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, fuss@cs.utexas.edu

Abstract

The paper 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.

1 Introduction

This paper describes the design, implementation, and evaluation of Depot, a cloud storage system in the spirit of S3 [1], Azure [4], and Google Storage [3] but with a crucial difference: Depot clients do not have to *trust*, that is *assume*, that Depot servers operate correctly.

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 software bugs [9], correlated manufacturing defects [57], misconfigured servers and operator error [53], malicious insiders [68], bankruptcy [5], undiagnosed problems [14], Acts of God (e.g., fires [20]) and Man [50]. 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 [18].

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 temporary [8] and permanent [5, 14] cloud failures).

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 [15, 19, 30] 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 [12, 13, 43, 44] 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 [12, 13, 43, 44], 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 [11, 23, 37, 61, 67] 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 [7, 40, 56]. 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 implements a layered architecture that 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.[1] 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.

[1] We are not explicitly addressing confidentiality and privacy, but, as discussed in §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. 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 *and* 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, they carry non-negligible risks. First, they are opaque (by nature). Second, they are complex distributed systems. Indeed, coping with known hardware failure modes in *local* file systems is difficult [59]; in 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" [69]. 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 [52, 72] may lose recent updates, leading to forks in history. Or, a network failure might delay propagation

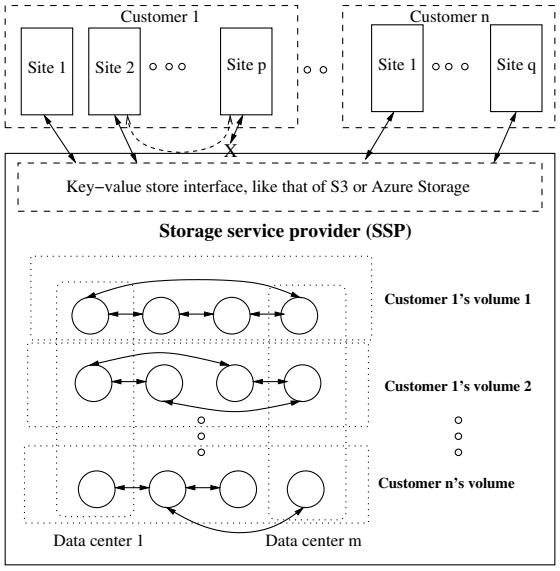


FIG. 1—Architecture of Depot. The arrows between servers indicate replication and exchange.

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 above events seem unlikely. Is tolerating them worth the cost? One of our purposes in this paper is to report for the first time what that cost is. Whether to "purchase" 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. In our target scenario, the servers are operated by a storage service provider (SSP) that is distinct from the data owner that operates the clients.[2] Keys and values are arbitrary strings, with overhead engineered to be low when values are at least a few KB. A Depot client exposes an interface of GET and PUT to its application users.

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 [22, 36, 51], and we leave the assignment of keys to volumes to the layers above Depot.

The servers for each volume may be geographically distributed, a client can access any server, and servers replicate updates using any topology (chain, mesh, star, etc.). As in Dynamo [22], to maximize availability, Depot does not require overlapping read and write quorums. In fact, as the dotted lines suggest, Depot can even function under complete server unavailability: the protocol permits clients to communicate directly with each other. If the SSP later recovers, clients can continue using the SSP (after sending the missed updates to the servers). This raises a question: why have the SSP at all? We point to the usual benefits of cloud services: cost, scalability, geographic replication, and management.

We use the term *node* to mean either a client or a server. Clients and servers run the same basic Depot protocol, though they are configured differently.

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* (§4–§5.2) and *bounded staleness* (§5.4): Once a write occurs, the update should be visible to reads "soon". Consistency and staleness properties limit the extent to which the storage system can reorder, delay, or omit making updates visible to reads.
- *Availability and durability* (§5.3): 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 permanently lose data.
- *Integrity and authorization* (§5.5): Only clients authorized to update an object should be able to create valid updates that affect reads on that object.
- *Data recovery* (§5.6): 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* (§5.7): 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.

[2] Because Depot does not require nodes to trust each other, different data centers in Figure 1 could be operated by different SSPs. Doing so might reduce the risk of correlated failures across replicas [6, 38]. For simplicity, we describe and evaluate only single-SSP configurations.

Depot provides the above properties with a layered approach. Its core protocol (§4) addresses *consistency*. Specifically, the protocol enforces Fork-Join-Causal consistency (FJC), which is the same as causal consistency [7, 40, 56] in benign runs. This protocol is the essential building block for the other properties listed above. In §5, 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 [1, 4], 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 above list of issues is exhaustive. For example, it may be useful to audit storage service providers with black box tests to verify that they are storing data as promised [38, 62], 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 & 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 [2], 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 [56], Coda [37], and others—is handling concurrent writes to the same object. Such *conflicts* are unfortunate but unavoidable: they are provably the price of high availability [26].

Many approaches to resolving conflicting updates have been proposed [37, 61, 67], 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 re-executing transactions [67]), 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.[3]

This approach is similar to that of S3's replication substrate, Dynamo [22], 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]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.

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 [41]) 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, above we 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, to ensure the liveness of garbage collection, 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 [15].

4 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 [21]. However, Depot has two challenges. First, it aims for maximum availability, which fundamentally conflicts with the strictest orderings [26]. 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 above, this basic protocol is run by both clients and servers. This symmetry not only simplifies the design but also provides flexibility. For example, if

servers are unreachable, clients can share data directly. For simplicity, the description below does not distinguish between clients and servers.

4.1 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 [10, 56]; 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* [56]. 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 [40]. The remaining fields, *dVV* and *H(history)*, and the writer's signature, σ_{nodeID}, defend against faults and are discussed in subsections 4.2 and 4.3.

Each node maintains two local data structures: a *log* of updates it has seen and a *checkpoint* reflecting the current state of the system. For efficiency, Depot separates data from metadata [10], so the log and checkpoint contain collision-resistant hashes of values. If a node knows the hash of a value, it can fetch the full value from another node and store the full value in its checkpoint. 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 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 [55]. 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.

4.2 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. Note that while *dVV* logically represents a full version vector, 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.

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 [43]. Such behavior certainly damages consistency. However, the mechanisms above limit that damage, as we now illustrate with an example. Then, in subsection 4.3 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 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 above 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 $N2$ 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 [56] and PRACTI [10] (update exchange), SUNDR [43] (signed version vectors), and BFT2F [44] (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 a technical report [45]. 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 [43]. 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.

4.3 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 (§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 binary search on the updates included in the nodes' version vectors to identify the latest version vector, VV_c, encompassing a common history. 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 pre-fork 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 post-fork entries, whose indices consist of the index before the fork augmented with the history hash of the respective first update after the fork. Each of these entries initially holds the logical clock of the first update after the fork (e.g., of $u_{1@M}$ and $u'_{1@M}$); 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 $M \parallel H(u_{fork_1}) \parallel H(u_{fork_2}) \parallel \ldots \parallel H(u_{fork_i})$ [56].

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 above) 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 above.

Bounding forks The overhead of this coping strategy is the space, bandwidth, and computation needed for fork detection and larger version vectors. Depot bounds

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	1 correct client
Eviction	Safety	Valid eviction	Any subset

FIG. 2—Summary of properties provided by Depot.

the number of forks that faulty nodes can introduce by (1) making nodes "vouch" for updates by a forking node that they had received before learning of the fork and (2) making them promise not to communicate with known forking nodes. We omit the details for space.

5 Properties and guarantees

This section describes how Depot enforces needed properties with minimal trust assumptions. Figure 2 summarizes these properties and lists the required assumptions. Below, we define these properties and describe how Depot provides them. 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.

5.1 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 are observed by any correct node: *Fork-Join-Causal consistency* (FJC). A formal description of FJC appears in our technical report [45]. 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 [25].)

To explain FJC, we contrast it with causal consistency (CC) in fail-stop systems [7, 40, 56]. CC is based on a dependency preservation property that is identical to the one above, 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 (§4.3).

Though FJC is weaker than linearizability, sequential consistency, or causal consistency, it still provides properties that are critical to programmers. First, FJC implies a number of useful *session guarantees* [66] for programs at correct nodes, including monotonic reads, monotonic writes, read-your-writes, and writes-follow-reads. Second, as we describe in the subsections below, 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 [22, 26].

5.2 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 is ensured by the replication protocol (§4), which entangles updates to prevent selective transmission, and by the communication heuristics (§6), which allow a node that is unable to communicate with a server to communicate with any other server or client.

5.3 Availability and durability

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

durability. We begin by noting that the following strong availability properties follow from the protocol in §4 and the communication heuristics (§6):

- *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 they can communicate as per our model in §3.3.
- *Write propagation.* If a correct node issues a write, eventually all correct nodes observe that write, assuming 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 my data"—manifests as a liveness property):

- *Read availability.* If during a sufficiently long synchronous interval any reachable and correct node has an object's value, then a read by a correct node will succeed.
- *Durability.* If any correct *hoarding node*, as defined below, 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 (§5.6). 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. 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 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 [5, 14] or become unavailable [8].

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.

Thus, 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. Finally, 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. Thus, in the worst case Depot can guarantee only that an object value not stored locally 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 (§5.6) until the read completes with either v or a newer version.

5.4 Bounded staleness

A client expects that soon after it updates an object, other clients that read the object see the update. The following guarantee codifies this expectation:

- *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 $t_0 + 2T_{ann} + T_{prop} + \Delta$, 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; both are typically a few tens of seconds.

Depot uses FJC consistency to guarantee that a client always either knows it has seen all recent updates or *suspects* it has not. Every T_{ann} seconds, each client updates a per-client *beacon object* [43] 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 $C1$.

When $C2$ *suspects* that it has missed updates from $C1$, it switches to receiving updates from a different server. If that does not resolve the problem, $C2$ tries to contact $C1$ directly to fetch any missed updates and the updates on which those missed updates depend.

Applications use the above 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 above. Alternatively, an application that prefers to trade worse availability for better consistency [26] 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.

Bounded staleness v. 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.

5.5 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 [48, 71] is future work. We speculate that FJC will make it relatively easy to ensure a sensible ordering of policy updates and access control decisions [24, 71].

5.6 Data recovery

Even if a storage system retains a consistent, fresh view of the data written to it, data owners care about end-to-end reliability, and the applications and users above the storage system pose a significant risk. For example, many of the failures listed in the introduction may corrupt or destroy data. 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 non-laddered versions by *unanimous consent of clients*. 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 [56]). 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 above approach provides the same read availability and durability guarantees for backup versions as for the current version (§5.3). However, a faulty client can delay garbage collection. 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 (§3.3).

5.7 Evicting faulty nodes

Depot evicts nodes that provably deviate from the protocol (e.g., by issuing forking writes) and ensures:

- *Valid eviction.* No correct node is ever evicted.

For space, we discuss eviction only at a high level; details are in our technical report [45]. We use proofs of misbehavior (POMs): because nodes' updates are signed, many misbehaviors are provable as such. For example, when a node N observes forking writes from a faulty node M, it creates a POM and slots the POM into the update log, ensuring that the POM will propagate. Note that eviction occurs only if there is a true proof of misbehavior. If a faulty node is merely unresponsive, that is handled exactly as SLA violations are today.

6 Implementation

Our prototype is implemented in Java. It keeps every version written so does not implement laddered backups or garbage collection (§5.6). It 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.[4]

Implementation of GET & PUT. Depot clients expose a PUT and GET API and implement these calls over the log exchange protocol (§4). Recall that Depot separates data from metadata and that an *update* is only the metadata. Each client node chooses a (usually nearby) *primary server* and fetches updates via background gossip.

On a PUT, a client first locally stores the update and value. As an optimization, rather than initiate the log exchange protocol, a client just sends the update and value of each PUT directly to its primary server. If the update passes all consistency checks and 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. Similarly, servers send updates and bodies to each other "out of band" as they are received; if two servers detect that they are out of sync, they fall back on log exchange.

On a GET, a client sends the requested lookup key, k, to its primary server along with a *staleness hint*. The staleness hint 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 staleness hint contains one element. If the staleness hint 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 to the application, completing the GET. If the server rejects the staleness hint 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 (§4.2), 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.

[4] 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 [52]. 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.

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 14.8 ms to 27.7 ms.	§7.1
Depot's main resource overheads are client-side storage and client- and server-side CPU use.	§7.1
Depot imposes little additional cost for read-mostly workloads. For example, Depot's weighted dollar cost of 10KB GETs and PUTs are 2% and 56% higher than the baseline.	§7.2
When failures occur, Depot continues operating correctly, with little impact on latency or resource consumption.	§7.3

FIG. 3—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. 4—Baseline variants whose costs we compare to Depot's.

7 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 (§7.1). We then convert the resource overheads into a common currency [29] using a cost model loosely based on the prices charged by today's storage and compute services (§7.2). We then move from "stick" to "carrot", illustrating Depot's end-to-end guarantees under faults (§7.3). Figure 3 summarizes our results.

Method and environment Most of our experiments compare our Depot implementation to a set of *baseline* storage systems, described in Figure 4. All of them repli-

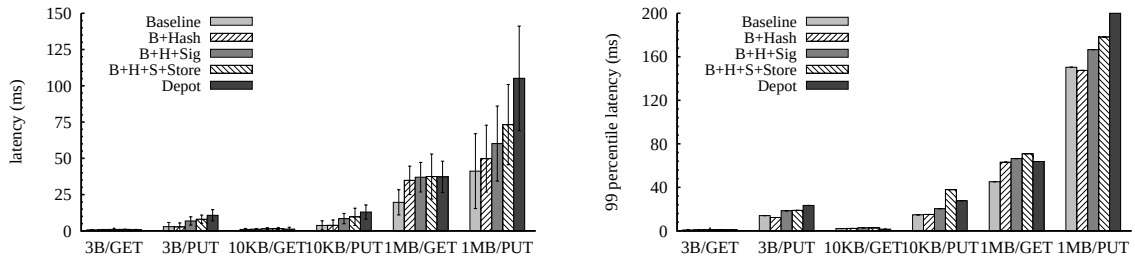


FIG. 5—Latencies ((a) mean and standard deviation and (b) 99th percentile) for GETs and PUTs for various object sizes in Depot and the four baseline variants. For small- and medium-sized requests, Depot introduces negligible GET latency and sizeable latency on PUTs, the extra overhead coming from signing, synchronously storing a local copy, and Depot's additional checks.

cate key-value pairs to a set of servers, using version vectors to detect precedence, but omit 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. Servers gossip with each other once per second; a client gossips with its primary server every 5 seconds. We experiment with a slightly older implementation that runs without receipts (§5.3) and beaconing (§5.4). Since receipts require signature checks, our 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 non-overlapping 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, 10 KB, and 1 MB) 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 [70]. 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, 8 GB of RAM, two 7200RPM local disks, and one Gigabit Ethernet port.

7.1 Overhead of Depot

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 5 depicts the results. For the GET runs, the difference in means between Baseline and B+Hash are 0.0, 0.2, and 15.2 ms for 3B, 10KB, and 1MB, respectively, which are explained by our measurements [45] of mean SHA-256 latencies in the cryptographic library that Depot uses: 0.1, 0.2, and 15.7 ms for those object sizes. Similarly, the means of RSA-Verify operations explain the difference between B+Hash and B+H+Sign for 3B and 10KB, but not for 1MB; we are still investigating that latter case. Depot's GET latency is lower than that of the strongest two baselines because Depot clients verify signatures in the background, whereas the baselines do so on the critical path. Note that 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 3.8 ms, 3.9 ms, 8.5 ms, 9.7 ms, 13.0 ms as we step through the systems.

We can explain the observed Depot PUT latency with a model based on measurements of the main steps in the protocol [45]. For example, for 10KB PUTs, the client hashes the value (mean measured time: 0.2 ms), hashes history ($\approx$ 0.1 ms), signs the update (4.2 ms), stores the body (2.6 ms, with the DB cache enabled), stores the update ($\approx$ 1.5 ms), and transfers the update and body over the 1 Gbps network ($\approx$ 0.1 ms); the server verifies the signature (0.3 ms), hashes the value (0.2 ms), hashes history ($\approx$ 0.1 ms), and stores the body (2.6 ms) and update ($\approx$ 1.5 ms). The sum of the means (13.4 ms) is close to the observed latency (13.0 ms). The model is similarly

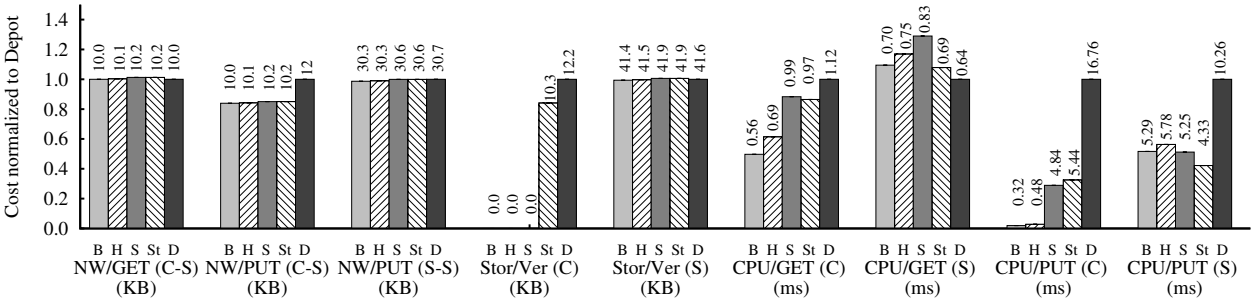


FIG. 6—Per-request average resource use of Baseline (B), B+Hash (H), B+H+Sig (S), B+H+S+Store (St), and Depot (D) in the 100/0 (GET) and 0/100 (PUT) workloads with 10KB objects. The bar heights represent resource use normalized to Depot. The labels indicate the absolute per-request averages. (C) and (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.

accurate for the 3B experiments but off by 20% for 1MB; we hypothesize that the divergence owes to queues that build in front of BDB during periodic log exchange.

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 6 depicts the average use of various resources in the experiments above for 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.

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 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 (and/or larger objects), this CPU overhead is lower, as shown by the measurements summarized immediately below.

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 measured above, but a well-tuned version ought to see sequential disk bandwidth with the bottleneck being signature checks (0.3 ms per core).

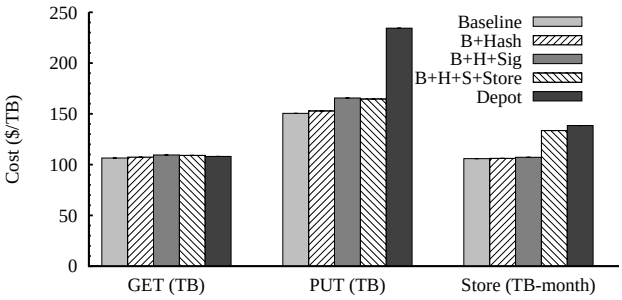


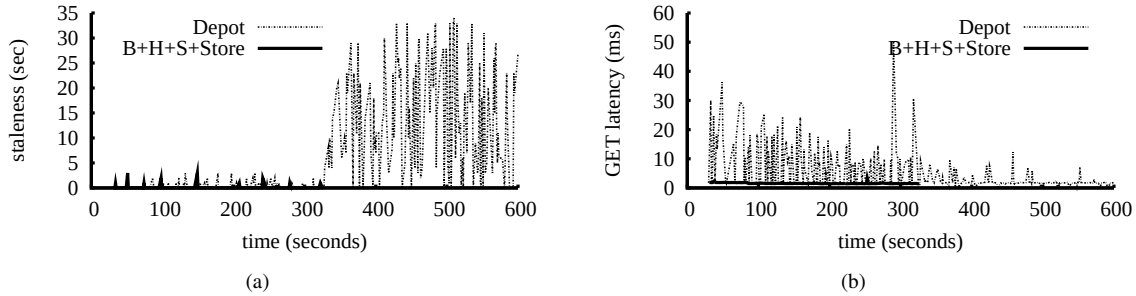
FIG. 7—Dollar cost to GET 1TB of data, PUT 1TB of data, or store 1TB of data for 1 month. Each object has a small key and a 10KB value. 1TB of PUTs or GETs corresponds to 10^8 operations, and 1TB of storage corresponds to 10^8 objects.

7.2 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 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

Figure 7 shows the overheads from Figure 6 weighted by these costs. Depot's overheads are modest for read-mostly workloads. Depot's GET costs are only slightly higher than Baseline's: $108.10 v. $106.50 for 10^8 GET operations on 10KB objects. However, Depot's PUT costs are over 50% higher: $234.40 v. $150.50 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: $138.50 v. $105.50 to store 10^8 10KB objects for a month. Most of the extra



(a) (b)

FIG. 8—The effect of total server failure ($t = 300$) on (a) staleness and (b) latency. The workload is 50/50 R/W and 10KB objects. For space, we do not depict PUT latency for this experiment. Depot maintains availability through client-to-client transfers whereas the baseline system blocks, and GET latency actually improves (at the expense of staleness).

cost is from storing a copy of each object at the issuing client; the rest is from storing metadata.

7.3 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. 300 seconds into the experiment, we stop all servers. By post-processing 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.

Figure 8(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. Staleness increases noticeably because (1) disseminating updates takes more network hops and (2) the lower gossip frequency increases the delay between hops.

Figure 8(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 §7.1 because each object is often updated (because there are few objects in the workload), so the optimization described in §6 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. Though not depicted, Depot's PUT latency also 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.

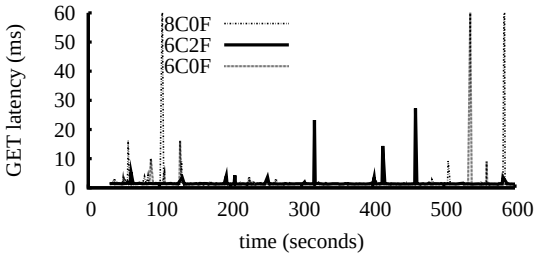


FIG. 9—GET latency seen by a correct client in three runs: 8 correct clients (8C0F), 6 correct clients and 2 faulty clients (6C2F), and 6 correct clients (6C0F). The results for PUT latency are not depicted but are the same: Depot survives forks without affecting client-perceived latency.

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 9 depicts the results for GETs. Forks introduced by faulty clients do not have obvious effect on GET or PUT latency; note that the spikes in GET latency prior to $t = 300$ are unrelated to client failures. We also measured CPU consumption and found no interesting differences among the intervals before the failures, at the time of the failures, or after the faulty nodes had been evicted.

8 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

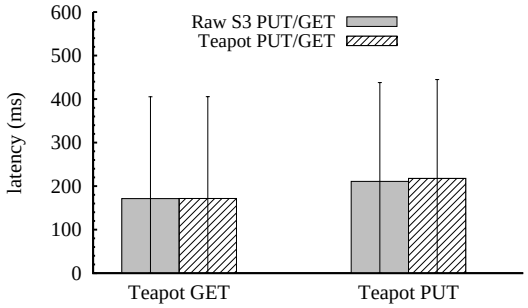


FIG. 10—Average latencies (with standard deviations) perceived by Teapot for GET and PUT operations with 10KB payload when using Amazon S3 for storage.

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 §5. In this section, we give a brief overview of *Teapot*, a variation of Depot that uses legacy SSPs.

Teapot assumes an API like that of S3: 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 10, accessing 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 (§7.1).

Discussion Teapot has two key differences from Depot. First, if a client fails in particular ways, Teapot cannot guarantee *valid discard* (§5.6). A client can, for example, issue a PUT, allow the update to be observed by other clients, and then delete the associated value. Second, Teapot servers cannot provide the durability receipts that Depot clients use to avoid depending on insufficiently replicated data (§5.3). 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 allow a correct client to avoid depending 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 keyrange K in one client's bucket (b_c) to be *linked to* another client's bucket ($b_{c'}$), where a key/value pair *linked to* 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* for garbage collecting versions (§5.6). 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 (§5.3). These extensions seem plausible. Others have proposed receipts [38, 58, 62, 74], 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.

9 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 [16, 21, 22] take a pragmatic approach, using system structure and relaxed semantics to provide high availability. Also, systems like Bayou [67], Ficus [61], PRACTI [10], and Cimbiosys [60] 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 [15, 19, 30, 33]) and Byzantine Quorums [46]. Note that researchers are keenly interested in reducing trust: compared to classic BFT systems, the recently proposed A2M [17], TrInc [42], and BFT2F [44] 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 [31] 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 [43] and FAUST [12], the server is totally untrusted, yet even under faults provides a safety guarantee: fork-linearizability, fork-sequential consistency, etc. [54]. However, these systems provide reduced liveness and availability compared to Depot. First, in benign runs, their admittedly stronger semantics 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 [38, 58, 62, 74]), 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 [63] 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 [24] 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. 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 [64] 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 [47] ensures that correct nodes can pass the blame of any mal-activity to culprit nodes, and S2D2 [35] uses tamper-evident history summaries to detect forks. However, unlike Depot, these two systems neither repair forks nor target cloud storage (which requires addressing staleness, durability, and recoverability). Other systems target scenarios similar to cloud storage but do not protect consistency [28, 34, 65].

Some systems have, like Depot, been designed to resist large-scale correlated failures. Glacier [32] can tolerate a high threshold, but still no more than this threshold, of faulty nodes, and it stores only immutable objects. OceanStore [39] is designed to minimize trust for durability but does not tolerate nodes that fail perniciously.

Distributed revision control. Distributed repositories like Git [27], Mercurial [49], and Pastwatch [73] 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.

10 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.

Acknowledgments

Insightful comments by Marcos K. Aguilera, Hari Balakrishnan, Brad Karp, David Mazières, Arun Seehra, Jessica Wilson, the anonymous reviewers, and our shepherd, Michael Freedman, improved this paper. The Emulab staff was a great help, as always. This work was supported by ONR grant N00014-09-10757, AFOSR grant FA9550-10-1-0073, and NSF grant CNS-0720649. The Depot code can be downloaded from:

http://www.cs.utexas.edu/depot

References

[1] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3.

[2] AWS forum: Customer app catalog. http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=66.

[3] Google storage for developers. http://code.google.com/apis/storage/docs/overview.html.

[4] Windows Azure Platform. http://www.microsoft.com/windowsazure/windowsazure.

[5] Victims of lost files out of luck. http://news.cnet.com/Victims-of-lost-files-out-of-luck/2100-1023_3-887849.html, Apr. 2002.

[6] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In *Proc. 1st ACM Symp. on Cloud Comp.*, 2010.

[7] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, 1995.