarrow v \prec_G v'$.
(2) *Reads by correct nodes return the latest preceding concurrent writes.* For any read operation $r = (p, s, oId, wl)$ issued by a correct node p, and writes w and w' to object oId, the following condition holds.

$$w \in wl \Leftrightarrow w \prec_G r \wedge \nexists w' : w \prec_G w' \prec_G r$$

Comparison with fork-causal consistency. Fork-Causal Consistency (FCC) enforces the following three conditions:

(1) Serial ordering at each correct node;
(2) Reads by correct nodes return the latest preceding concurrent writes;
(3) Correct nodes observe that every node issues totally ordered writes.

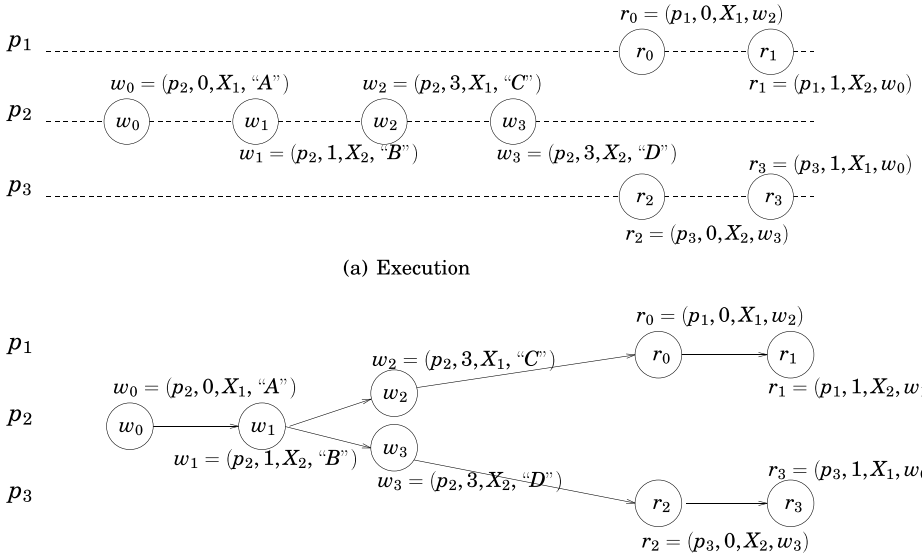

Fig. 14. (a) An execution with a faulty node p_2 that issues two writes with $s = 3$ and (b) an observer graph that is FJC and FCC consistent. There is no causally consistent observer graph because w_0, w_1, w_2, and w_3 are not serially ordered according to any possible history of node p_2. The observer graph is both FJC and FCC consistent because FJC and FCC do not require total ordering of p_2's operations.

The first two conditions are identical to those required by FJC. The third condition trivially holds for writes issued by correct nodes, since, by the first condition, such writes are totally ordered. The difference between FJC and FCC is in the degree to which writes by incorrect nodes are admissible. Consider a faulty node p that issues writes w_0, w_1, and w_2 such that w_0 precedes both w_1 and w_2, but neither $w_1 \prec_G w_2$ nor $w_2 \prec_G w_1$. By its third condition, FCC prevents any correct node from observing both w_1 and w_2. Hence, under FCC, once two correct nodes have observed w_1 and w_2 respectively, they become partitioned from each other. Under FJC, in contrast, this restriction does not exist; there, w_1 and w_2 are treated as concurrent writes, allowing a correct node to observe both.

Comparison with causal consistency. *Causal consistency* enforces conditions that are analogous to those enforced by FJC, but it requires them to hold for operations issued by *all* nodes, not just correct nodes. Specifically, an execution α is said to be *causally consistent* if there exists an edge assignment for α that produces an observer graph G that satisfies the following consistency check.

(1) *Serial ordering at each node.* The ordering of operations by any node is reflected in the observer graph. Specifically, if p is a node and v and v' are vertices corresponding to operations by p, then $v.s < v'.s \Leftrightarrow v \prec_G v'$.
(2) *Reads return the latest preceding concurrent writes.* For any read operation $r = (p, s, oId, wl)$ issued by a node p, and writes w and w' to object oId, the following condition holds.

$$w \in wl \Leftrightarrow w \prec_G r \wedge \nexists w' : w \prec_G w' \prec_G r$$

Figure 14(a) shows an execution that is both FJC and FCC but not causally consistent. In this example, node p_2 is faulty and produces four writes w_0, w_1, w_2, and w_3.

Node p_1 observes w_0, w_1, and w_2 but not w_3, and node p_3 observes w_0, w_1, and w_3 but not w_2. As Figure 14(b) illustrates, we can produce an edge assignment and observer graph that passes all FJC/FCC tests by dispensing with the serial ordering constraint at the faulty node. Conversely, it is impossible to produce an edge assignment to produce an observer graph G' that passes the causal consistency checks.

ACKNOWLEDGMENTS

Insightful comments by Marcos K. Aguilera, Hari Balakrishnan, Michael Freedman, Brad Karp, David Mazières, Arun Seehra, Jessica Wilson, the anonymous OSDI 2010 reviewers, and the anonymous TOCS reviewers improved this article. The Emulab staff was a great help, as always. The Depot code can be downloaded from: http://www.cs.utexas.edu/depot.

REFERENCES

ABU-LIBDEH, H., PRINCEHOUSE, L., AND WEATHERSPOON, H. 2010. RACS: A case for cloud storage diversity. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*.

AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. 1995. Causal memory: Definitions, implementation and programming. *Distrib. Comput. 9*, 1, 37–49.

AMAZON. 2011. Summary of the Amazon EC2 and Amazon RDS service disruption in the US East region. http://aws.amazon.com/message/65648.

AMAZON S3 TEAM. 2008. Amazon S3 Team. Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html.

BECKMANN, C. 2009. Google App Engine: Information regarding 2 July 2009 outage. http://groups.google.com/group/google-appengine/browse_thread/thread/e9237fc7b0aa7df5/ba95ded980c8c179.

BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. 2006. PRACTI replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

BIRRELL, A., LEVIN, R., NEEDHAM, R., AND SCHROEDER, M. 1982. Grapevine: An exercise in distributed computing. *Comm. ACM 25*, 4.

CACHIN, C., SHELAT, A., AND SHRAER, A. 2007. Efficient fork-linearizable access to untrusted shared memory. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing (PODC)*.

CACHIN, C., KEIDAR, I., AND SHRAER, A. 2009. Fail-aware untrusted storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*.

CALORE, M. 2009. Magnolia suffers major data loss, site taken offline. *Wired Mag*.

CASTRO, M. AND LISKOV, B. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4.

CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W., WALLACH, D., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. 2007. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

CIRCLEID. 2009. Survey: Cloud computing 'no hype', but fear of security and control slowing adoption. http://www.circleid.com/posts/20090226_cloud_computing_hype_security/.

CLEMENT, A., KAPRITSOS, M., LEE, S., WANG, Y., ALVISI, L., DAHLIN, M., AND RICHÉ, T. 2009. UpRight cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

CNET NEWS. 2011. Google probing lost Gmail messages, contacts. http://news.cnet.com/8301-1023_3-20037019-93.html.

COOK, B. 2009. Seattle data center fire knocks out Bing Travel, other web sites. http://www.techflash.com/seattle/2009/07/Seattle_data_center_fire_knocks_out_Bing_Travel_other_Web_sites_49876777.html.

COOPER, B., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H., PUZ, N., WEAVER, D., AND YERNENI, R. 2008. PNUTS: Yahoo!'s hosted data serving platform. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

Demmer, M., Du, B., and Brewer, E. 2008. TierStore: A distributed filesystem for challenged networks in developing regions. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.

Feldman, A. J., Zeller, W. P., Freedman, M. J., and Felten, E. W. 2010. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Frigo, M. and Luchangco, V. 1998. Computation-centric memory models. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

Fu, K., Kaashoek, M. F., and Mazières, D. 2002. Fast and secure distributed read-only file system. *ACM Trans. Comput. Syst. 20*, 1, 1–24.

Gilbert, S. and Lynch, N. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 51–59.

Goh, E.-J., Shacham, H., Modadugu, N., and Boneh, D. 2003. SiRiUS: Securing remote untrusted storage. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*.

Gray, J. and Shenoy, P. 2000. Rules of thumb in data engineering. In *Data Engineering*. 3–12.

Guerraoui, R., Knezevic, N., Quema, V., and Vukolic, M. 2010. The next 700 BFT protocols. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

Haeberlen, A., Mislove, A., and Druschel, P. 2005. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Haeberlen, A., Kouznetsov, P., and Druschel, P. 2007. PeerReview: Practical accountability for distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

Hendricks, J., Ganger, G. R., and Reiter, M. K. 2007. Low-overhead Byzantine fault-tolerant storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

Herlihy, M. P. and Wing, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3.

Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., and Fu, K. 2003. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.

Kang, B. 2004. S2D2: A framework for scalable and secure optimistic replication. Ph.D. thesis, University of California Berkeley.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*.

Kistler, J. and Satyanarayanan, M. 1992. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst. 10*, 1, 3–5.

Kotla, R., Alvisi, L., and Dahlin, M. 2007. SafeStore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. 2000. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*, 9, 690–691.

Lamport, L., Shostak, R., and Pease, M. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3, 382–401.

Levin, D., Douceur, J. R., Lorch, J. R., and Moscibroda, T. 2009. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Li, J. and Mazières, D. 2007. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Li, J., Krohn, M., Mazières, D., and Shasha, D. 2004. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Malkhi, D. and Reiter, M. 1998. Byzantine quorum systems. *Distrib. Comput. 11*, 4, 203–213.

MANIATIS, P. 2003. Historic integrity in distributed systems. Ph.D. thesis, Stanford.

MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. 1999. Separating key management from file system security. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

MILLER, R. 2009. FBI siezes servers at Dallas data center.
http://www.datacenterknowledge.com/archives/2009/04/03/fbi-seizes-servers-at-dallas-data-center/.

NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. 2006. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

NEWS, C. 2002. Victims of lost files out of luck.
http://news.cnet.com/Victims-of-lost-files-out-of-luck/2100-1023_3-887849.html.

NIGHTINGALE, E., VEERARAGHAVAN, K., CHEN, P., AND FLINN, J. 2008. Rethink the sync. *ACM Trans. Comput. Syst. 26*, 3.

OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. 2003. Why do Internet services fail, and what can be done about it? In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*.

OPREA, A. AND REITER, M. 2006. On consistency of encrypted files. In *Proceedings of the International Symposium on Distributed Computing (DISC)*.

PARKER, D. S., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., KISER, S., EDWARDS, D., AND KLINE, C. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Engin. 9*, 3, 240–247.

PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. 1997. Flexible update propagation for weakly consistent replication. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

PINHEIRO, E., WEBER, W., AND BARROSO, L. 2007. Failure trends in a large disk drive population. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.

POPA, R. A., LORCH, J. R., MOLNAR, D., WANG, H. J., AND ZHUANG, L. 2011. Enabling security in cloud storage SLAs with CloudProof. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

PRABHAKARAN, V., BAIRAVASUNDARAM, L., AGRAWAL, N., GUNAWI, H., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. 2005. IRON file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

RAMASUBRAMANIAN, V., RODEHEFFER, T., TERRY, D. B., WALRAED-SULLIVAN, M., WOBBER, T., MARSHALL, C. C., AND VAHDAT, A. 2009. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

REIHER, P., HEIDEMANN, J., RATNER, D., SKINNER, G., AND POPEK, G. 1994. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer Technical Conference*.

S3 APP CATALOG. 2011. AWS forum: Customer app catalog.
http://developer.amazonwebservices.com/connect/kbcategory.jspa?category ID=66.

SCHIPER, A., BIRMAN, K., AND STEPHENSON, P. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9*, 3.

SHAH, M., BAKER, M., MOGUL, J., AND SWAMINATHAN, R. 2007. Auditing to keep online storage services honest. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*.

SHRAER, A., CACHIN, C., CIDON, A., KEIDAR, I., MICHALEVSKY, Y., AND SHAKET, D. 2010. Venus: Verification for untrusted cloud storage. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*.

SINGH, A., FONSECA, P., KUZNETSOV, P., RODRIGUES, R., AND MANIATIS, P. 2009. Zeno: Eventually consistent Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

STRUNK, J., GOODSON, G., SCHEINHOLTZ, M., SOULES, C., AND GANGER, G. 2000. Self-Securing storage: Protecting data in compromised systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M., THEIMER, M., AND WELCH, B. W. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*.

TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

US. 2005. US Secret Service report on insider attacks.
http://www.sei.cmu.edu/about/press/insider-2005.html.

VOGELS, W. 2006. Life is not a state-machine: The long road from research to production. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing (PODC)*.

WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. 2002. An integrated experimental environment for distributed systems and networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

WOBBER, T., RODEHEFFER, T. L., AND TERRY, D. B. 2010. Policy-Based access control for weakly consistent replication. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

YAHOO. 2011. Yahoo's offloading of delicious a reminder of cloud risks.
http://www.infoworld.com/t/cloud-computing/
yahoos-offloading-delicious-reminder-cloud-risks-735.

YANG, J., SAR, C., AND ENGLER, D. 2006. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

YIP, A., CHEN, B., AND MORRIS, R. 2006. Pastwatch: A distributed version control system. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

YUMEREFENDI, A. AND CHASE, J. 2007. Strong accountability for network storage. *ACM Trans. Storage 3*, 3.

Received December 2010; revised September 2011; accepted October 2011