# Depot: Cloud storage with minimal trust

P. Mahajan, S. Setty, S. Lee, A. Seehra, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish
UTCS TR-10-21
The University of Texas at Austin
fuss@cs.utexas.edu

**Abstract:** We describe the design, implementation, and evaluation of Depot, a cloud storage system that minimizes trust assumptions. Depot assumes less than any prior system about the correct operation of participating hosts—Depot tolerates Byzantine failures, including malicious or buggy behavior, by *any number* of clients or servers—yet provides safety and availability guarantees (on consistency, staleness, durability, and recovery) that are useful. The key to safeguarding safety without sacrificing availability (and vice versa) in this environment is to *join forks*: participants (clients and servers) that observe inconsistent behaviors by other participants can join their forked view into a single view that is consistent with what each individually observed. Our experimental evaluation suggests that the costs of protecting the system are modest. Depot adds a few hundred bytes of metadata to each update and each stored object, and requires hashing and signing each update.

## 1 Introduction

This paper describes the design, implementation, and evaluation of Depot, a cloud storage system in the spirit of S3 [3] and Azure [5]. However, given that to customers a storage service provider (SSP) is a potentially complex black box controlled by another party, it seems prudent to rely on end-to-end checks of well-defined properties rather than to make strong assumptions about the SSP's design, implementation, operation, and status. Depot is therefore designed to tolerate Byzantine failures, including malicious or buggy behaviors by the SSP. More precisely, Depot minimizes trust assumptions among nodes with respect to both safety and availability:

- *Depot **eliminates** trust for safety.* A client needs to trust only itself to ensure correct operation. Depot guarantees that any subset of correct clients sharing data observe 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 availability.* We wish we could say "trust only yourself" for 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 [11] and permanent [6, 23] cloud failures).

**Safety vs. availability vs. trust.** Though prior efforts have *reduced* trust assumptions in storage systems, they have not *minimized* trust with respect to safety, availability, or both. For example, quorum and replicated state machine approaches [9, 24, 28, 29, 32, 41, 53, 54, 85, 95, 99, 100] tolerate failures by a fraction of servers. However, they sacrifice *safety* when faults exceed a threshold and *availability* when too few servers are reachable. Systems like SUNDR [61] and FAUST [20], and other fork-based systems [19, 22, 62, 64, 74] that remain safe without trusting a server minimize trust for safety. However, they compromise availability in two ways. First, if the server is unreachable, clients must block. Second, a fault server can make correct clients' views diverge *permanently*, preventing them from ever observing each other's new updates.

Indeed, it is challenging to guarantee safety and protect availability while minimizing trust assumptions: without *some* assumptions about correct operation, providing even a weak guarantee like eventual consistency 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 provides far stronger semantics.

A client in the Depot storage system is guaranteed to see eventual consistency, bounded staleness, and a slight weakening of causal consistency that we call *Fork-Join-Causal consistency* (FJC). Roughly speaking, FJC means that all nodes eventually see the same updates, that all correct nodes' updates are eventually visible, and that a correct node's updates and their dependencies are always observed in a causal order.



المنارة للاستشارات
www.manaraa.com

**Approach.** Depot is designed around three key ideas.

1. *Local verification:* Depot clients and servers maintain sufficient local state to validate updates for safety [61].

2. *Join forks:* Faulty nodes can *fork* the system's view of history by introducing incompatible updates [61]. Thus, a crucial requirement for availability is that nodes be able to *join forks*: nodes that observe inconsistent behaviors by other nodes must join their forked histories into a single view that is consistent with what each individually observed. Such joining is challenging; as mentioned above, prior systems permanently strand forked nodes on different branches of history.

3. *Unify protection of safety and availability.* The key to Depot's simplicity is in the realization that the forks caused by faulty nodes can be joined by leveraging the same mechanisms used to handle concurrency in systems that remain globally available during partitions or disconnections. Depot introduces mechanisms to detect when a faulty node forks its history and to treat the faulty node's writes on each fork as concurrent writes by two virtual nodes. Thus, rather than inventing exotic new abstractions for dealing with forks by faulty nodes, Depot employs familiar techniques from the literature on disconnected operation [17, 35, 42, 51, 80, 92] to protect both safety and availability.

Although in principle reducing trust assumptions is always desirable [58, 59, 81], in practice, cost matters. We therefore evaluate the costs of providing untrusted storage in our implementation of Depot. We also evaluate a modified Depot client that uses Amazon S3's as an untrusted storage platform. We find costs to be modest. Depot adds a few hundred bytes to each request and a few milliseconds of processing to small requests and a few tens of milliseconds (due to larger overheads to perform secure hashes) on large files.

## 2 Why untrusted storage?

When we say that "servers are untrusted", we do not suggest that they should be implemented or selected less carefully than they are today. Data owners should still try to hire a SSP that follows best practices. Rather, removing trust is about exercising more caution: it means tolerating a larger number of failures by making fewer and weaker assumptions. Thus, under Depot, it is *good* for nodes to operate correctly, but we do not *assume* that they do. Instead, participants can *verify* other nodes and *ensure continued operation* with clean semantics if some nodes fail to act as hoped.

It is often desirable to minimize trust and employ end-to-end correctness checks in any system [58, 59, 81], but we take particular pains to minimize trust assumptions in our cloud storage service for three reasons.

First, from a client's point of view, the SSP is a potentially complex black box controlled by another party, so it seems prudent not to *assume* the correctness of the SSP's internals. While most storage service providers may follow best practices, some may not, and it may be hard to tell the difference (until it is too late). For example, one customer discovered after repeated disk failures that his large ISP reused old disk drives in new servers until they failed [36]. Though this is only an anecdote, it is rooted in the reality of providers' opacity. Furthermore, *any* storage service, well-managed or otherwise, is subject to non-negligible risks: coping with known hardware failure modes in *local* file systems is difficult [78]; in cloud storage, this difficulty can only grow.

Second, replication across servers and locations is not a panacea. As Vogels notes, "[The] absolutely unrealistic assumption [of uncorrelated failures] will come back to haunt you in real life, where failures frequently are correlated, as they are often triggered by external or environmental events" [96]. In the context of cloud services, one must consider software bugs and vulnerabilities [15], correlated manufacturing defects [77], misconfiguration and operator error [73], malicious insiders [94], bankruptcy [6], undiagnosed problems [23], and acts of God [30] and man [70]. Moreover, even when failures *are* uncorrelated, the risk that some objects are unlucky and struck by simultaneous failures rises rapidly as systems grow [72].

Third, from an SSP's point of view, lack of trust may be a significant barrier to the adoption of cloud services, so client-verifiable end-to-end guarantees may help convince customers to accept the approach.

We also minimize trust towards clients. Clients are vulnerable to several of the same types of failures discussed above. Having pushed the envelope on protecting the system against server misbehavior, we do not want a single faulty client to disrupt the operation of the system.

## 3 Architecture and scope

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. Keys and values are arbitrary strings, with overhead engineered to be low when values are at least a few KB.

For scalability, we slice the system into groups of servers, where each group is 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 among nodes are possible [13, 34, 48, 50, 72, 90], and we leave the assignment of keys to volumes to layers above Depot.



للاستشارات المنارة
www.manaraa.com

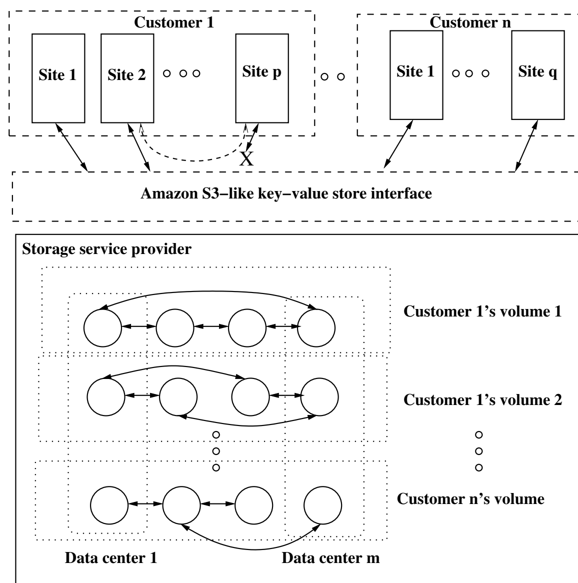



FIG. 1—Architecture of Depot. The arrows between servers indicate replication and exchange.

The servers for each volume may be geographically distributed, a client can access any server, and servers replicate updates using any arbitrary topology (chain, mesh, star, etc.) As in Dynamo [34], to maximize availability Depot does not require overlapping read and write quorum, and—as the dotted lines suggest—Depot can even continue to function during periods of complete server unavailability by having clients communicate directly with one another.

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 have asked, what is the minimum assumption that we need to provide useful guarantees, and what are those guarantees? Below we list the issues we examine. The next two sections then describe the core Depot protocol (§4) and explain how Depot builds on it to provide these properties (§5).

- *Consistency* (§5.1) and *bounded staleness* (§5.3): We want to limit the extent to which the storage system can reorder, delay, or omit updates in a way that is visible to a client's reads. The goal is to provide sufficiently strong and precise guarantees that users and programmers can understand and predict how the system will behave.
- *Availability and durability* (§5.2): Durability means that we want to ensure that a client eventually succeeds in reading an object. Availability means that we want to maximize the fraction of time that a client succeeds in reading or updating an object.
- *Integrity and authorization* (§5.4): Only clients authorized to update an object should be able to create valid updates that affect reads on that object.
- *Recovery* (§5.5): Data owners care about end-to-end reliability. Data integrity, consistency, and durability are not enough when the layers above Depot—faulty client nodes, 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 *recovery* to earlier versions of data.
- *Evicting faulty nodes* (§5.6): If a faulty node provably deviates from the protocol, we wish to evict it from the system so that it will not continue to cause confusion. However, it is vital that we never evict correct nodes.

Note that we explicitly do not attempt to solve the confidentiality/privacy problem within Depot. Instead, like commercial storage systems [3, 5], 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 do not claim that the above list of issues is comprehensive. For example, it may be useful to audit storage service providers with black box tests to verify that they are storing data as promised [52, 86, 87], but we do not examine that issue. Still, we believe that the properties are sufficient to make the resulting system useful.

## 3.2 System and threat model

Before continuing, we briefly describe 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 [57]) ways: they can crash, corrupt data, lose data, process some updates but not others, process messages incorrectly, collude, etc.

Third, we assume an unbounded number of synchronous intervals of sufficient length to allow a pair of timely, connected, and correct nodes to exchange a finite number of messages. This assumption implies that a faulty node cannot forever prevent correct nodes from communicating. However, we make no assumptions about when these synchronous intervals happen.

Fourth, above we used the term *correct node* somewhat loosely. This term refers to a node that neither devi-



المنارة للاستشارات
www.manaraa.com

ates from the protocol nor becomes *permanently* unavailable; a node that crashes and recovers is equivalent to a node that never crashes but that is sometimes slow. Two final technical points related to durability and liveness of garbage collection. First, a node that obeys the protocol for a time but later deviates is never counted as correct; second, we assume that unrecoverable clients are eventually replaced: an administrator needs only the old client's keys and configuration to bring up a new machine [24].

# 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 [31]. However, Depot has two challenges: first, it wants maximum availability, which fundamentally conflicts with the strictest orderings [38] . Second, it wants to continue 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.

Of course, ordering updates and reads is not the only thing that Depot must do. However, it is the essential building block for Depot's other properties. In §5 we define precisely the consistency guarantee that Depot enforces and discuss how Depot provides the other properties listed in §3.

Note that clients and servers run the same basic protocol. 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 discussion in this section does not distinguish between clients and servers.

## 4.1 Basic protocol for update propagation

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 [16, 76], but we describe it here for background and to define terms. Subsections 4.2 and 4.3 describe how Depot defends against faulty nodes.

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

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

Updates are associated with logical times. A node assigns each update an *accept stamp* of the form *logicalClock@nodeID* [76]. A node increments its logical clock on each local write. Also, when a node $N$ receives an update $u$ from another node, $N$ advances its logical clock to exceed $u$'s so that an update's accept stamp exceeds the accept stamp of any update on which it depends [56].

Each node maintains two main local data structures: a *log* of updates it has seen and a *checkpoint* reflecting the current state of the system to support random access reads and writes. Note that Depot separates data from metadata [16], 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. To garbage collect its log, a node creates a second checkpoint corresponding to a past logical time and removes all log entries with accept stamp prior to that time [76].

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 $M$ [75]. To transmit updates from node $M$ to node $N$, $M$ sends to $N$ the updates from its log that $N$ has not seen.

Each node sorts the writes 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. Also, the checkpoint state for any object $o$ is the most recent write to $o$ in the log, so (assuming no faulty nodes) reads from the checkpoint are causally consistent.

**Conflict resolution.** Two updates are *logically concurrent* if neither appears in the other's history. Logically concurrent updates that modify different objects can be readily applied by the local state of the node that receives them. However, if two concurrent updates modify the same object, these updates *conflict*.

Many approaches to resolving conflicting updates have been proposed [51, 80, 92], and Depot does not claim to extend the state of the art on this front. Our prototype implements a simple mechanism that supports a range of application-level conflict resolution policies: a read of key $k$ in Depot returns the *set* of logically most recent updates to $k$. This set includes any update to $k$ that has not been superseded by a logically later update of $k$. Applications may then 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 is guaranteed to appear logically after and thereby supersede the conflicting writes.)

## 4.2 Defending against faulty nodes

There are three fields in an update that defend the protocol against faulty nodes. The first is a *history hash* that encodes the history on which the update depends using



المنارة للاستشارات
www.manaraa.com

a collision-resistant hash that covers the most recent update by each node in the system 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, *dVV* actually contains only the entries that have changed since the last write by $N$ [83]. Third, a node signs its updates with its private key. These signatures ensure that, to be viewed as valid and to be applied to the local state at any correct node, an update on a given object must be signed by a node authorized to update the object.

To defend against faulty nodes, a correct node $N$ uses the update's *history hash* and *dVV* to enforce the following invariant: an update $u$ is accepted only if it is properly signed and $N$ has already accepted all the updates on which $u$ depends. Attempts by a faulty node to fabricate $u$ and pass it as coming from a correct node; reorder or omit updates on which $u$ depends; or include in $u$ incompatible *dVV* and *history hashes*, will all result in $N$ rejecting $u$.

To compromise consistency, a faulty node has one remaining option: make the system violate causal consistency by *forking*, that is, showing different histories to different communication partners [61]. The rest of this section describes how Depot tolerates such attacks.

**Detecting forked histories.** A correct node produces a sequence of updates with monotonically increasing histories captured by each update's *dVV* and *history hash*. In contrast, a faulty node $M$ can *fork* its updates, creating two updates $u_{1@M}$ and $u'_{1@M}$ such that neither write'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, $N2$.

If updates did not include their history hashes, such forking updates might confuse other nodes. To continue the running example, $N1$ could issue some new updates that depend on updates from one of $M$'s forked updates (e.g., $u_{1@M}$) and then send these new updates to $N2$. $N2$ might receive $N1$'s new updates, but not the updates by $M$ on which they depend—because $N2$ already received $u'_{1@M}$, its version vector appears to already include the prior updates. However, if now $N2$ applies just $N1$'s writes to its log and checkpoint, multiple consistency violations may 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 additional information with each update prevents such confusion. In the example, 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 break its connection; the reverse happens if $N2$ tries to send updates to $N1$. As a result, $N1$ and $N2$ will be unable to exchange any updates after the *fork point* introduced by $M$ after $u_{0@M}$.

**Discussion.** If we stopped here, the protocol would enforce *fork causal consistency*, which we define precisely in a technical report [63]. It means that each node sees a causally consistent subset of the system's updates even though the system as a whole is no longer causally consistent. Informally, history has branched, but each node peers backward from its branch to the beginning of time, seeing causal events the entire way.

Though these forks are regrettable, they are impossible to prevent if nodes are allowed to misbehave arbitrarily. More precisely, as proved in [63], fork causal consistency is the strongest consistency guarantee that one can provide in a system in which (a) nodes can misbehave; (b) causal consistency is provided in the absence of misbehavior; and (c) a node can exchange updates with another without needing to involve a third party.

Unfortunately, enforcing this strong consistency would compromise availability: fork causal consistency requires that once two nodes have been forked, they can never observe one another's updates after the fork point [61]. In many environments, this lack of availability is unacceptable. In those cases, it would be far preferable to weaken consistency slightly to ensure an availability property: *correct nodes can always share updates.* We now describe how Depot upholds this property.

## 4.3 Joining forks

To protect availability, nodes must be able to *join* forked branches of the system's history by receiving non-causally-consistent updates by a faulty node and updates by other nodes that depend on them.

At a high level, Depot converts concurrent updates by a single faulty node into concurrent updates by a pair of virtual nodes. Depot then applies well-studied techniques for weakly consistent systems in benign settings [51, 92]. We now fill in the details that underly this approach.

**Tracking forked histories.** A node identifies a fork when it receives two updates issued by the same writer (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 a history that includes the same previous update by that writer (e.g., $u_{0@M}$).

If a node $N2$ receives from node $N1$ an update that is incompatible with the updates it has received, and if neither node has yet identified the fork point, $N1$ and $N2$ perform a binary search on the updates included in the nodes' version vectors to identify the latest version



www.manaraa.com

vector, $VV_{common}$, encompassing a common history. $N1$ then sends its log of updates beginning from $VV_{common}$. At some point, $N2$ receives the update (e.g., $u_{1@M}$) that is incompatible with two updates (e.g., $u_{0@M}$ and $u'_{1@M}$) that $N2$ has already received.

After a node identifies the three updates responsible for a fork, 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., node ID) 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 two 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. These entries initially hold the logical clock of the first updates after the fork (e.g., of $u_{1@M}$ and $u'_{1@M}$), and these values advance as the node receives new updates after the fork point.

A faulty node can be responsible for multiple forks [76], so we must ask whether multiple forks stymie the construction immediately above. The answer is no: this construction (augmenting the prior index with the hash of the update after the latest fork) is an operation that composes. More specifically, after $i$ dependent forks, a virtual node's index in the version vector is well-defined: it is $nodeID + H(u_{fork_1}) + H(u_{fork_2}) + ... + H(u_{fork_i})$.

**Log exchange revisited.** This expanded version vector makes it easy to identify which updates to send to a peer. In the standard protocol, when a node wants to receive updates from another node, it sends its current version vector to the sender so that the sender knows which updates are needed. After a node detects a fork and splits one version vector entry into three, it includes all three entries when asking for updates. If the sender is already aware of the fork, it is already maintaining the same three entries summarizing its state, and as in the standard protocol, the difference between the version vector entries identifies which updates from each fork must be sent. If the sender has received updates from one branch but not the other, it can identify which branch it is on using the history hash and then use the logical time from that branch to identify which updates to send. Finally, if the sender has received updates that belong to neither branch, a new fork point is created as above.

### 4.4 Client access protocol

So far, we have described the update propagation protocol, which is the core of Depot, but we have not described how GETs and PUTs are handled by clients. In this section, we describe the protocol clients use to interact with servers. Unlike the update propagation protocol, which is identical for both clients and servers, clients and servers take different actions in the client access protocol. The client access protocol can be divided into a PUT-protocol and a GET-protocol.

A PUT in Depot involves the following steps. The issuing client generates an appropriate update (as defined in §4.1) and value for the PUT request and stores this value and update in its local store. It then sends the value and update to a server for storage (which is usually a nearby server). On receiving this value and update from a client, the server verifies the update and that the value hash present in the update matches the hash of the received value. If so, the server stores the value and update on its persistent store and sends an acknowledgment to the client. In the background, the server propagates this update and value to the other servers through periodic gossip messages. A client retrieves these new updates during a GET or during background gossiping.

A GET in Depot has two paths: *fast* and *slow*. The fast path is optimized for the scenario when the background gossips have propagated most updates to the client performing the GET. Therefore, in the fast path, we assume that the client has already received and verified all the updates and is missing only the value(s) for the accessed key. The client sends the requested key to a server which responds by sending back the most recent value(s) for the requested key from its local store. The client verifies that the hash(es) of the value(s) received for the requested key matches the value hash(es) present in the most recent update(s) it knows to that key. If successful, these steps constitute the fast path of Depot.

The slow path is taken when a client with a stale update to a key tries to access that key. In this case, the hash(es) of value(s) sent by the server don't match the value hash(es) of the most recent update(s) to the accessed key. On detecting this mismatch, the client initiates a value and update transfer by sending its version vector—to initiate the log-exchange as described in §4.1, and the key that it is interested in accessing—to request the value(s) for this key. The server sends back the new update(s) and the most recent value(s) for the requested key. The client verifies the received updates as described in §4.2 and the value(s) as in the case of fast path.

Depot includes an additional optimization to prevent the server from transferring the value(s) in the fast path when the client has a stale update. When a client issues a GET for a key, it includes a 2-byte compact, but insecure, hash of the most recent value hash(es) for the requested key. The server uses this compact hash to ensure, with high probability, that both the client have the same value(s) for the accessed key. If so, the server proceeds as described in the fast path description earlier. If not, the server sends back a message requesting the client to take the slow path, thereby avoiding the transfer of value(s).

To avoid the slow path, Depot clients periodically retrieve new updates from their preferred servers and, af-



المنارة للاستشارات
www.manaraa.com

| Dimension | Safety/ Liveness | Property | Correct nodes required |
|---|---|---|---|
| Consistency | Safety<br>Safety<br>Liveness | Fork-Join Causal<br>Bounded staleness<br>Eventual consistency | Any subset<br>Any subset<br>Any subset |
| Availability | Liveness<br>Liveness<br>Liveness | Always write<br>Always exchange<br>Read availability / durability | Any subset<br>Any subset<br>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.

ter verifying them, store them in their local store. Note that the verification of received updates is required to ensure consistency. Storing these verified updates (∼285 bytes in our implementation) for future GETs minimizes latency and avoids duplicate transfers. By appropriately configuring their prefetch frequencies for their environment, clients can avoid the slow path.

Alternatively, in environments with high update frequency, clients can configure their system to take the slow path always, as a way of saving bandwidth and CPU cycles, and reducing latency. This configuration avoids the additional latency of failed fast paths, when a slow path is highly likely to be taken, and avoids the bandwidth and CPU cycles to retrieve and verify signatures on all but the most recent updates. Recall that in Depot, each update includes a history hash of prior updates and therefore, a signature on an update can also serve as signature on prior updates by the same client. Omitting all but the most recent signatures saves bandwidth because clients don't need to receive these omitted signatures, which constitute about half of our metadata, and it saves CPU because a client doesn't spend CPU cycles to verify these omitted signatures. While this optimization of omitting all but last signature from each client is useful in general, it is especially useful in this configuration when updates are fetched on demand rather than prefetched during the background gossips.

# 5 Properties and guarantees

This section describes how Depot uses the replication protocol as a building block to enforce useful properties with minimal trust assumptions. Figure 2 summarizes these properties and lists the assumptions required to uphold them.

Below, we define these properties more precisely and describe how Depot provides them. The key idea is that the protocol described in §4 enforces a new consistency semantic called *fork-join causal consistency* (FJC). Given FJC consistency, we can constrain and reason about the order that updates propagate through the system and use those constraints to help enforce the remaining properties.

## 5.1 FJC consistency

Clients expect storage services to provide consistent access to stored data. Depot guarantees *fork-join-causal (FJC) consistency* for all reads and updates to a volume that are observed by any correct node. A more formal description of FJC consistency appears in Appendix A. Here we describe two important aspects of FJC consistency:

- *Eventual consistency.* Any update issued or read by a correct node is eventually observable by all correct nodes. Also, reads of a lookup-key at correct nodes that can observe the same set of updates to that key return the same values. *Observable* simply means that, if an application were to issue a read of the updated object, it would receive a version that is at least as new as the indicated update [37].
- *Dependency preservation.* If update $u_1$ by a correct node depends on an update $u_0$ by any node, then any correct node will observe $u_0$ before it observes $u_1$. This property implies a number of useful *session guarantees* [91] for programs running on correct nodes including monotonic reads, monotonic writes, read-your-writes, and writes-follow-reads.

More broadly, from the point of view of applications and users, FJC consistency appears almost identical to causal consistency. There are two main differences. First, under FJC consistency, a faulty node can issue writes $w$ and $w'$ such that one correct node observes $w$ without first observing $w'$ while another observes $w'$ without first observing $w$. Note, though, that Depot ensures that all such writes eventually become visible to correct nodes: Depot uses fork joining to transform $w$ and $w'$ into causally concurrent writes by two virtual nodes. Second, under FJC consistency, faulty nodes can issue updates whose histories do not include all updates on which they actually depend. For example, a faulty node can read an update $u_c$ from a correct node and then create an update $u_f$ that does not include $u_c$ in its history hash or *dVV*.

**Stronger consistency during benign runs.** Depot guarantees FJC consistency semantics for all runs, even if an arbitrary number of nodes fail in arbitrary ways. During benign runs, Depot ensures causal consistency. Although causal consistency is weaker than the strictest consistency of linearizability [46], we accept this weakening because it allows Depot to remain available to reads and writes during partitions [34, 38].

## 5.2 Availability and durability

Availability (roughly: "I can get to my data now") and durability (roughly: "I can eventually get to my data")



المنارة للاستشارات
www.manaraa.com

are vital properties for storage services. There is, unfortunately, a limit to what any storage system can guarantee: 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.

Depot copes with these fundamental limits in two ways. First, the *protocol* minimizes the number of correct nodes required for availability. Second, the system *configuration* increases the likelihood that an object is available and durable across important failure scenarios.

**Protocol.** The replication protocol described in §4 maximizes durability and availability for reads and writes by providing the following guarantees:

- *Always write:* An authorized node can always update any object.
- *Always exchange:* Any subset of correct nodes can exchange any updates they have observed if they can communicate during a sufficiently long synchronous interval.
- *Read availability:* If during a sufficiently long synchronous interval any reachable correct node has an object, then a read by a correct node will succeed.
- *Durability:* If any correct *hoarding node* has an object, then a read of that object will eventually succeed.

The durability property includes the term *hoarding node.* Recall from §4 that Depot separates data from metadata: updates and checkpoints include a *hash* of a key's value but the value itself is sent/stored separately. A *hoarding node* for an object is a node that always stores the object's value; a hoarding node for a *version* of an object stores that version of the object's value as long as that version is valid. In contrast, a *caching node* for an object may discard the object's value at any time and can fetch values that match currently valid hashes from other (hoarding or caching) nodes. Thus, an update is durable once its value reaches a correct node that will not prematurely discard it.

It may not be surprising that operations can succeed if a correct/reachable node has the needed data, but note that when an operation *succeeds* it not only accesses the requested data but also guarantees that the requested data is safe to access under the system's consistency, staleness, and integrity guarantees.

Notice that a hoarding node must atomically process a key's update (metadata) and store the corresponding value. When a hoarding node updates an object, it stores the update and value locally. When a hoarding node requests updates from another node it sets a flag to indicate that values also need to be sent; if the sender does not have the required values, then the hoarding receiver requests the updates from another node.

**Configuration.** Depot's protocol allows us to trade off availability and replication. In principle, we could realize our goal of having each node trust only itself for availability and durability by having all nodes hoard all objects, but in practice such an approach is not appealing for many cloud storage applications.

In the Depot prototype servers for a volume hoard that volume, and each client hoards object versions for updates they write. On a get(), a client first tries to read from any server. If that fails, it tries other servers. Finally, it goes directly to the client that produced the most recent update for the object stored in the local checkpoint.

This configuration allows us to survive diverse failure scenarios including not only the *routine failure* case where some subset of servers or clients fail and lose data but also the *client disaster* or *cloud disaster* case where all clients *or* all servers fail [6, 23] or become unavailable [11]. As an aside, Depot servers do not trust each other either, so a customer can configure replication to span several storage service providers (SSPs) [52] to further safeguard durability and availability without affecting Depot's safety guarantees; client write hoarding can be deactivated in such scenarios.

## 5.3 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 $\Delta$ of a true clock and $c1$ updates an object at time $t_0$, then by no later than $t_0 + 2T_{announce} + T_{prop} + \Delta$, either (1) the update is visible to $c2$ or (2) $c2$ *suspects* that it has missed an update from $c1$.

$T_{announce}$ 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.

We use Depot's FJC consistency to guarantee that a client always either knows it has seen all recent updates or suspects it has not. Every $T_{announce}$ seconds, each Depot client updates a per-client *beacon object* [61] 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 $c1$. Suspicion guarantees that if the servers are failing to propagate $c2$'s updates to $c1$, $c1$ can take action, but false positives are possible. For example, the servers may be operating correctly but $c1$ may have crashed.

When $c2$ *suspects* it has missed updates from $c1$, it switches to a different server. If that fails to resolve the



المنارة للاستشارات
www.manaraa.com

problem, $c2$ attempts to contact $c1$ directly to fetch any missed updates and the updates on which they depend. As noted in §5.2, $c2$ will retry with different servers (and, if necessary clients), until it succeeds in receiving the needed updates. Furthermore, because the replication protocol enforces FJC consistency regardless of the topology over which updates flow, the updates $c2$ receives are always consistent.

**Get staleness semantics v. availability.** Applications have two options if a node *suspects* it is missing updates when the application issues a get(). The get() can return a *warning* that the result might be stale. This option is our default and it provides the guarantee stated at the start of this subsection. Alternatively, if applications prefer to trade worse availability for better consistency [38], they can choose to *block* until the node has succeeded in receiving all recent beacons.

The reader may notice 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? There are three. First, recall from §2 that although we are prepared for bad failures, we hope (and expect) that most of the time we will operate in more benign conditions. When clients, servers, and the network are *not* faulty, clients are assured that they are reading fresh data. Second, when servers or some network paths are faulty, *suspicion* causes clients to fail-over to other communication paths to get the recent updates they need. Third, if a client like $c1$ *is* faulty, this protocol (correctly) warns other clients that some of the faulty clients' updates may be missing.

**Bounded staleness v. FJC.** Bounded staleness and FJC consistency are complementary properties; both are needed in our context. 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 such a thing is possible; recall that we bound staleness by relying on consistency) is not enough. For engineering reasons, our staleness guarantees are typically on the order of tens of seconds; absent consistency guarantees, applications could be confused because there could be significant periods of time when some updates are visible, but related ones are not.

## 5.4 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. Depot enforces authorization and integrity by requiring nodes to sign their updates, and correct nodes treat unauthorized updates as no-ops that do not affect future reads of the update's lookup-key. Note that because updates' histories are entangled, nodes must continue to propagate these updates.

In our prototype, when a volume is created, it is configured to statically associate ranges of lookup-keys with specific nodes' public keys. This simple approach allows specific client nodes to read and write specific subsets of the system's objects and to prevent servers from reading or modifying the objects they store on behalf of clients. More sophisticated approaches to key management [68, 98] are left as future work.

## 5.5 Recovery

Even if a storage system retains a consistent and 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, careless users or administrators [71, 82], buggy applications, malware, and malicious insiders [7, 14] all may corrupt or destroy valuable data. Depot does not attempt to distinguish "good" updates from "bad" ones 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 laddered backup scheme (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).

**Discarding versions by unanimous consent.** Given FJC consistency, implementing laddered backups is straightforward. Initially, servers retain the *update* and full *value* for every update they receive, and clients retain the *update* and *value* for every update 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 [76]). 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 CDL rotates over the clients, and happens every day at midnight. The key to correctness here is (a) a correct client will not sign a CDL that would delete a checkpoint version 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*. This ensures the *valid discard* 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



www.manaraa.com

(§5.2). However, a faulty client can delay garbage collection. If a backup certificate fails to garner unanimous consent of clients, clients notify a system administrator who trouble-shoots the problem or, if all else fails, replaces the faulty client with a new machine. Thus, faulty clients can cause the system to temporarily consume extra storage resources, but assuming clients that prevent garbage collection progress are eventually repaired, these resources are guaranteed to be released eventually.

### 5.6 Evicting faulty nodes

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

- *Valid eviction.* No correct node is ever evicted from the system.

Eviction only occurs 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. Due to space constraints, we omit further discussion of eviction.

## 6 Experimental evaluation

In this section we first say a few words about our prototype, and then we evaluate Depot experimentally. Our principal question is: what is the cost of the guarantees that Depot provides relative to a baseline storage system?

### 6.1 Implementation

We have implemented the Depot key-value store prototype in Java.

The prototype implements the protocol described in §4 and most of the mechanisms described in §5. The Depot prototype does not implement the laddered backup scheme described in §5.5. We have also not implemented the optimization to omit unneeded signatures in Depot as described in §4.4. Additionally, we do not implement any mechanisms to prevent faulty nodes from exhausting the timestamp space by issuing writes with an artificially large accept stamp; a simple defense is for correct nodes to delay receiving any update with a logical clock that exceeds the node's local time in microseconds.

Our prototype uses Berkeley DB for local storage. Each write to Berkeley DB is committed by calling *commit* before returning. Berkeley DB is configured to write data to disk buffers (by calling `fsync`) for every transaction.

In our experiments, servers gossip every second to exchange updates while clients gossip with a chosen primary server every 5 seconds. Note that different clients may have different primary servers. We use 1024 bit RSA keys and SHA-256 hash for all experiments unless noted otherwise.

| System | Description |
|---|---|
| Baseline | Clients send get/put requests to a server and trust the server to operate correctly. Clients don't maintain any local state or perform any check. |
| B+Hash | Clients attach SHA-256 hashes to the values that they store and verify these hashes on gets. |
| B+H+Sig | Clients sign the values that they store and verify these signatures on gets. |
| B+H+S+Store | Same as B+H+Sig, but clients also locally store the values that they write to ensure durability and availability in presence of server failures. |

FIG. 3—Summary of baseline variants. Our evaluation compares the costs of these variants to those of Depot.

### 6.2 Setup and method

Most of our experiments compare our implementation of Depot to a set of *baseline* key-value storage systems that provide increasingly stronger properties. Figure 3 describes these baseline variants. All of them replicate the key-value pairs to a set of servers but omit one or more of Depot's safeguards. We have implemented these baseline variants using the same code base as Depot. Therefore, these comparison systems are not heavily optimized. In particular, our baseline variants attach some metadata ($\sim$ 100 bytes) to each PUT to detect precedence and ensure eventual consistency. This metadata is also logged to the disk. In addition, like in Depot, the baseline variants separate data from metadata, causing writes to two different Berkeley DB tables on every PUT, which is possibly inefficient compared to what a real storage application would do. Thus, we may slightly underestimate Depot's true overheads.

We focus on measuring the "price of distrust" with an emphasis on the network bandwidth consumption, the storage overheads at both clients and servers, the CPU cycles consumed at both clients and servers, and latency of processing requests. Our evaluation converts these resource overheads into a common currency [40]. For this conversion, we use the following cost model, derived from what Amazon's S3 and EC2 charge their customers [1, 2], but we also provide the raw per-resource overheads so that the reader can compute costs under other assumptions:

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

For intuition, note that the following cost about the same (roughly one nano-dollar, or $\frac{\$1}{10^9}$): transmitting 10 bytes between a client and a server, or storing 10 bytes for a month. In comparison, signing one small message takes $\approx$4 ms of CPU and costs about 100 nano-dollars.



المنارة للاستشارات
www.manaraa.com

**Configuration.** We run most experiments on a *local testbed* comprising up to 25 Dell PowerEdge r200 nodes, each with a quad-core Intel Xeon X3220 2.40 GHz processor, 8 GB RAM, two local disks, and one 1 gigabit Ethernet port. The operating system is Fedora Core 2.6.25.14-69. The nodes are part of a local Emulab [97], allowing us to vary the topology and performance of the network connecting them.

Our default configuration is 8 clients and 4 servers with the servers connected in a mesh and two clients connecting to each server. We have disabled garbage collection §5.5 and beaconing §5.3 for the experiments reported in this section.

In addition to these controlled configurations, we use Amazon S3 as a testbed: we configure Depot to use the existing, unmodified Amazon S3 service for storage rather than storing data on full-fledged Depot servers. Note that as detailed below, the guarantees Depot provides in this configuration are somewhat weaker than it provides when Depot is used at both clients and servers. We compare the performance and costs of this configuration with that of a baseline system that uses S3 directly, without Depot's safeguards.

We first report the result of a few microbenchmarks for reference. We then present the results from our local testbed experiments in §6.4 and from our S3 experiments in §6.5.

## 6.3 Microbenchmarks

**Berkeley DB.** We show the result of a simple benchmark on Berkeley DB to profile its latency and CPU overheads. The benchmark issues a sequence of GET and PUT operations to randomly chosen keys from a volume of 1000 keys. We report the latency and CPU utilization for various object sizes. Berkeley DB was configured to commit data to disk after every GET and PUT. The Berkeley DB cache size was set to 100 MB for all these benchmarks. We used the *base* API for Java version of Berkeley DB and we invoked `DbTransaction.commitSync()` after every PUT and GET transaction. Figure 4 reports the observed CPU utilization and latency values averaged over 1000 operations.

We observe that the Berkeley DB latencies have significant standard deviation, and that both the mean and standard deviation increase as object size increases. Berkeley DB is not log-structured: it stores data in small files on the underlying filesystem. Both the allocation of new data blocks and the creation of new files require additional disk seeks. We speculate that these additional periodic disk seeks are responsible for high variance in Berkeley DB PUT latencies. Furthermore, for 1 MB objects, the GET latency increases and the write latency slightly decreases with the increase in the number of PUTs. We speculate that, because Berkeley DB stores data as a B-tree, small number of PUTs cause frequent reorganization of tree whereas with large number of PUTs reorganizations of the PUTs are relatively rare causing improved PUT latencies. Conversely, with increasing number of PUTs, GETs require more B-tree traversals and therefore incur higher latencies.

| OpType | Workload Size | Workload # of Ops | Latency (ms) ($\mu\pm\sigma$) | Latency (ms) 90 Perc | CPU (ms/req) |
|---|---|---|---|---|---|
| PUT | 3B | 700 | 1.4±0.8 | 1.2 | 0.9 |
| PUT | 3B | 1500 | 1.4±0.7 | 1.2 | 0.9 |
| PUT | 3B | 3250 | 1.3±0.7 | 1.2 | 0.6 |
| PUT | 3B | 5500 | 1.4±1.5 | 1.2 | 0.4 |
| PUT | 10KB | 700 | 2.6±1.2 | 2.4 | 1.2 |
| PUT | 10KB | 1500 | 2.6±1.4 | 2.4 | 1.1 |
| PUT | 10KB | 3250 | 2.6±1.7 | 2.4 | 0.7 |
| PUT | 10KB | 5500 | 2.6±1.6 | 2.4 | 0.5 |
| PUT | 1MB | 700 | 26.1±12.4 | 23.1 | 9.2 |
| PUT | 1MB | 1500 | 26.1±16.5 | 22.9 | 8.7 |
| PUT | 1MB | 3250 | 26.0±12.2 | 23.1 | 9.4 |
| PUT | 1MB | 5500 | 25.8±12.5 | 22.8 | 8.0 |
| GET | 3B | 700 | 0.3±0.5 | 0.2 | 1.0 |
| GET | 3B | 1500 | 0.2±0.5 | 0.1 | 1.0 |
| GET | 3B | 3250 | 0.2±0.5 | 0.1 | 0.6 |
| GET | 3B | 5500 | 0.2±0.4 | 0.1 | 0.5 |
| GET | 10KB | 700 | 0.4±0.9 | 0.2 | 1.1 |
| GET | 10KB | 1500 | 0.3±0.9 | 0.2 | 1.1 |
| GET | 10KB | 3250 | 0.3±1.0 | 0.2 | 0.8 |
| GET | 10KB | 5500 | 0.3±1.1 | 0.1 | 0.6 |
| GET | 1MB | 700 | 8.0±9.1 | 6.2 | 10.2 |
| GET | 1MB | 1500 | 7.8±8.9 | 6.2 | 10.1 |
| GET | 1MB | 3250 | 20.7±10.9 | 18.6 | 8.6 |
| GET | 1MB | 5500 | 23.9±9.7 | 21.8 | 8.2 |

FIG. 4—Latency (mean, standard deviation, and 90th percentile) and CPU utilization of Berkeley DB on 1000 random PUTs and GETs of varying object sizes. We observe that the Berkeley DB latencies have significant standard deviation and both the mean and standard deviation increase with the increase in object size. Furthermore, for 1 MB objects, the GET latency increases and the write latency slightly decreases with the increase in the number of PUTs.

| OpType | Size | Latency (ms) ($\mu\pm\sigma$) | Latency (ms) 90 Percentile | CPU (ms/req) |
|---|---|---|---|---|
| Hash | 3B | 0.1±0.3 | 0.0 | 0.0 |
| Hash | 10KB | 0.3±0.8 | 0.2 | 0.0 |
| Hash | 1MB | 15.5±0.5 | 15.5 | 14.2 |
| RSA-Sign | 10KB | 4.2±0.7 | 4.0 | 3.2 |
| RSA-Verify | 10KB | 0.3±0.5 | 0.2 | 0.0 |

FIG. 5—Latency and CPU utilization of a SHA-256, RSA-sign, and RSA-verify averaged over 1000 operations for various object sizes. The SHA-256 computation time grows with the increase in input size.

**SHA-256 and RSA sign/verify.** Figure 5 shows the typical costs of executing cryptographic operations like signature generation, signature verification, SHA-256 hash computation for various object sizes. We use the Sun Java security library to implement these crypto-



www.manaraa.com

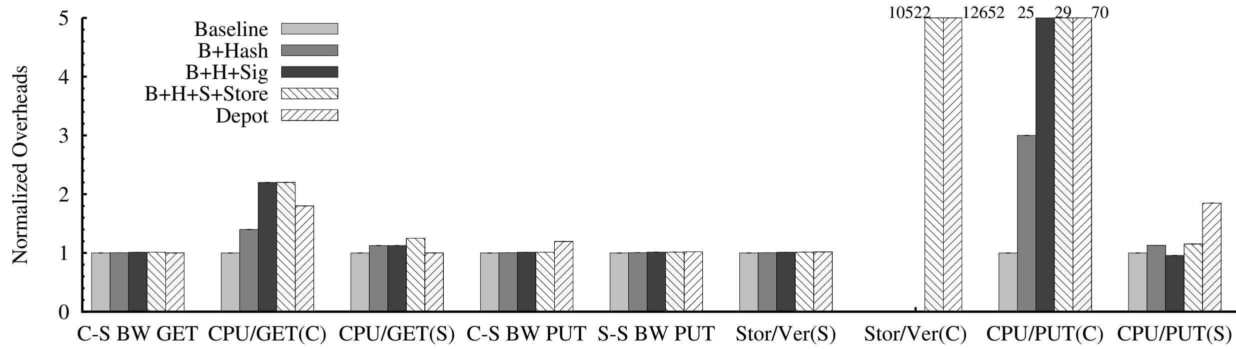



FIG. 6—Normalized resource utilization of Depot running on the local testbed for 10 KB objects. (S) indicates the aggregate resource utilization at servers whereas (C) indicates the aggregate resource utilization at clients. The labels on the Stor/Ver (C) bars indicate the actual cost; we can't report normalized cost because the storage cost in the baseline is 0. The labels, rather than the y-axis, provide the correct normalized values for CPU/PUT (C) bars of B+Hash, B+H+Sig, and Depot systems because the values for these bars are disproportionately large and representing these large values on y-axis would have compromised the visual clarity for other bars. Overall, we observe that Depot's overheads are small for GET bandwidth and CPU, server-server bandwidth, and server storage cost. The PUT client-server bandwidth overheads are modest: about 20%. The PUT client CPU overheads are substantial due to the additional Berkeley DB access and cryptographic checks. Depot's client storage overheads are also substantial due to the added requirement for the clients to store data for PUTs that they create and metadata for all PUTs.

graphic operations. The cost of hash computation grows with the increase in input size.

## 6.4 Local testbed

In this section, we use the *local testbed* configuration described above to answer the question: how much do Depot's safeguards cost over baseline variants in terms of network bandwidth, client and server storage, client and server CPU cycles, dollars spent, and latency?

To understand these costs, we run workloads under various parameters. The workload consists of a sequence of PUTs and GETs from and to a volume preloaded with 1000 key-value pairs. We use the term object and value interchangeably in this discussion. We partition the write key set into several non-overlapping ranges, one for each client. This approach simplifies the analysis but does not measure the cost of concurrent writes. Write keys are chosen randomly from the write key set while read keys are chosen randomly from the entire volume. We fix the key size at 32 bytes and experiment with three different value sizes: 3 bytes, 10 KB, and 1 MB. We experiment with the following read-write percentages: 0/100, 10/90, 50/50, 90/10, 100/0. Each client issues one operation per second, and each run lasts for 10 minutes.

We first consider the cost in terms of network bandwidth, storage, CPU cycles (§6.4.1) and then convert these numbers to an estimated dollar cost (§6.4.2). We then evaluate the latency overheads (§6.4.3).

### 6.4.1 Resource utilization

Figure 6 shows the normalized resource utilization of Depot for 10 KB objects compared to other baseline variants in the local testbed experiments described above. Figure 7 shows the measured resource consumption numbers in the experiments with Depot and the baseline variants for various object sizes and read/write percentages. The network bandwidth numbers depicted in this figure correspond to the payload size and not the actual network traffic. The storage numbers give the on-disk space consumed by Berkeley DB across the client and server nodes. Similarly, the CPU numbers reflect aggregate CPU utilization across client and server nodes as reported through Linux's `/proc` interface. We don't measure CPU utilization for individual operations. As a result, the reported numbers include GETs, PUTs, and any background processing done by our system.

**Network bandwidth.** The network bandwidth overheads of our system are small for 10 KB values: $\sim$20% additional client-server PUT network bandwidth, $\sim$2% additional server-server network bandwidth, and $\sim$0.02% additional client-server GET network bandwidth. For larger values, these overheads further diminish. For small values (3 bytes), our metadata overheads dominate the total network bandwidth. However, for larger values on the order of megabytes, the relative costs are far smaller.

Depot increases bandwidth consumption for two reasons. First, Depot includes additional information in updates to prevent faulty nodes from modifying or omitting updates. The absolute bandwidth overheads due to this factor are small and constant—285 bytes of metadata per-PUT for Depot (resulting from a 128 byte signature, 32 byte value hash, 32 byte history hash, 32 byte key, and few other fields which carry length and version information) versus 38 bytes of metadata for the baseline system



المنارة للاستشارات
www.manaraa.com

| Percentage (r/w) | Workload Size | Machinery | Client-Server BW (B) GET | Client-Server BW (B) PUT | Server-Server BW (B) Bytes/PUT | Storage/Version (B) Client | Storage/Version (B) Server | CPU (ms/req) Client | CPU (ms/req) Server |
|---|---|---|---|---|---|---|---|---|---|
| 100/0 | 3B | Baseline | 40 | NA | 0 | 0 | 868 | 0.5 | 0.7 |
| 100/0 | 3B | B+Hash | 72 | NA | 0 | 0 | 1032 | 0.6 | 0.9 |
| 100/0 | 3B | B+H+Sign | 173 | NA | 0 | 0 | 1419 | 0.9 | 0.9 |
| 100/0 | 3B | B+H+Sig+Store | 173 | NA | 0 | 269 | 1412 | 1.0 | 0.9 |
| 100/0 | 3B | Depot | 42 | NA | 0 | 2146 | 1611 | 0.9 | 0.7 |
| 90/10 | 3B | Baseline | 40 | 39 | 262 | 0 | 1023 | 0.4 | 1.1 |
| 90/10 | 3B | B+Hash | 72 | 71 | 357 | 0 | 1146 | 0.6 | 1.3 |
| 90/10 | 3B | B+H+Sign | 173 | 172 | 662 | 0 | 1568 | 1.3 | 1.2 |
| 90/10 | 3B | B+H+Sig+Store | 173 | 172 | 665 | 296 | 1554 | 1.4 | 1.2 |
| 90/10 | 3B | Depot | 42 | 2027 | 797 | 2300 | 1724 | 2.9 | 1.6 |
| 0/100 | 3B | Baseline | NA | 39 | 227 | 0 | 1071 | 0.2 | 2.9 |
| 0/100 | 3B | B+Hash | NA | 71 | 322 | 0 | 1198 | 0.4 | 3.4 |
| 0/100 | 3B | B+H+Sign | NA | 172 | 624 | 0 | 1600 | 4.7 | 3.5 |
| 0/100 | 3B | B+H+Sig+Store | NA | 172 | 624 | 288 | 1594 | 5.4 | 3.5 |
| 0/100 | 3B | Depot | NA | 2029 | 769 | 2414 | 1811 | 13.3 | 6.2 |
| 100/0 | 10KB | Baseline | 10278 | NA | 0 | 0 | 41844 | 0.5 | 0.8 |
| 100/0 | 10KB | B+Hash | 10310 | NA | 0 | 0 | 41954 | 0.7 | 0.9 |
| 100/0 | 10KB | B+H+Sign | 10410 | NA | 0 | 0 | 42378 | 1.1 | 0.9 |
| 100/0 | 10KB | B+H+Sig+Store | 10410 | NA | 0 | 10508 | 42403 | 1.1 | 1.0 |
| 100/0 | 10KB | Depot | 10280 | NA | 0 | 12386 | 42563 | 0.9 | 0.8 |
| 90/10 | 10KB | Baseline | 10278 | 10277 | 30862 | 0 | 41899 | 0.4 | 1.2 |
| 90/10 | 10KB | B+Hash | 10310 | 10309 | 30947 | 0 | 41984 | 0.8 | 1.4 |
| 90/10 | 10KB | B+H+Sign | 10410 | 10409 | 31295 | 0 | 42491 | 1.5 | 1.3 |
| 90/10 | 10KB | B+H+Sig+Store | 10410 | 10409 | 31232 | 10531 | 42409 | 1.6 | 1.4 |
| 90/10 | 10KB | Depot | 10280 | 12271 | 31251 | 12531 | 42526 | 2.8 | 1.9 |
| 50/50 | 10KB | Baseline | 10278 | 10277 | 30880 | 0 | 41910 | 0.4 | 2.9 |
| 50/50 | 10KB | B+Hash | 10310 | 10309 | 30844 | 0 | 41912 | 0.7 | 3.2 |
| 50/50 | 10KB | B+H+Sign | 10410 | 10409 | 31095 | 0 | 42233 | 3.1 | 3.4 |
| 50/50 | 10KB | B+H+Sig+Store | 10410 | 10409 | 31045 | 10507 | 42242 | 3.5 | 3.3 |
| 50/50 | 10KB | Depot | 10280 | 12265 | 31323 | 12605 | 42664 | 8.4 | 5.6 |
| 10/90 | 10KB | Baseline | 10278 | 10277 | 30692 | 0 | 41838 | 0.3 | 4.4 |
| 10/90 | 10KB | B+Hash | 10310 | 10309 | 30798 | 0 | 41992 | 0.4 | 4.4 |
| 10/90 | 10KB | B+H+Sign | 10410 | 10409 | 31130 | 0 | 42389 | 4.3 | 4.1 |
| 10/90 | 10KB | B+H+Sig+Store | 10410 | 10409 | 31090 | 10528 | 42341 | 4.9 | 4.6 |
| 10/90 | 10KB | Depot | 10280 | 12268 | 31256 | 12647 | 42592 | 12.9 | 8.0 |
| 0/100 | 10KB | Baseline | NA | 10277 | 30791 | 0 | 42090 | 0.2 | 4.6 |
| 0/100 | 10KB | B+Hash | NA | 10309 | 30933 | 0 | 42267 | 0.6 | 5.2 |
| 0/100 | 10KB | B+H+Sign | NA | 10409 | 31203 | 0 | 42618 | 5.0 | 4.4 |
| 0/100 | 10KB | B+H+Sig+Store | NA | 10409 | 31216 | 10522 | 42649 | 5.7 | 5.3 |
| 0/100 | 10KB | Depot | NA | 12269 | 31390 | 12652 | 42829 | 14.0 | 8.5 |
| 100/0 | 1MB | Baseline | 1048615 | NA | 0 | 0 | 4214734 | 1.5 | 7.9 |
| 100/0 | 1MB | B+Hash | 1048647 | NA | 0 | 0 | 4214891 | 16.9 | 8.5 |
| 100/0 | 1MB | B+H+Sign | 1048747 | NA | 0 | 0 | 4228880 | 17.7 | 8.7 |
| 100/0 | 1MB | B+H+Sig+Store | 1048747 | NA | 0 | 1048992 | 4214247 | 18.0 | 8.4 |
| 100/0 | 1MB | Depot | 1048617 | NA | 0 | 1050936 | 4253474 | 17.1 | 8.3 |
| 90/10 | 1MB | Baseline | 1048615 | 1048614 | 3145990 | 0 | 4206477 | 1.6 | 12.8 |
| 90/10 | 1MB | B+Hash | 1048647 | 1048646 | 3135363 | 0 | 4213542 | 17.2 | 14.2 |
| 90/10 | 1MB | B+H+Sign | 1048747 | 1048746 | 3158797 | 0 | 4209598 | 18.3 | 14.2 |
| 90/10 | 1MB | B+H+Sig+Store | 1048747 | 1048746 | 3142271 | 1049036 | 4203402 | 20.6 | 13.2 |
| 90/10 | 1MB | Depot | 1048617 | 1050624 | 3171806 | 1051131 | 4227101 | 21.5 | 21.1 |
| 0/100 | 1MB | Baseline | NA | 1048614 | 3128129 | 0 | 4178196 | 0.4 | 55.1 |
| 0/100 | 1MB | B+Hash | NA | 1048646 | 3129267 | 0 | 4182253 | 18.9 | 55.0 |
| 0/100 | 1MB | B+H+Sign | NA | 1048746 | 3127915 | 0 | 4179411 | 23.4 | 55.1 |
| 0/100 | 1MB | B+H+Sig+Store | NA | 1048746 | 3128115 | 1048477 | 4178022 | 36.3 | 54.3 |
| 0/100 | 1MB | Depot | NA | 1050604 | 3129905 | 1051091 | 4183385 | 44.0 | 123.1 |

FIG. 7—Comparison of bandwidth and storage costs of various approaches over a range of workloads and object sizes. The network bandwidth overheads of our system are small—20% additional client-server PUT network bandwidth, 2% additional server-server network bandwidth, 0.02% addition client-server GET network bandwidth—for 10 KB objects. The server storage overheads are likewise small. The client storage overheads are non-trivial because Depot requires clients to storage all metadata and data for PUTs they create. The CPU overheads are significant and result primarily from Berkeley DB and cryptographic operations like SHA-256 computation, RSA signing, and RSA verification.



المنارة للاستشارات
www.manaraa.com

regardless of the size of the value. Second, to ensure consistency despite faults, clients in Depot receive and verify all updates and not just the updates whose values are returned by a subsequent GET by that client.

Due to both these factors, the PUT network bandwidth of Depot is higher than that of baseline variants. For example, in our experimental configuration with 8 clients, we pay roughly 2 KB ($= 285 \times 7$) of overhead per PUT. In comparison, the B+Hash system pays about 70 bytes and the B+H+Sign system pays about 170 bytes for each PUT.

The GET network bandwidth of Depot matches that of the baseline system because clients in Depot prefetch and store updates. Once a client receives the update for a key, subsequent GETs to the same key don't require any update transfer until a new PUT to that key occurs. Hence, storing this update locally makes GETs very cheap in Depot, making it an effective choice for read-dominated workloads. Depot transfers fewer bytes per-GET than B+Hash, B+H+Sign, and B+H+S+Store because these systems need to fetch the hash or signature attached to the value on every GET whereas Depot stores the update for subsequent uses.

The *total server-server* network bandwidth includes the bandwidth required to distribute the PUT obtained by one server to other servers. We used 4 servers in our experiment and therefore the expected server-server bandwidth is 3 times the value and update size for each PUT (servers distribute the value and update to other servers). The network bandwidth of Depot is higher than that of baseline variants due to the additional per-PUT metadata. However, for objects of size 10 KB or larger, the increase is small: about 2% for 10 KB objects and about 0.8% for 1 MB objects.

**Storage.** The client and the server storage columns in Figure 7 show the aggregate disk utilization across the system for each *version*. Recall that each PUT creates a new version in our implementation. The disk utilization for each version includes the value and update storage cost at the authoring client and all the servers, and update storage cost at the remaining clients.

The server storage overheads of our system are small and result primarily from the additional metadata attached to each PUT. As in the case of network bandwidth, the additional cost due to update-metadata is small and diminishes in relative magnitude for moderate object sizes. For moderately large objects, the storage cost is dominated by the data storage cost; that is, the update storage cost is a negligible fraction of the total storage cost. For small objects, the metadata overheads dominate the storage cost. In our implementation, the baseline system stores around 200 bytes of metadata (including the 100 bytes of metadata added by Berkeley DB) for each PUT whereas Depot stores around 400 bytes for each PUT including the 285 bytes of update metadata for each put and 100 bytes of Berkeley DB metadata.

The client storage cost is significant and consists of the cost of storing the value and update at clients. Applications that don't care about availability or durability in the presence of server failures may reduce this overhead by not storing values at clients. However, clients must still store their PUT updates locally. The reason is that, as explained in §4, this per-PUT update is needed for consistency. Moreover, according to our rough model (§6.2), storing a byte for a month costs the same as fetching this byte once. Therefore, if this update is likely to be reused within a month (perhaps for serving another GET to the same key or for verifying a later GET to another key in the same volume), it is economical to store the data.

**CPU cycles.** Depot's CPU overheads are significant compared to the CPU consumption of the baseline system. Just like other metrics, the numbers reported in Figure 7 reflect the aggregate utilization across clients and servers. Note that while the CPU number per request for GET-all (100/0) and PUT-all (0/100) workloads accurately reflect the CPU utilization of GETs and PUTs respectively, workloads like 90/0, 50/50, 10/90 do not provide accurate CPU utilization measurements. The reason behind this limitation of our measurement is that, as stated earlier, we don't measure CPU utilization on a per-request level.

The difference of CPU utilization between the baseline variants and Depot is dominated by the CPU cost of additional Berkeley DB accesses at clients and servers and added cryptographic operations. For example, consider the client CPU time of executing a GET for 1 MB object. The baseline system takes 1.5 ms of CPU time per-GET whereas B+Hash, that performs an additional SHA-256 computation on 1 MB object, takes 16.9 ms of CPU time per-GET. The difference ($16.9 - 1.50 = 15.40$ ms) is dominated by the 14.45 ms of CPU time required to compute the SHA-256 of 1 MB object. As another example, consider the difference in client CPU time of a PUT in B+H+S+Store and B+H+Sig. The B+H+S+Store system takes 36.30 ms whereas the B+H+Sig system takes 23.4 ms. The difference ($36.30 - 23.40 = 12.90$ms) is dominated by the 8 ms of CPU time required to store 1 MB object in Berkeley DB.

The server CPU utilization for various baseline variants remains mostly constant as expected—the only difference between these variants from the point of view of server is the size of the object.

We note that, contrary to the intuition, GETs in Depot are cheaper than the GETs in the B+H+Sig and B+H+S+Store for 1 MB objects. Two factors are responsible for this behavior. First, unlike B+H+Sig and



المنارة للاستشارات
www.manaraa.com

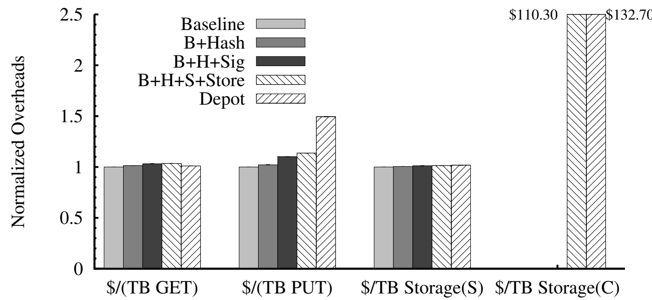



FIG. 9—Dollar cost of Depot running on the local testbed. Using our cost model, we convert Depot's network, storage, and CPU consumption to a common currency and compare this aggregate cost to that of the Baseline, B+Hash, B+H+Sign, and B+H+S+Store systems. For objects 10KB or larger, Depot's cost overheads are modest. For visual clarity, we scale the figure to include all but client storage cost ($/TB Storage (C)), which we depict by suitably labeling the bars. The GET and server storage overheads of Depot are very small. PUTs are 50% more expensive in Depot relative to the baseline, owing to the additional metadata transfers between clients and servers, and between servers. The client storage overheads are significant because in the Baseline system, clients do not store their PUTs.

B+H+S+Store, Depot clients don't verify signatures on fast get paths thereby avoiding the latency of signature verification. Second, our B+Hash, B+H+Sign, and B+H+S+Store implementations do an additional memory allocation where we copy the original payload into a new buffer after appending appropriate cryptographic information, such as a SHA-256 hash or a RSA signature. This extra work makes our baseline variants more expensive than an ideal implementation and makes our results look slightly better than their true value. We speculate that this extra memory allocation and copying poses noticeable CPU and latency overheads and are responsible for the anomalous GET values. While we have not fixed this anomaly yet, we have verified that excess memory pressure can significantly increase overall CPU utilization and latency. This anomaly is masked for PUTs, perhaps, due to the other overheads dominating in Depot, and it doesn't manifest for small objects.

**Summary.** We observe that Depot has minimal impact on the GET bandwidth and the server-server storage cost for moderate size objects. Depot causes modest increase in the PUT bandwidth, which gradually diminishes with increasing object sizes. Depot has significant cost on client storage because, unlike baseline variants that don't require any client storage, Depot requires clients to store the values that they insert on PUTs and the updates for values that they retrieve on GETs.

### 6.4.2 Dollar cost

Figure 7 shows several aspects of Depot's overheads, but in any given environment, some aspects are more important than others. Figure 8 shows the same overheads weighted according to the cost model presented above to obtain an estimated operation cost for GET-all and PUT-all workloads. The dollar costs in Figure 8 are scaled to 1 TB of objects stored for 30 days. For example, for 10 KB objects, the dollar costs reflect our estimate of the cost of executing $10^8$ PUTs or GETs of the 10 KB object. Figure 9 shows a normalized slice of Figure 8 corresponding to the object size of 10 KB in the form of a bar graph.

**Analytical model.** In addition to the measured resources, Figure 8 also includes a lower bound on dollar cost for baseline variants. We use an analytical model to estimate the storage and network bandwidth costs for this lower bound. The model assumes that objects of a specified value size are stored at the servers. The model augments the objects to include any metadata needed for a given baseline variant (for example, B+Hash stores a 32 byte SHA 256 hash of the value in the object. B+H+Sign stores a 128 byte RSA signature in the object). In case of B+H+S+Store, these objects are also stored at the client. We also add a fixed length metadata field to account for any additional information like key length, value length etc that a real protocol will have to send. The fixed length metadata is about 2 bytes for Baseline system, 4 bytes for B+Hash, 8 bytes for B+H+Sign and B+H+S+Store. The storage cost in bytes is computed by first fixing the number of objects and the value size, and then estimating the amount of storage consumed in bytes for storing these objects. The storage cost in bytes can be easily translated to dollar costs using the cost model described in §6.2.

We model GET as transferring a key from the client to a server and transferring an object from the server back to the client, and we model PUT as transferring (key, object) from the client to a server and then the server transferring this pair to other servers in the cluster. We assume the CPU costs to be zero so that we could treat the costs obtained from the model as a lower bound on the true costs. The cost of a request in dollars is computed by estimating the number of bytes transferred for each operation and then converting the bandwidth costs into dollars using the cost model described in §6.2. These computed costs are presented in Figure 8.

One of the limitations of this model is that for a PUT, the model assumes that the (key, object) pair is synchronously transferred from the receiving server to the remaining servers in the cluster. Instead, our implementation of Depot and baseline variants propagates key-value pairs asynchronously and as a result, may not finish all the propagation when experiment is terminated. Due to this factor, we may, at times, observe a slightly



المنارة للاستشارات
www.manaraa.com

| Workload | | | Cost ($/TB) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Perc(r/w) | Size | Machinery | M-PUT | L-PUT | M-PUT | L-GET | M-S Store | L-S Store | M-C Store | L-C Store |
| 100/0 | 3B | Baseline | NA | NA | 1194.7 | 13811.7 | 102.4 | 7409.2 | 0.0 | 0.0 |
| 100/0 | 3B | B+Hash | NA | NA | 2423.5 | 18419.0 | 1331.2 | 8813.2 | 0.0 | 0.0 |
| 100/0 | 3B | B+H+Sign | NA | NA | 5836.8 | 24530.7 | 4744.5 | 12111.4 | 0.0 | 0.0 |
| 100/0 | 3B | B+H+Sig+Store | NA | NA | 5836.8 | 25126.0 | 4744.5 | 12057.2 | 0.0 | 9212.1 |
| 100/0 | 3B | Depot | NA | NA | NA | 17562.4 | NA | 13753.5 | NA | 73280.5 |
| 0/100 | 3B | Baseline | 1587.2 | 34439.1 | NA | NA | 102.4 | 9146.1 | 0.0 | 0.0 |
| 0/100 | 3B | B+Hash | 3184.6 | 42886.4 | NA | NA | 1331.2 | 10225.3 | 0.0 | 0.0 |
| 0/100 | 3B | B+H+Sign | 7622.0 | 91615.0 | NA | NA | 4744.5 | 13654.2 | 0.0 | 0.0 |
| 0/100 | 3B | B+H+Sig+Store | 7622.0 | 98749.7 | NA | NA | 4744.5 | 13607.2 | 4744.5 | 9859.9 |
| 0/100 | 3B | Depot | NA | 271177.6 | NA | NA | NA | 15460.7 | NA | 82421.4 |
| 100/0 | 10KB | Baseline | NA | NA | 102.7 | 106.6 | 102.4 | 104.6 | 0.0 | 0.0 |
| 100/0 | 10KB | B+Hash | NA | NA | 103.1 | 108.1 | 102.8 | 104.9 | 0.0 | 0.0 |
| 100/0 | 10KB | B+H+Sign | NA | NA | 104.1 | 110.0 | 103.8 | 105.9 | 0.0 | 0.0 |
| 100/0 | 10KB | B+H+Sig+Store | NA | NA | 104.1 | 110.3 | 103.8 | 106.0 | 0.0 | 105.1 |
| 100/0 | 10KB | Depot | NA | NA | NA | 107.7 | NA | 106.4 | NA | 123.9 |
| 0/100 | 10KB | Baseline | 133.5 | 148.0 | NA | NA | 102.4 | 105.2 | 0.0 | 0.0 |
| 0/100 | 10KB | B+Hash | 134.0 | 151.2 | NA | NA | 102.8 | 105.7 | 0.0 | 0.0 |
| 0/100 | 10KB | B+H+Sign | 135.3 | 163.1 | NA | NA | 103.8 | 106.5 | 0.0 | 0.0 |
| 0/100 | 10KB | B+H+Sig+Store | 135.3 | 168.2 | NA | NA | 103.8 | 106.6 | 103.8 | 105.2 |
| 0/100 | 10KB | Depot | NA | 221.2 | NA | NA | NA | 107.1 | NA | 126.5 |
| 100/0 | 1MB | Baseline | NA | NA | 102.4 | 102.7 | 102.4 | 102.9 | 0.0 | 0.0 |
| 100/0 | 1MB | B+Hash | NA | NA | 102.4 | 103.1 | 102.4 | 102.9 | 0.0 | 0.0 |
| 100/0 | 1MB | B+H+Sign | NA | NA | 102.4 | 103.2 | 102.4 | 103.2 | 0.0 | 0.0 |
| 100/0 | 1MB | B+H+Sig+Store | NA | NA | 102.4 | 103.2 | 102.4 | 102.9 | 0.0 | 102.4 |
| 100/0 | 1MB | Depot | NA | NA | NA | 103.1 | NA | 103.8 | NA | 102.6 |
| 0/100 | 1MB | Baseline | 133.1 | 134.6 | NA | NA | 102.4 | 102.0 | 0.0 | 0.0 |
| 0/100 | 1MB | B+Hash | 133.1 | 135.1 | NA | NA | 102.4 | 102.1 | 0.0 | 0.0 |
| 0/100 | 1MB | B+H+Sign | 133.1 | 135.3 | NA | NA | 102.4 | 102.0 | 0.0 | 0.0 |
| 0/100 | 1MB | B+H+Sig+Store | 133.1 | 135.6 | NA | NA | 102.4 | 102.0 | 102.4 | 102.4 |
| 0/100 | 1MB | Depot | NA | 138.0 | NA | NA | NA | 102.1 | NA | 102.6 |

FIG. 8—Comparison of dollar costs derived from the bandwidth, server storage, and CPU costs of various approaches over a range of workloads. Server storage costs are reported on a per server basis whereas client storage costs are aggregated across all the clients. We also show the analytical costs computed using the model described in Section 6.4.2. The analytical costs are prefixed with "M-" whereas the empirical costs are prefixed with "L-". We don't report the analytical costs for Depot because we have not modeled Depot analytically. Depot has small costs on GETs and server storage. PUT costs are higher but diminish in relative magnitude as the object size is increased. Client storage costs are significant because Depot requires clients to store data for PUTs they create and metadata for all PUTs. The analytical costs differ significantly from the empirical costs for 3 byte objects because the CPU cycles, that constitute a dominant fraction of the overall costs, are ignored by the analytical model.

lower put bandwidth and server storage cost than expected. However, looking at the logs reveals that no more than 1% values were missing at any node in any of our execution. Thus, 1% of extra overhead should be added to Depot and other baseline variant system's overheads when comparing against the analytical model.

**Comparison.** We observe that Depot's dollar overheads for GET and server storage are small and diminish for large object sizes. For example, for 10 KB objects, a GET is only 5% more expensive than the analytical lower bound for the baseline system. For small objects, the excessive CPU cost and update make Depot an order of magnitude more expensive than the baseline system.

The server storage dollar overhead is also very small. Depot is 4% more expensive compared to the analytical lower bound for baseline and 2% more expensive compared to the empirical baseline system. The Berkeley DB metadata accounts for a significant fraction of this cost and we are looking to optimize this overhead in near future. Like in resource overhead discussion (§6.4.1), the storage costs in Figure 8 are reported for each version.

The PUT overheads are significant due to the additional metadata that PUTs require to be transmitted. For small objects, even the processing costs introduce significant overhead. For large objects, the PUT costs become more favorable. In the future, we are looking to optimize the PUT cost by reducing the size of an update.

The client storage costs are also significant. Note that while Figure 8 includes the average storage costs for servers, it includes the cumulative storage costs for clients.

**Summary.** In summary, for 10 KB objects replicated to 4 servers, Depot costs $776.1 per TB of PUTs (including storage costs) in comparison to the $543.1 per TB required by the analytical lower bound for the baseline system and $568.8 per TB required by the empirical



المنارة للاستشارات
www.manaraa.com

| Workload | | | $\mu \pm \sigma$ (ms) | Percentiles (ms) | | |
|---|---|---|---|---|---|---|
| Op | Size | Machinery | | 50 | 90 | 99 |
| GET | 3B | Baseline | 0.7±0.5 | 1 | 1 | 1 |
| GET | 3B | B+Hash | 0.8±0.9 | 1 | 1 | 1 |
| GET | 3B | B+H+Sign | 1.2±0.5 | 1 | 2 | 2 |
| GET | 3B | B+H+S+Store | 1.2±0.5 | 1 | 2 | 2 |
| GET | 3B | Depot | 0.8±0.6 | 1 | 1 | 1 |
| PUT | 3B | Baseline | 2.8±3.6 | 2 | 6 | 16 |
| PUT | 3B | B+Hash | 3.4±6.6 | 2 | 5 | 37 |
| PUT | 3B | B+H+Sign | 6.8±3.0 | 6 | 8 | 21 |
| PUT | 3B | B+H+S+Store | 8.5±5.8 | 7 | 10 | 28 |
| PUT | 3B | Depot | 11.4±4.7 | 10 | 17 | 26 |
| GET | 10KB | Baseline | 1.0±0.6 | 1 | 1 | 2 |
| GET | 10KB | B+Hash | 1.3±0.6 | 1 | 2 | 3 |
| GET | 10KB | B+H+Sign | 1.6±0.7 | 2 | 2 | 3 |
| GET | 10KB | B+H+S+Store | 1.6±0.7 | 2 | 2 | 3 |
| GET | 10KB | Depot | 1.4±1.1 | 1 | 2 | 4 |
| PUT | 10KB | Baseline | 3.3±3.2 | 2 | 5 | 15 |
| PUT | 10KB | B+Hash | 4.1±5.3 | 3 | 7 | 24 |
| PUT | 10KB | B+H+Sign | 8.2±3.9 | 7 | 10 | 23 |
| PUT | 10KB | B+H+S+Store | 10.1±5.7 | 9 | 13 | 27 |
| PUT | 10KB | Depot | 14.8±6.5 | 13 | 21 | 34 |
| GET | 1MB | Baseline | 16.3±8.1 | 15 | 21 | 43 |
| GET | 1MB | B+Hash | 32.3±8.3 | 30 | 40 | 59 |
| GET | 1MB | B+H+Sign | 32.8±8.3 | 31 | 40 | 60 |
| GET | 1MB | B+H+S+Store | 32.8±8.4 | 31 | 40 | 62 |
| GET | 1MB | Depot | 31.5±8.2 | 30 | 38 | 58 |
| PUT | 1MB | Baseline | 31.7±21.7 | 23 | 53 | 125 |
| PUT | 1MB | B+Hash | 54.0±31.1 | 42 | 83 | 192 |
| PUT | 1MB | B+H+Sign | 67.9±47.7 | 53 | 104 | 293 |
| PUT | 1MB | B+H+S+Store | 80.2±39.8 | 65 | 134 | 238 |
| PUT | 1MB | Depot | 104.8±47.3 | 89 | 159 | 307 |

FIG. 10—Mean, and {50,90,99} percentile latencies for a PUT-all (0/100) and GET-all (100/0) workload with varying object sizes. The absolute GET latencies of Depot are slightly higher than those of the Baseline system due to the additional Hash computation done in Depot. Depot pre-fetches metadata, making GETs cheaper than B+H+Sign and B+H+S+Store variants. The PUT latencies are dominated by the signature generation cost and Berkeley DB access cost for small objects whereas for large objects, hash computation cost also becomes significant. The latencies increase by an order of magnitude on increasing the object size from 3 bytes to 1 MB for all systems. The dominant factors contributing to the increase are network transfer latency, Berkeley DB access latency, increase in SHA-256 computation cost.

baseline system. GETs are more efficient. For a TB of 10 KB GETs, Depot requires $107.7 compared to $102.7 required by the analytical Baseline, and $106.6 required by the empirical Baseline system.

### 6.4.3 Latency

We now evaluate the latency overhead of Depot. Figure 10 shows the latency of PUT and GET for the baseline variants and Depot for various object sizes computed in the experiments described in §6.4. For each experiment, we report the mean and standard deviation, as well as the 50, 90, and 99 percentiles. Unfortunately, our current code was instrumented to report latencies at millisecond granularity, which is why the percentiles are reported as whole numbers. We plan to obtain more precise latency measurements in the near future.

The absolute increase in GET latency is small for small objects. The increase is solely determined by hash computation delay; for 1 MB objects, Depot increases the average GET latency of baseline system from 16.3 ms to 31.5 ms. The difference roughly corresponds to the cost of computing a SHA-256 hash on 1 MB object. The GET latencies are somewhat stable, and the median latency closely follows the mean. Note that Depot is slightly cheaper than the two baseline variants (B+H+Sign, B+H+S+Store) that sign data. The reason is that Depot retrieves the update and performs the signature verification in the background. In comparison, B+H+Sign and B+H+S+Store perform signatures on the critical path of a GET.

For large objects, Depot is even slightly cheaper than the B+Hash variant. As explained in CPU cost discussion in §6.4.1, we speculate that this anomaly is an artifact of our implementation for baseline variants.

The PUT latency of Depot is noticeably worse than the baseline system. For small objects, signature computation and Berkeley DB are the dominant source of latency. For larger objects, even the SHA-256 computation cost becomes significant. More precisely, the PUT latency in Depot results from the following main actions:

1. Signature generation at client (4.2 ms, c.f. Figure 5).
2. Value hash computation (0.3 ms for 10 KB, 15.5 ms for 1 MB, c.f. Figure 5).
3. Berkeley DB store of value and update at client (2.6 ms for 10 KB, 7.8-23.9 ms for 1 MB, c.f. Figure 4).
4. Network transfer (1 Gbps, from §6.2).
5. Serialization delays.
6. Server signature verification (0.3 ms, c.f. Figure 5).
7. Server value hash verification (0.3 ms for 10 KB, 15.5 ms for 1 MB, c.f. Figure 5).
8. Server history hash verification (0.3 ms, c.f. Figure 5).
9. Server Berkeley DB store.

Our current implementation is not optimized to pipeline these steps. We note that for small objects, the signature generation and Berkeley DB access at clients is the major source of Depot overheads. For large objects (1 MB), hash computation cost also becomes significant. Note that we run this experiment on a 1Gbit/s LAN; in many cloud storage deployments, WAN delays would shrink the relative gap.

**Variance.** There are three main sources of variance in our latencies. First, we observe that in our microbenchmarks, the Berkeley DB accesses show latencies rang-



المنارة للاستشارات
www.manaraa.com

| Percentage (r/w) | Workload Size | Machinery | GET Latency (ms) $\mu \pm \sigma$ | Percentiles 50 | 90 | 99 | PUT Latency (ms) $\mu \pm \sigma$ | Percentiles 50 | 90 | 99 |
|---|---|---|---|---|---|---|---|---|---|---|
| 100/0 | 10KB | Baseline | 1.0±0.6 | 1 | 1 | 2 | NA | NA | NA | NA |
| 100/0 | 10KB | B+Hash | 1.3±0.6 | 1 | 2 | 3 | NA | NA | NA | NA |
| 100/0 | 10KB | B+H+Sign | 1.6±0.7 | 2 | 2 | 3 | NA | NA | NA | NA |
| 100/0 | 10KB | B+H+Sig+Store | 1.6±0.7 | 2 | 2 | 3 | NA | NA | NA | NA |
| 100/0 | 10KB | Depot | 1.4±1.1 | 1 | 2 | 4 | NA | NA | NA | NA |
| 90/10 | 10KB | Baseline | 1.0±1.1 | 1 | 1 | 3 | 3.7±5.7 | 3 | 4 | 42 |
| 90/10 | 10KB | B+Hash | 1.3±1.0 | 1 | 2 | 3 | 3.6±3.9 | 3 | 5 | 13 |
| 90/10 | 10KB | B+H+Sign | 1.7±0.9 | 2 | 2 | 3 | 8.5±8.1 | 7 | 8 | 56 |
| 90/10 | 10KB | B+H+Sig+Store | 1.7±1.1 | 2 | 2 | 3 | 9.9±6.3 | 8 | 11 | 41 |
| 90/10 | 10KB | Depot | 1.4±1.5 | 1 | 2 | 7 | 16.2±10.3 | 13 | 23 | 69 |
| 50/50 | 10KB | Baseline | 1.1±1.6 | 1 | 1 | 5 | 3.3±3.3 | 3 | 4 | 13 |
| 50/50 | 10KB | B+Hash | 1.3±0.8 | 1 | 2 | 3 | 3.4±3.1 | 3 | 4 | 15 |
| 50/50 | 10KB | B+H+Sign | 1.7±1.7 | 2 | 2 | 4 | 8.0±4.5 | 7 | 9 | 21 |
| 50/50 | 10KB | B+H+Sig+Store | 1.8±1.3 | 2 | 2 | 6 | 10.3±6.1 | 9 | 14 | 40 |
| 50/50 | 10KB | Depot | 1.5±1.9 | 1 | 2 | 8 | 15.1±6.4 | 13 | 22 | 40 |
| 10/90 | 10KB | Baseline | 1.1±1.0 | 1 | 1 | 4 | 3.2±3.7 | 2 | 4 | 20 |
| 10/90 | 10KB | B+Hash | 1.2±0.7 | 1 | 2 | 3 | 2.9±1.5 | 3 | 3 | 10 |
| 10/90 | 10KB | B+H+Sign | 1.6±0.9 | 1 | 2 | 5 | 7.9±3.4 | 7 | 10 | 21 |
| 10/90 | 10KB | B+H+Sig+Store | 1.6±1.1 | 2 | 2 | 7 | 8.9±3.4 | 8 | 10 | 19 |
| 10/90 | 10KB | Depot | 1.9±5.5 | 1 | 2 | 11 | 14.9±6.4 | 13 | 21 | 33 |
| 0/100 | 10KB | Baseline | NA | NA | NA | NA | 3.3±3.2 | 2 | 5 | 15 |
| 0/100 | 10KB | B+Hash | NA | NA | NA | NA | 4.1±5.3 | 3 | 7 | 24 |
| 0/100 | 10KB | B+H+Sign | NA | NA | NA | NA | 8.2±3.9 | 7 | 10 | 23 |
| 0/100 | 10KB | B+H+Sig+Store | NA | NA | NA | NA | 10.1±5.7 | 9 | 13 | 27 |
| 0/100 | 10KB | Depot | NA | NA | NA | NA | 14.8±6.5 | 13 | 21 | 34 |

FIG. 11—Latency table for varying workloads (r/w percentages) and 10 KB objects. We show mean, standard deviation, and 50, 90, 99 percentile latency values. The GET latency increases with the increase in number of PUTs whereas the PUT latency decreases with the increase in number of PUTs. The variation of the read/write percentage has no impact on the median latencies but noticeable impact on the 99 percentile latencies. We speculate that increase in number of PUTs makes it less likely for GETs to be served from Berkeley DB cache. Conversely, PUTs become cheaper because B-tree expansions become less likely with increase in number of PUTs. This theory also explains why the median latencies are not affected but higher percentiles and mean latencies are affected.

ing from 15 ms to 100 ms for 1 MB objects. Second, we observed that JVM scheduling is not very optimal. For example, using JRockit JVM [8] showed improved and stable latencies (by 2 ms for PUTs on 10 KB objects). We have not yet switched to JRockit because (1) JRockit performance for computing SHA-256 hashes is worse than Sun JVM (JRockit takes 18 ms for computing SHA-256 hash on 1 MB as opposed to 15 ms taken by Sun JVM), and (2) Berkeley DB crashes with JRockit with over 1200 1 MB writes.

The final source of latencies in Depot is the queuing delays. The server-server gossips induce bursty load on the system, making the system transiently overloaded, even though the system as a whole is not overloaded. For example, on further investigating logs, we discovered average queuing delays of 21 ms in Depot in comparison to those of 3 ms in B+H+S+Store for 1 MB objects. The queuing delays are more prominent in Depot because server processing in Depot takes longer (∼48 ms for 1 MB objects) than that in B+H+S+Store (∼31 ms). In the future, we expect to reduce queuing delays by pipelining various steps of request processing, and by reducing burstiness of traffic by adding support for forward PUTs at servers.

**Variation with read/write percentage** Figure 11 shows the mean, standard-deviation, and {50, 90, 99} percentiles of PUT and GET latencies for various read/write percentages for 10 KB objects on the random workload described in §6.4. Figures 12, 13, 14, and 15 report the 50 percentile GET, 50 percentile PUT, 99 percentile GET, and 99 percentile PUT latencies for the same random workload as before. The expected behavior is that we should see no variation in latencies with change in read/write percentages. While the median latencies follow this expected behavior, other percentiles and mean latencies vary with the change in read/write percentage. The increase in total number of PUTs causes a slight increase in GET latency and slight decrease in PUT latency. We have yet to fully understand the reason behind this trend. However, we speculate that with the increase in the number of PUTs, the GETs are less likely be served from the Berkeley DB cache, causing an increase in GET latency. Similarly, because Berkeley DB organizes data as a B-tree, increase in the number of writes makes B-tree expansion less likely, making PUTs



المنارة للاستشارات
www.manaraa.com

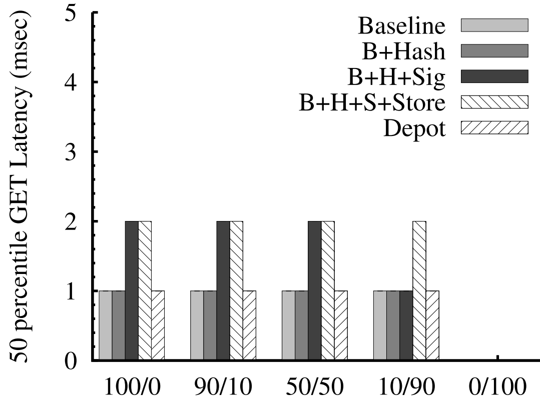



FIG. 12—50th percentile of latency for GETs with 10 KB objects and varying r/w percentages for Depot and baseline variants running on the local testbed. The latency values are stable and are roughly constant. The actual in mean between the B+Hash and B+H+Sig is small (∼0.2 ms) but because we measure latencies at the granularity of ms, this small effect is magnified due to rounding error.

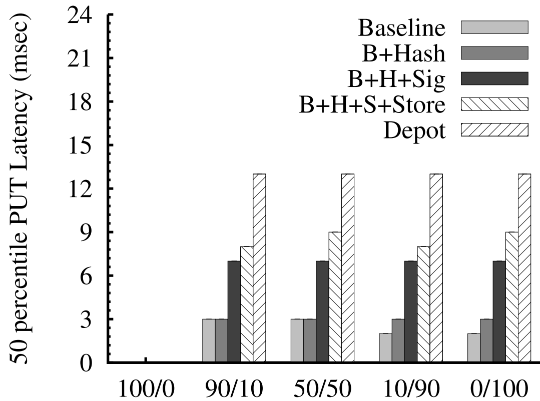



FIG. 13—50th percentile of latency of PUTs with 10 KB objects and varying r/w percentage for Depot and other baseline systems running on the local testbed. The latency values are stable but PUT latency decreases slightly as number of PUTs is increased, We speculate that this behavior results from Berkeley DB's B-tree organization of data. The difference between bars in the same group can be explained by the additional Berkeley DB accesses and cryptographic operations.

faster on average.

**Summary.** In summary, we observe that Depot has higher latencies. The GET latencies have small overhead which grows with increasing object size. The PUT overheads are dominated by the signature creation cost for small object sizes but as the object size increases, hash computation and Berkeley DB cost becomes significant. We speculate that we can see significant improvements

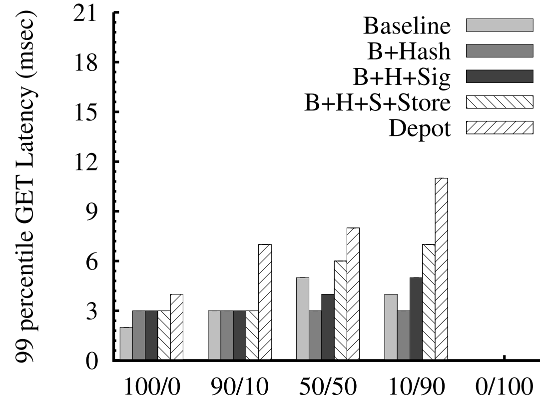



FIG. 14—99th percentile of latency for GETs with 10 KB objects and varying r/w percentages for Depot and baseline variants running on the local testbed. The latency values are not very stable yet. Generally, the latencies increase on the increasing the number of PUTs.

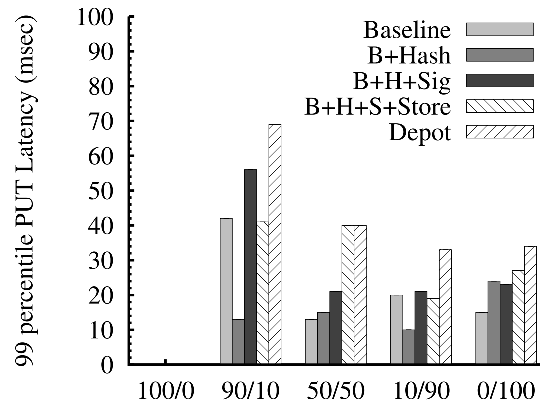



FIG. 15—99th percentile of latency for PUTs with 10 KB objects and varying r/w percentages for Depot and baseline variants running on the local testbed. The latency values are not very stable yet. Generally, the latencies decrease on the increasing the number of PUTs.

by fixing the storage layer and pipelining disk accesses with network transfers and cryptographic computations. Furthermore, moving to a WAN environment may further mitigate the increase in these latencies. We note similar behavior in our S3 experiments that we present in the next section.

## 6.5 S3 testbed

In this section we use Amazon S3 as a testbed. The high level question we would like to answer is: what is the cost of not trusting Amazon S3? As in the local testbed experiments (§6.4), we decompose this question into two further ones: (1) what are the storage and bandwidth overheads of a Depot client over a Baseline client both in



المنارة للاستشارات
www.manaraa.com

| Metric | Baseline | Depot | % Overhead |
|---|---|---|---|
| Bytes stored per 10KB object at S3 | 10240 | 10479 | 2.3 |
| Bytes stored per 10KB object at client | 0 | 10479 | NA |
| Server storage cost for 1TB with 10KB objects | \$102.72 | \$104.80 | 2.0 |
| Client storage cost for 1TB with 10KB objects | 0 | \$104.80 | NA |
| Total bytes transferred per request | 10272 | 10511 | 2.3 |
| Bandwidth cost for 1TB with 10KB objects | \$102.72 | \$105.12 | 2.3 |

FIG. 16—Bandwidth and storage overheads of the Baseline and Depot clients when both use Amazon S3 to store data. Costs are reported in dollars. Depot adds negligible overheads to server storage and network bandwidth both in terms of bytes and dollars. Client storage overheads are sizeable: to avoid trusting other nodes for durability, Depot clients have to store full copies of objects.

bytes and dollars? and (2) what is the additional latency perceived by Depot clients when compared to a Baseline client?

To answer these questions, we configure Depot to use the existing, unmodified Amazon S3 service for storage. Rather than the configuration used above (8 clients and 4 servers), here we still use 8 clients, but we do not know how many servers are employed by S3 to service our requests. The workload is 1 request per second from each client to a randomly selected object in a volume of 1000 objects, each of size 10KB. The percentage of reads/writes is 90/10. We compare storage costs in this configuration with the storage, network bandwidth and latency costs of a Baseline client that PUTs (GETs) keys to (from) S3 directly, without Depot's safeguards. Our evaluation setup does not yet allow measuring CPU overheads if the workload used is a mix of GETs and PUTs, so we have not presented the CPU overheads of Depot client that uses S3. But, we expect CPU overheads similar to those of the LAN experiments.

We use a separate code base to support this configuration. Owing to limitations of our current implementation, the guarantees provided by Depot in this case are weaker than when Depot is used at both clients and servers. This code base supports FJC consistency, bounded staleness, eventual consistency, and integrity, as described in §5. However, it does not yet support the client-to-client exchanges needed to provide the full always-exchange, read availability, and durability guarantees, and we do not implement versioning for recovery or proof of misbehavior for eviction.

Figure 16 states the overhead introduced by Depot over the baseline for storage and bandwidth in the above experiments. Total bytes transferred per request increases by about 230 bytes. Server storage overhead is about 230 bytes per object. Client storage overhead is significant: in Depot, the clients store keys and values (i.e., entire objects) while the Baseline client stores only keys. Note, though, that Depot clients could be configured to store only metadata, if they are willing to trust other nodes for durability and availability (§5.2).

We quantify these overheads in terms of extra dollars that users have to pay if they used Depot clients. For this,

| Operation | S3 roundtrip | Client addition | Total latency |
|---|---|---|---|
| Baseline GET | 164.9±199 | 0.16±0.0 | 165.0±199 |
| Depot GET | 171.3±234 | 0.22±0.1 | 171.6±234 |
| Baseline PUT | 187.0±202 | 0.30±0.0 | 187.3±202 |
| Depot PUT | 210.9±227 | 6.96±0.3 | 217.8±227 |

FIG. 17—Average latencies along with their standard deviations in milliseconds perceived by the Baseline and Depot clients for GET and PUT operations with 10KB payload when both use Amazon S3 for storage, broken down into S3 round trip time and client-side processing time. Depot adds about 7 milliseconds for PUT latency over S3 roundtrip time and negligible overheads to GET latency. Variance present in the end-to-end latency comes from the S3 roundtrip time variance, and Depot adds negligible variance to the end-to-end latency.

we use the cost model presented in §6.2 and calculate the storage and network bandwidth costs of using Depot and Baseline clients. For storage costs, we calculate the cost of storing 1TB data broken down into 10KB objects at S3, and for network bandwidth costs, we find the cost of transferring this 1TB data to/from S3 using GET and PUT requests. With this setup, the storage overhead of Depot over Baseline is about 230 bytes per 10KB object and it translates to about \$3.00. The bandwidth overhead for a request with 10KB object of Depot over Baseline is about 230 bytes and it translates to about \$2.50.

Figure 17 lists the latencies perceived by a Depot client for GET and PUT with a Baseline client in the above experiments. Depot adds about 7 msec to the end-to-end latency for PUT operations and negligible overhead for GET operations. The extra latency added by Depot comes mainly from the secure hash calculation and signature generation.

To illustrate the relative magnitude of the Depot-introduced latency, we plot the end-to-end latency experienced by Depot clients next to the S3 roundtrip component for GET and PUT operations. Figure 18 contains that plot; as can be seen, Depot's contribution to the total latency in the above experiments is negligible.

It is worth noting that the Depot clients have to transfer few extra bytes to and from S3 as compared to a Baseline client for the same payload size. However, the S3 roundtrip time has huge variance (as evident from our results presented in Figure 17). The roundtrip time to trans-



المنارة للاستشارات
www.manaraa.com

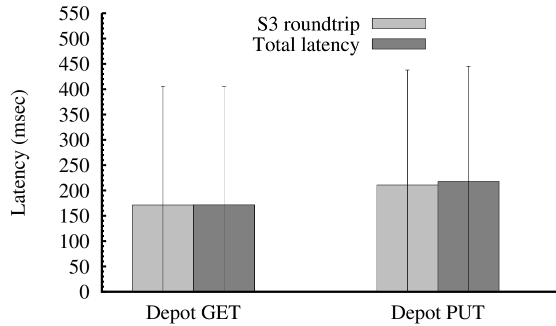



FIG. 18—Average latencies perceived by Depot and Baseline client for GET and PUT operations with 10KB payload when both use Amazon S3 for storage. Latencies are reported in milliseconds. The error bars represent the standard deviation. It is easy to conclude that the variance present in the end-to-end latency comes from the S3 roundtrip time variance and Depot adds negligible variance to the end-to-end latency. Depot operations add a small latency on top of S3 roundtrip which mainly comes from the cryptographic operations performed by Depot clients

fer these extra bytes and the latency overhead added by the Depot clients are effectively masked by the variance.

In summary, Depot adds negligible overheads in terms of latency, bandwidth, and server storage, but Depot clients have non-negligible overheads in terms of client storage if they are not willing to trust other nodes for durability and availability.

### 6.6 Summary

We have evaluated the resource utilization, dollar cost, and latency overheads of Depot in two different environments: a local testbed and unmodified Amazon S3. In both these environments, we observed that Depot has modest resource overheads for 10 KB or larger objects. Converting these raw resource utilization numbers into dollars of operation cost for 10 KB objects indicated that Depot incurs 35% overhead (in terms of dollars) on PUTs and 1% overhead on GETs in our testbed deployment and 30% overhead in Amazon S3 deployment. Client storage was the most significant cost component in both these settings; Depot requires clients to store values and metadata locally to avoid trusting other nodes for durability and availability. The latency overheads of Depot in LAN settings were modest for GETs but significant for PUTs. In the Amazon S3 deployment, Depot adds negligible additional latency over S3.

## 7 Related work

We present prior work in terms of a trade-off between availability and fault-tolerance. In contrast, Depot both maximizes availability *and* tolerates arbitrary failures.

**Low fault-tolerance, high availability.** System designers have long optimized for availability, both via provisioning and via system structure. Lately, key-value stores in clouds [3–5, 25, 31, 34] have taken a pragmatic approach to availability, achieving this property through both system design and choice of reliable nodes. However, these systems are still not maximally available. In contrast, decentralized data replication systems like Bayou [76, 89, 92], Ficus [42, 80], Grapevine [17], Quicksilver [84, 93], Coda [51], PRACTI [16], TierStore [35], and Cimbiosys [79] are maximally available, even when connectivity is intermittent. Some of the approaches in these systems are used in decentralized repositories, such as git [39], Mercurial [69], DARCS [33], and Pastwatch [101]. Unlike our cloud service environment, nodes typically keep a full copy of the entire history and trust the nodes from which they fetch updates; nothing prevents or guarantees detection of a node that issues forking writes or sends different independent subsets of writes to the nodes that fetch updates from it.

**Medium fault-tolerance, medium availability.** Another class of work provides safety even when only a subset (for example, 2/3 of the nodes) is trustworthy. However, the price for this increased fault tolerance compared to the prior category is decreased availability: to complete, an operation must reach a quorum of nodes. Such systems include Byzantine-Fault Tolerant replicated state machines (BFT RSMs) [9, 12, 24, 28, 29, 32, 41, 45, 47, 53, 54, 85, 95, 99, 100] and Byzantine Quorums [65, 66]. Note that researchers are keenly interested in reducing trust: compared to the classic BFT RSM literature, the recently-proposed A2M [26], Trinc [60], BFT2F [62], and Bonafide [27] all tolerate more failures, the former two by assuming trusted hardware and the latter two by weakening guarantees. However, unlike Depot, these systems still have a fault threshold, and none works disconnectedly.

**High fault-tolerance, low availability.** The next category further reduces the assumptions needed for safety. In SUNDR [61], FAUST [20], and related systems [18, 21, 22, 64], the server is totally untrusted, yet even under faults provides a safety guarantee: fork-linearizability, fork-sequential consistency, etc. [74]. All of these systems severely impose on availability. First, in benign runs, their admittedly stronger semantics (versus Depot's causal consistency) simplifies detecting forks (since all correct nodes operate in lock-step) but means that they cannot be available during a network partition or server failure. Second, after a forking attack, nodes are stranded on different forks and cannot talk to each other. We view this as an unacceptable encroachment on availability: to truly tolerate faulty nodes means to not stop the system when it experiences a fault.



المنارة للاستشارات
www.manaraa.com

**Other systems.** A number of other systems have sought to minimize trust for correctness and availability. We now cover these briefly. None of them gives a correctness guarantee under arbitrary faults. We stand in solidarity with Zeno [88], whose motivation is similar to ours. However, Zeno does not come close to operating with maximum availability or minimal trust assumptions: it assumes $f+1$ available servers per partition, where $f$ is the number of faulty servers. PeerReview [43] is geared to the distributed case, but it requires a quorum of witnesses at all times, one of which must be correct (a trust requirement that Depot does not have), and all of which must have complete information (which hinders availability). FARSITE [10] targets decentralized environments and seeks to minimize trust; nevertheless, it uses BFT RSM techniques so, unlike Depot, does not minimize trust for correctness or availability.

Other decentralized systems have, like Depot, been designed to resist large-scale correlated failures. Glacier [44] can tolerate a high threshold of faulty nodes. However, it cannot tolerate more than this threshold, and it stores only immutable objects. OceanStore [55] is designed to minimize trust for durability but does not tolerate nodes that fail perniciously. The S2D2 peer-to-peer storage system [49] uses a mechanism like Depot's history hash, but S2D2 cannot prevent faulty nodes from exposing correct nodes to arbitrary subsets of writes. TimeWeave [67] is designed to provide a sensible ordering of distributed events, but it also does not tolerate the most pernicious behavior: a faulty node can undetectably expose two different histories to two different nodes, so TimeWeave cannot provide fork-causal consistency.

# 8 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 everyone stores a full copy of the data, then nodes must rely on one another for durability and availability. Nonetheless, we believe that Depot appears to significantly expand the boundary of the possible by demonstrating how to *eliminate trust assumptions for safety* and to *minimize trust assumptions for liveness*. In particular, we show that providing strong safety guarantees need not imperil liveness, and we show how to leverage a novel consistency semantic (Fork-Join Causal consistency) to provide other useful properties.


## Acknowledgments

We thank Marcos K. Aguilera, Hari Balakrishnan, Brad Karp, and David Mazières for insightful comments. We also thank Utah Emulab. This work was supported by ONR grant N00014-09-10757, AFOSR grant FA9550-10-1-0073, and NSF grant CNS-0720649.


# References


[1] Amazon EC2 pricing. http://aws.amazon.com/ec2/pricing.

[2] Amazon S3 pricing. http://aws.amazon.com/s3/pricing.

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

[4] The Apache Cassandra project. http://cassandra.apache.org.

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

[6] 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.

[7] US secret service report on insider attacks. http://www.sei.cmu.edu/about/press/insider-2005.html, 2005.

[8] Oracle JRockit JVM . www.oracle.com/technology/products/jrockit, 2010.

[9] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant servers. In *SOSP*, 2005.

[10] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *OSDI*, 2002.

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

[12] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine Replication Under Attack. In *DSN*, 2008.

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

[14] L. Bassham and W. Polk. Threat assessment of malicious code and human threats. Technical report, National Institute of Standards and Technology Computer Science Division, 1994.

[15] C. Beckmann. Google app engine: Information regarding 2 July 2009 outage. http://groups.google.com/group/google-appengine/browse_thread/thread/e9237fc7b0aa7df5/ba95ded980c8c179, July 2009.

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

[17] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An Exercise in Distributed Computing. *CACM*, 25(4), 1982.

[18] C. Cachin. From Byzantine-Tolerant to Intrusion-Safe Services. In *BFTW3*, 2009.

[19] C. Cachin and M. Geisler. Integrity protection for revision control. In *Applied Cryptography and Network Security (ACNS)*, 2009.

[20] C. Cachin, I. Keidar, and A. Shraer. Fail-Aware Untrusted Storage. In *DSN*, 2009.




المنارة للاستشارات
www.manaraa.com

[21] C. Cachin, I. Keidar, and A. Shraer. Trusting the cloud. *SIGACT News*, 2009.
[22] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *PODC*, 2007.
[23] M. Calore. Ma.gnolia suffers major data loss, site taken offline. In *Wired*, Jan. 2009.
[24] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4), 2002.
[25] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
[26] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In *SOSP*, 2007.
[27] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Tiered Fault Tolerance for Long-Term Integrity. In *FAST*, 2009.
[28] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *SOSP*, 2009.
[29] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, 2009.
[30] B. Cook. 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`, July 2009.
[31] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!s Hosted Data Serving Platform. In *VLDB*, 2008.
[32] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum procol for byzantine fault tolerance. In *OSDI*, 2006.
[33] DARCS. `http://darcs.net/`.
[34] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.
[35] M. Demmer, B. Du, and E. Brewer. TierStore: a distributed filesystem for challenged networks in developing regions. In *FAST*, 2008.
[36] D. Durkee. Why cloud computing will never be free. *ACM Queue*, Apr. 2010.
[37] M. Frigo and V. Luchangco. Computation-Centric Memory Models. In *SPAA*, 1998.
[38] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In *ACM SIGACT News, 33(2)*, 2002.
[39] Git: The fast version control system. `http://git-scm.com/`.
[40] J. Gray and P. Shenoy. Rules of Thumb in Data Engineering. In *"Data Engineering"*, pages 3–12, 2000.
[41] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. In *Eurosys*, 2010.
[42] R. Guy, J. Heidemann, W. Mak, T. Page, G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Summer*, 1990.
[43] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.
[44] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.
[45] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-Overhead Byzantine Fault-Tolerant Storage. In *SOSP*, 2007.
[46] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TPLS*, 1990.
[47] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *NSDI*, 2008.
[48] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM TOCS*, 6(1):51–81, Feb. 1988.
[49] B. Kang. *S2D2: A framework for scalable and secure optimistic replication*. PhD thesis, UC Berkeley, Oct. 2004.
[50] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, 1997.
[51] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM TOCS*, 10(1):3–5, Feb. 1992.
[52] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *USENIX Technical*, 2007.
[53] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.
[54] R. Kotla and M. Dahlin. High-throughput Byzantine fault tolerant services. In *DSN*, 2004.
[55] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ASPLOS*, 2000.
[56] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7), July 1978.
[57] L. Lamport, R. Shostack, and M. Pease. The Byzantine Generals Problem. *ACM TPLS*, 4(3):382–401, 1982.
[58] B. Lampson. Hints for computer system design. *ACM OSR*, 15(5):33–48, Oct. 1983.
[59] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM TOCS*, 10(4):265–310, Nov 1992.
[60] D. Levin, J. R. Douceur, J. R. Lorch, and



المنارة للاستشارات
www.manaraa.com

T. Moscibroda. TrInc: small trusted hardware for large distributed systems. In *NSDI*, 2009.
[61] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.
[62] J. Li and D. Mazieres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.
[63] P. Mahajan, S. Lee, J. Zheng, L. Alvisi, and M. Dahlin. ASTRO: Autonomous and trustworthy data sharing. Technical Report TR-09-29, University of Texas at Austin, Department of Computer Sciences, Oct. 2008.
[64] M. Majuntke, D. Dobre, M. Serafini, and N. Suri. Abortable fork-linearizable storage. In *OPODIS*, 2009.
[65] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.
[66] D. Malkhi and M. K. Reiter. Secure and Scalable Replication in Phalanx. In *SRDS*, Oct. 1998.
[67] P. Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford, 2003.
[68] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, 1999.
[69] Mercurial. `http://mercurial.selenic.com/`.
[70] R. Miller. FBI siezes servers at dallas data center. `http://www.datacenterknowledge.com/archives/2009/04/03/fbi-seizes-servers-at-dallas-data-center/`, Apr. 2009.
[71] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *FAST*, Berkeley, CA, USA, 2004. USENIX Association.
[72] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *NSDI*, 2006.
[73] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USITS*, 2003.
[74] A. Oprea and M. Reiter. On consistency of encrypted files. In *DISC*, 2006.
[75] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, S. Kiser, D. Edwards, and C. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE TSE*, SE-9(3):240–247, May 1983.
[76] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *SOSP*, 1997.
[77] E. Pinheiro, W. Weber, and L. Barroso. Failure trends in a large disk drive population. In *FAST*, Feb. 2007.
[78] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. Gunawi, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file system. In *SOSP*, 2005.
[79] V. Ramasubramanian, T. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *NSDI*, 2009.
[80] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *USENIX Summer*, 1994.
[81] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM TOCS*, 2(4):277–288, 1984.
[82] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Otir. Deciding when to forget in the elephant file system. In *SOSP*, pages 110–123, 1999.
[83] A. Schiper, K. Birman, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM TOCS*, 9(3), 1991.
[84] F. Schmuck and J. Wyllie. Experience with Transactions in Quicksilver. In *SOSP*, 1991.
[85] S. Sen, W. Lloyd, and M. Freedman. Prophecy: Using history for high-throughput fault tolerance. In *NSDI*, 2010.
[86] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, 2008.
[87] M. Shah, M. Baker, J. Mogul, and R. Swaminathan. Auditing to keep online storage services honest. In *HotOS* , 2007.
[88] A. Singh, P. F. P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent byzantine fault tolerance. In *NSDI*, Apr. 2009.
[89] M. Spreitzer, M. Theimer, K. Petersen, A. Demers, and D. Terry. Dealing with server corruption in weakly consistent, replicated data systems. In *MOBICOM*, 1997.
[90] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1), 2003.
[91] B. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *ICPDS*, 1994.
[92] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, 1995.
[93] M. Theimer, L.-F. Cabrera, and J. C. Wyllie. QuickSilver support for access to data in large, geographically dispersed systems. In *ICDCS*, 1989.
[94] US Secret Service report on insider attacks. `http://www.sei.cmu.edu/about/press/insider-2005.html`, 2005.
[95] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, 2007.
[96] W. Vogels. Life Is Not A State-Machine: The Long Road From Research To Production. In *PODC*, 2006.
[97] White, Lepreau, Stoller, Ricci, Guruprasad, Newbold, Hibler, Barb, and Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, Dec. 2002.
[98] T. Wobber, T. L. Rodeheffer, and D. B. Terry. Policy-based access control for weakly consistent replication. In *EuroSys*, New York, NY, USA, 2010. ACM.
[99] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. ZZ: Cheap practical BFT using virtualization. Technical Report TR14-08, University of



المنارة للاستشارات
www.manaraa.com

Massachusetts, 2008.
[100] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP*, 2003.
[101] A. Yip, B. Chen, and R. Morris. Pastwatch: A distributed version control system. In *NSDI*, 2006.

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

**Definition 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 $oId$ .
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 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 executions it allows. In particular, showing than an *execution is consistent* under some semantics simply requires to show that an oracle can produce an edge assignment that passes the consistency check; showing instead that a *system enforces* some consistency semantics requires presenting an algorithm that for every execution of the system constructs an edge assignment that passes the consistency check.

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

**Definition 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 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.*

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:

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

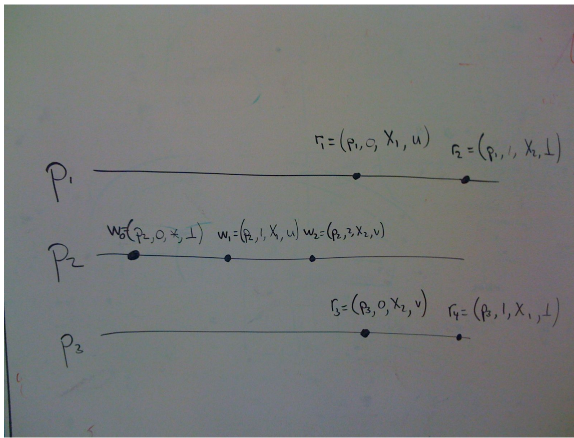
***Comparison with causal consistency.*** *Causal consistency* enforces conditions that are analogous to the one enforced by FJC, but it requires them to hold for operations issued by *all* nodes—not just correct nodes. Specifically, an execution α is said to be *causally consistent* if there exists 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 correct 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:

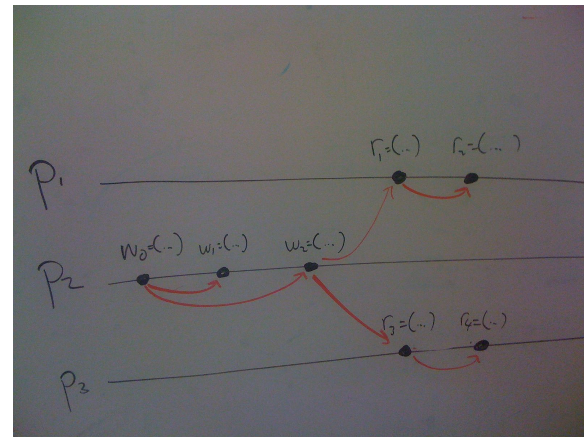
$$w \in wl \Leftrightarrow w \prec_G r \wedge \nexists w' : w \prec_G w' \prec_G r$$



المنارة للاستشارات
www.manaraa.com




(a) Execution




(b) Observer graph

FIG. 19—An execution with a faulty node $p_2$ and its corresponding observer graph. The observer graph is not causally consistent because $w_1$ and $w_2$ are not ordered according to the history of node $p_2$. The observer graph is, however, FJC consistent because $p_2$ is faulty and therefore FJC consistency doesn't require total ordering of $p_2$'s operations.

Figure 19-(a) shows an execution that is FJC consistent but not causally consistent. In this example, node $p_2$ is faulty and produces two writes $w_1$ and $w_2$. Node $p_1$ observes $w_1$ but not $w_2$, and node $p_3$ observes $w_2$ but not $w_1$. As Figure 19-(b) illustrates, we can produce an edge assignment and observer graph that passes all FJC tests by violating 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.



المنارة للاستشارات
www.manaraa.com