Published in proceedings of DSN 2010, The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 28 - July 1, 2010, Chicago Illinois USA

Efficient eventual consistency in Pahoehoe, an erasure-coded key-blob archive

Eric Anderson, Xiaozhou Li, Arif Merchant, Mehul A. Shah,
Kevin Smathers, Joseph Tucek, Mustafa Uysal, Jay J. Wylie
Hewlett-Packard Laboratories

Abstract

Cloud computing demands cheap, always-on, and reliable storage. We describe Pahoehoe, a key-value *cloud storage system we designed to store large objects cost-effectively with high availability. Pahoehoe stores objects across multiple data centers and provides eventual consistency so to be available during network partitions. Pahoehoe uses erasure codes to store objects with high reliability at low cost. Its use of erasure codes distinguishes Pahoehoe from other cloud storage systems, and presents a challenge for efficiently providing eventual consistency.*

We describe Pahoehoe's put, get, and convergence *protocols—convergence being the decentralized protocol that ensures eventual consistency. We use simulated executions of Pahoehoe to evaluate the efficiency of convergence, in terms of message count and message bytes sent, for failure-free and expected failure scenarios (e.g., partitions and server unavailability). We describe and evaluate optimizations to the naïve convergence protocol that reduce the cost of convergence in all scenarios.*

1. Introduction

Cloud computing offers the promise of always-on, globally accessible services that lower total cost of ownership. To meet this promise, cloud services must run on low-cost and highly available infrastructure. High availability means offering responsive service even in the face of multiple simultaneous failures (e.g., node crashes or partitions) to clients in diverse geographic regions. For many cloud applications, like social networking or photo sharing, available storage is paramount and, given their scale (and cut-rate cost constraint) partitions are inevitable. Unfortunately, Brewer's *CAP Theorem* [13] states that only two of consistency, availability, and partition-tolerance are simultaneously achievable. Hence cloud storage systems cannot provide the same consistency semantics as traditional storage systems do and still be available.

In light of these constraints, we designed Pahoeoe, a highly available, low-cost, and scalable key-value store. Pahoehoe offers a get-put interface similar to Amazon's S3 service. The get and put methods take a key (unique application-provided name) as a parameter that specifies the object to retrieve or store into the system, respectively. Because of the implications of the CAP Theorem, we designed Pahoehoe to offer eventual consistency so that it achieves high availability and partition-tolerance. Further, to achieve high reliability and availability at low cost, Pahoehoe stores objects using erasure coding. Erasure codes enable space-efficient fault-tolerant storage, but they require careful implementation to avoid using more network bandwidth to propagate data than a replica-based system. To the best of our knowledge, Pahoehoe is the first distributed erasure-coded storage system that provides eventual consistency.

In this paper, we describe the put and get protocols for storing and retrieving erasure-coded objects in Pahoehoe. We also describe *convergence*, the decentralized protocol that provides eventual consistency. We first describe naïve convergence which is simple and robust, but potentially inefficient. Then we describe extensions to make convergence efficient—both in terms of message bytes and message counts sent.

We evaluate the various convergence optimizations in various failure scenarios. Pahoehoe achieves network efficiency and low message counts in the common, failure-free case. Under failure scenarios in which some servers are unavailable for some period of time, Pahoehoe incurs a roughly constant overhead regardless of the severity of the failure. Under failure scenarios in which the network is lossy, the work convergence does to achieve eventual consistency increases commensurate with the loss rate.

2. Design

Pahoehoe is a key-value store tailored for binary large objects such as pictures, audio files or movies of moderate size ($\sim 100 \times 2^{10}$ Bytes (B) to 100×2^{20} B). Pahoehoe exports two interfaces for clients: **put**$(key, value, policy)$

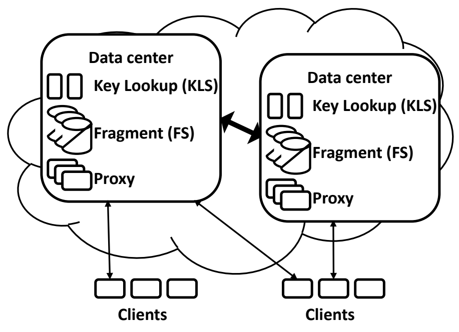

Figure 1. Pahoehoe architecture.

and **get**(key). The put interface allows a client to associate a value with the object identified by the key. The policy specifies durability requirements for the stored value. Pahoehoe allows different put operations to specify the same key, i.e., multiple *object versions* may be associated with the same key. Different object versions associated with the same key are distinguished by a unique Pahoehoe-assigned timestamp. The get interface allows a client to retrieve an object version associated with the specified key. Although Pahoehoe will attempt to retrieve the most recent object version, because Pahoehoe is eventually consistent, there may be multiple versions that it can safely return.

The high-level architecture of Pahoehoe is illustrated in Figure 1. Clients use a RESTful interface [10] to interact with a proxy server at a data center which performs get and put operations on behalf of the client. Pahoehoe itself has two types of servers: Key Lookup Servers (KLSs) and Fragment Servers (FSs). Key Lookup Servers store a metadata list of (timestamp, policy, locations) tuples which maps a key to its object versions. The locations list the FSs that store fragments of the object version. Pahoehoe's separation of metadata (KLS) and data (FS) servers is similar to that of object-store-based file systems such as NASD [12].

Pahoehoe provides several properties: high availability, durability, and eventual consistency. It provides availability by permitting a client to put and get objects even when many servers have crashed or the network is partitioned, either WAN or LAN [6]. Even if a proxy can only reach a minority of KLSs and FSs, a put or a get may complete successfully.

By durability, we mean that an object version can be recovered even if many of the servers are crashed. Pahoehoe uses erasure codes to achieve durability at reduced storage cost. An erasure code encodes a value into $n = k + m$ fragments such that any k fragments can be used to recover the object. Pahoehoe uses a *systematic* Reed-Solomon erasure code [16] in which a value is striped across the first k *data fragments*, with the remaining m being *parity fragments*. Modern erasure code implementations are sufficiently efficient [19] that we believe encoding and decoding can be performed fast enough to meet our performance requirements. We refer to the erasure-coded fragments of an object version as *sibling fragments*, and FSs that host sibling fragments as *sibling FSs*.

A durability policy can be specified for each put operation. The default policy is a $(k = 4, n = 12)$ erasure code with up to 2 fragments per FS, 6 fragments per data center, and all 4 data fragments at the same data center. This policy has the same storage overhead as triple replication, but can tolerate many more failure scenarios: up to eight simultaneous disk failures; or a network partition between data centers in conjunction with either two simultaneous disk failures or a single unavailable FS.

Once the complete metadata and all the sibling fragments for an object version have been stored at all KLSs and sibling FSs (respectively), we say that the object version is *at maximum redundancy* (AMR). To ensure that object versions put into Pahoehoe eventually achieve AMR, each FS runs *convergence*. Convergence is a decentralized gossip-like protocol in which each sibling FS repeatedly and independently attempts to make progress towards achieving AMR for each object version, until AMR is achieved. The AMR property dictates Pahoehoe's eventual consistency guarantee: once an object version is AMR, a subsequent get will not return any prior object version.

Unfortunately, eventual consistency—a necessity for Pahoehoe to be available during network partitions—requires sibling FSs to propagate or recover erasure-coded fragments. "Gossiping" erasure-coded fragments among FSs is expensive relative to gossip in replica-based systems, because a sibling FS must receive k fragments to recover their sibling fragments. To avoid this bandwidth cost, proxies generate all the sibling fragments and send them to all of the sibling FSs directly. Therefore convergence is a mechanism to deal with failures rather than the common means to propagate fragments.

3. Core protocols

In this section, we describe the system model, the put, get, and naïve convergence protocols, discuss implementation details, and sketch correctness arguments. In Section 4, we describe optimizations to naïve convergence that reduce its demands on message bytes and message counts.

3.1. System model

Pahoehoe tolerates nodes (clients, proxies, FSs, and KLSs) that crash and recover [2]. Pahoehoe tolerates benign failures; it does not tolerate Byzantine failures. We assume that servers have stable storage that persists throughout the crash and recover process. While our protocols can rebuild destroyed disks or nodes, and detect disk corruption using hashes, we do not discuss these due to space limitations.

Published in proceedings of DSN 2010, The 40th Annual IEEE/IFIP International Conference
on Dependable Systems and Networks, June 28 - July 1, 2010, Chicago Illinois USA

Informally, we assume that eventually there will be a period in which all nodes are available and during which messages between clients, proxies, KLSs, and FSs are delivered successfully within some time bound. More formally, we assume a partially synchronous system model [9] and point-to-point channels with fair losses and bounded message duplication [2].

We assume that nodes have access to a local clock for the purposes of scheduling periodic tasks and for tuning policies like backoff and probing. We also assume that proxies have access to a loosely synchronized clock, that can order concurrent put operations to the same key. Pahoehoe orders concurrent puts in the order they were received, subject to the synchronization limits of NTP [17]. This order matches users' expected order for partitioned data centers when they happen to access different ones during the partition.

3.2. Put protocol

The basic put protocol proceeds in two rounds. Upon receiving a **put**$(key, value, policy)$ request from a client, the proxy selects an object version, and asks all KLSs to suggest potential fragment locations (i.e., a list of FSs for their data center) for it. We assume the set of all KLSs (**klss**) is known be every proxy and FS. The KLSs use method **which_locs** to interpret the policy and to balance load and capacity across the FSs. The proxy decides which fragment locations to use in each data center based on the first KLS response from that data center. The proxy then sends the metadata (i.e., policy and locations) for the object version to all of the KLSs, and sends the metadata and appropriate sibling fragments to each sibling FS. After receiving replies (or timeouts) from all the servers (KLSs and FSs), the proxy replies back to the client.

Although simple and correct, the above put protocol may suffer long latency under some conditions. For example, during a network partition, the proxy will wait for timeouts in both rounds. Two optimizations reduce latency during failures. First, the proxy sends out metadata with partial locations and the appropriate sibling fragments as soon as the locations for any data center (rather than all data centers) are decided. Second, the proxy reports success to the client as soon as it receives enough (specified by the policy) successful replies from FSs.

Figure 2 presents pseudocode for the put operation with these optimizations. Each KLS maintains two persistent data structures: a timestamp store ($store_{ts}$) which maps an object, uniquely identified by a key, to object versions, each of which is uniquely identified by ov, a (key, timestamp) pair; and a metadata store ($store_{meta}$), which maps an object version to its metadata, a (policy, locations) pair. Each FS maintains two persistent data structures: a metadata store, which it uses for convergence, and a fragment store ($store_{frag}$) which maps an object version to its meta-

```
Proxy server proxy
 1: $meta \leftarrow \bot$; $\mathbf{frags} \leftarrow \emptyset$; $\mathbf{locs} \leftarrow \emptyset$; $\mathbf{resps} \leftarrow \emptyset$
 2: upon receive put($key, value, policy$) from $client$
 3:     $ts \leftarrow$ now(); $ov \leftarrow (key, ts)$; $meta.policy \leftarrow policy$
 4:     $\mathbf{frags} \leftarrow$ encode($value, meta.policy$)
 5:     $\forall kls \in \mathbf{klss}$ : send decide_locs($ov, meta.policy$) to $kls$
 6: upon receive decide_locs reply($ov, \mathbf{locs}$) from $kls$
 7:     if useful_locs($meta, \mathbf{locs}$) then
 8:         $meta.\mathbf{locs} \leftarrow meta.\mathbf{locs} \cup \mathbf{locs}$
 9:         $\forall kls \in \mathbf{klss}$ : send store($ov, meta$) to $kls$
10:         $\forall fs \in meta.\mathbf{locs}$ : send store($ov, meta, \mathbf{frags}[fs]$) to $fs$
11: upon receive store reply($ov, status$) from $server$
12:     $\mathbf{resps} \leftarrow \mathbf{resps} \cup \{(server, status)\}$
13:     if can_reply($\mathbf{resps}, meta$) then
14:         send put reply(reply_status($\mathbf{resps}, meta$)) to $client$
Key Lookup Server kls
 1: $store_{ts} \leftarrow \emptyset$; $store_{meta} \leftarrow \emptyset$
 2: upon receive decide_locs($ov, policy$) from $proxy$
 3:     $\mathbf{locs} \leftarrow$ which_locs($ov, policy$)
 4:     send decide_locs reply($ov, \mathbf{locs}$) to $proxy$
 5: upon receive store($ov, meta$) from $proxy$
 6:     $store_{ts}[ov.key] \leftarrow store_{ts}[ov.key] \cup \{ov.ts\}$
 7:     $\mathbf{locs} \leftarrow store_{meta}[ov].\mathbf{locs} \cup meta.\mathbf{locs}$
 8:     $store_{meta}[ov] \leftarrow (meta.policy, \mathbf{locs})$
 9:     send store reply($ov, success$) to $proxy$
Fragment Server fs
 1: $store_{frag} \leftarrow \emptyset$; $store_{meta} \leftarrow \emptyset$
 2: upon receive store($ov, meta, frag$) from $proxy$
 3:     $\mathbf{locs} \leftarrow store_{meta}[ov].\mathbf{locs} \cup meta.\mathbf{locs}$
 4:     $store_{meta}[ov] \leftarrow (meta.policy, \mathbf{locs})$
 5:     $store_{frag}[ov] \leftarrow (store_{meta}[ov], frag)$
 6:     send store reply($ov, success$) to $proxy$
```

Figure 2. Put operation.

data and sibling fragment. Each proxy constructs a globally unique timestamp by concatenating the time from the loosely synchronized local clock with its own unique identifier ($proxy$ line 3). Locations from a KLS are considered useful if they are the first locations that a proxy receives for a data center ($proxy$ line 7). The function **can_reply** ($proxy$ line 13) returns true if, according to the given policy, enough fragments have been durably stored.

3.3. Get protocol

The basic get protocol also uses two rounds. Upon receiving a **get**(key) request from a client, the proxy first asks all KLSs for all of the object versions (timestamps) with the associated metadata for the object identified by the key. After collecting the object versions with metadata from all available KLSs, the proxy attempts to retrieve the object versions in timestamp order (i.e., from latest to earliest). Two optimizations reduce the latency during failures. First, the proxy starts retrieving fragments for the latest object version it identifies as soon as it hears back from any KLS. This optimization will not violate consistency because every KLS has the timestamp with complete metadata for an AMR object version (if any). Second, the proxy starts re-

Published in proceedings of DSN 2010, The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 28 - July 1, 2010, Chicago Illinois USA

trieving an earlier object version as soon as it determines that the current object version cannot be retrieved, and that it is safe (explained below) to try an earlier object version.

Figure 3 lists the pseudocode of the get protocol with these optimizations. Method **can_decode** (*proxy* line 16) returns true if sufficient sibling fragments for the current object version have been retrieved. Method **can_try_earlier** (*proxy* line 19) returns true if the proxy can safely try to retrieve an earlier object version. In particular, it returns true if, for the object version being retrieved, any KLS returned incomplete metadata (*proxy* line 6) or any FS returned a $\perp$ fragment (*proxy* line 13), implying that the current version is not AMR. Once all object versions that can be safely tried have been tried unsuccessfully, and object versions with metadata have been received from all KLSs, the proxy has to return failure to the client (*proxy* line 28).

3.4. Naïve convergence

In a failure-free execution, an object version is AMR when a proxy completes the put. Failures such as message drops, network partitions, or server unavailability result in an object version not being AMR when the proxy stops work on the put. Convergence ensures that all object versions for which sufficient fragments are durably stored eventually reach AMR.

Each FS runs convergence independently in periodic *rounds*. During each round, a convergence *step* is performed for each object version the FS has not yet verified is AMR. In a convergence step, an FS verifies the following: 1) it has complete metadata (i.e., sufficient locations to meet the durability requirements specified in the policy); 2) it stores the appropriate sibling fragment locally; 3) all KLSs store complete metadata for the object version; and 4) all sibling FSs store verified metadata and sibling fragments. If verification is successful, then the object version is AMR and the FS excludes it from subsequent convergence rounds. If the FS has incomplete metadata, it acts somewhat like a proxy performing a put and asks a KLS to suggest locations for the object version. If the FS does not store its fragment locally, it performs a get of the desired object version to retrieve sibling fragments so that it can generate its missing fragment. By repeatedly performing convergence steps on an object version with sufficient durably stored fragments, it will eventually be AMR.

Figure 4 presents the pseudocode for the naïve convergence protocol. The method **recover_fragment** (*fs* line 8) is a get operation that only retrieves the specified object version. Given that object version, the FS generates its missing fragment via erasure coding. The method **verify** (*fs* line 5 and *kls* line 4) verifies that metadata is complete (has sufficient locations as per the policy); when invoked on a (metadata, fragment) pair, it verifies that the metadata is complete and that the fragment is not $\perp$ (*fs* line 22). The method

```
Proxy server $proxy$
 1: $key \leftarrow \perp; ts \leftarrow \perp; meta \leftarrow \perp$
 2: $\mathbf{tss} \leftarrow \emptyset; \mathbf{resps_{kls}} \leftarrow \emptyset; \mathbf{resps_{fs}} \leftarrow \emptyset$
 3: upon receive get($key'$) from $client$
 4:    $key \leftarrow key'$
 5:    $\forall kls \in \mathbf{klss}$ : send retrieve_ts($key$) to $kls$
 6: upon receive retrieve_ts reply($\mathbf{tss'}, \mathbf{metas}$) from $kls$
 7:    for all $ts' \in \mathbf{tss'}$ do
 8:       $\mathbf{locs} \leftarrow \mathbf{resps_{kls}}[ts'].\mathbf{locs} \cup \mathbf{metas}[ts'].\mathbf{locs}$
 9:       $\mathbf{resps_{kls}}[ts'] \leftarrow (\mathbf{metas}[ts'].policy, \mathbf{locs})$
10:    $\mathbf{tss} \leftarrow \mathbf{tss} \cup \mathbf{tss'}$
11:    if $ts = \perp$ then
12:       next_ts()
13: upon receive retrieve_frag reply($ts', frag$) from $fs$
14:    if $ts = ts'$ then
15:       $\mathbf{resps_{fs}}[ts] \leftarrow \mathbf{resps_{fs}}[ts] \cup \{(fs, frag)\}$
16:       if can_decode($meta, \mathbf{resps_{fs}}[ts]$) then
17:          $value \leftarrow$ decode($meta, \mathbf{resps_{fs}}[ts]$)
18:          send get reply($success, value$) to $client$
19:       else if can_try_earlier($meta, \mathbf{resps_{fs}}[ts]$) then
20:          next_ts()
21: upon next_ts()
22:    $ts \leftarrow$ max($\mathbf{tss}$)
23:    if $ts \neq \perp$ then
24:       $\mathbf{tss} \leftarrow \mathbf{tss} \setminus \{ts\}$
25:       $ov \leftarrow (key, ts); meta \leftarrow \mathbf{resps_{kls}}[ts].meta$
26:       $\forall fs \in meta.\mathbf{locs}$ : send retrieve_frag($ov$) to $fs$
27:    else if all_kls_replied($\mathbf{resps_{kls}}$) then
28:       send get reply($failure, \perp$) to $client$
Key Lookup Server $kls$
 1: upon receive retrieve_ts($key$) from $proxy$
 2:    $\mathbf{metas} \leftarrow \{store_{meta}[(key, ts)] : ts \in store_{ts}[key]\}$
 3:    send retrieve_ts reply($store_{ts}[key], \mathbf{metas}$) to $proxy$
Fragment Server $fs$
 1: upon receive retrieve_frag($ov$) from $proxy$
 2:    send retrieve_frag reply($ov.ts, store_{frag}[ov].frag$) to $proxy$
```

Figure 3. Get operation.

is_amr (*fs* line 25) confirms that all KLSs and sibling FSs have replied with success in response to the converge requests. Once the object version is determined to be AMR, the FS removes it from $store_{meta}$ so that it does no further work in future convergence rounds for this object version.

3.5. Discussion

We have elided many implementation details, including some optimizations, from the descriptions of the core protocols. For example, during a put, the proxy iteratively retrieves timestamps with associated metadata from KLSs instead of retrieving information about all object versions at once. As another example, a location in Pahoehoe actually identifies both an FS and a disk on that FS so that multiple sibling fragments may be collocated on the same FS. Three topics do warrant further discussion though: 1) proxy timeouts and return codes; 2) exceeding the locations needed by the policy; and 3) object versions with insufficient durably stored fragments.

A client may timeout waiting for a proxy to return from

Published in proceedings of DSN 2010, The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 28 - July 1, 2010, Chicago Illinois USA

```
Fragment Server $fs$
 1: upon start_round()
 2:   resps ← $\emptyset$
 3:   for all $ov \in store_{meta}$ do
 4:     $(meta, frag) \leftarrow store_{frag}[ov]$
 5:     if $\neg$verify($meta$) then
 6:       $\forall kls \in$ klss : send decide_locs($ov, meta.policy$) to $kls$
 7:     else if $frag = \bot$ then
 8:       recover_fragment($ov$)
 9:     else
10:       $\forall kls \in$ klss : send converge($ov, meta$) to $kls$
11:       $\forall fs \in meta.$locs : send converge($ov, meta$) to $fs$
12: upon receive decide_locs reply($ov$, locs$'$) from $kls$
13:   if useful_locs($store_{meta}[ov].meta$, locs$'$) then
14:     $store_{meta}[ov] \leftarrow store_{meta}[ov]$.locs $\cup$ locs$'$
15:     $store_{frag}[ov].meta \leftarrow store_{meta}[ov]$
16: upon receive converge($ov, meta$) from $fs'$
17:   if $ov \notin store_{meta} \wedge ov \notin store_{frag}$ then
18:     $store_{meta}[ov] \leftarrow meta$; $store_{frag}[ov] \leftarrow (meta, \bot)$
19:   else if $ov \in store_{meta}$ then
20:     $store_{meta}[ov]$.locs $\leftarrow store_{meta}[ov]$.locs $\cup$ $meta$.locs
21:     $store_{frag}[ov].meta \leftarrow store_{meta}[ov]$
22:   send converge reply($ov$, verify($store_{frag}[ov]$)) to $fs'$
23: upon receive converge reply($ov, status$) from $server$
24:   resps[$ov$] ← resps[$ov$] $\cup$ $\{(server, status)\}$
25:   if is_amr(resps[$ov$], $store_{meta}[ov]$) then
26:     $store_{meta} \leftarrow$ remove($store_{meta}, ov$)
Key Lookup Server $kls$
 1: upon receive converge($ov, meta$) from $fs$
 2:   locs ← $store_{meta}[ov]$.locs $\cup$ $meta$.locs
 3:   $store_{meta}[ov] \leftarrow (meta.policy$, locs$)$
 4:   send converge reply($ov$, verify($store_{meta}[ov]$)) to $fs$
```

Figure 4. Naïve convergence.

either a put or a get. This timeout is necessary because proxies may crash and so the client must handle such timeouts. Beyond this, a proxy may choose to return "unknown" in response to a put request after some amount of time. This response is effectively handled like a timeout by the client. Because of the nature of distributed systems, the proxy may not know whether or not certain fragment store requests arrived at certain FSs and so cannot know whether sufficient fragments have been durably stored to meet the policy. There is a similar difficulty for get operations: after some time, if neither **can_decode** nor **can_try_earlier** returns true, then the proxy must timeout or return failure.

It is possible for the locations of an object version to exceed the policy, that is, for there to be "too many" locations. There are two ways in which this may occur: an FS doing a convergence step sends a KLS a **decide_locs** message concurrent to the proxy doing the same, or concurrent to another sibling FS doing the same. If this happens, it is a form of inefficiency (too many sibling fragments end up being stored); it does not affect correctness. To reduce the chance of this happening, every FS probes KLSs in each data center in a specific order, unlike a proxy doing a put which broadcasts to all KLSs simultaneously. Beyond this, a KLS treats a **decide_locs** request from an FS differently than from a proxy: the KLS updates its $store_{meta}$ with the locations it suggests before replying to the FS, and sends an indication to all sibling FSs of its locations decision.

Finally, if an object version has insufficient durably stored fragments (i.e., fewer than k sibling fragments), then it can never achieve AMR. This state results in the sibling FSs invoking convergence steps forever but in vain. Exponential backoff is used to decrease the frequency with which convergence steps are actually attempted—the older the non-AMR object version, the longer before a convergence step is tried again. Beyond this, Pahoehoe can be configured to stop trying convergence steps after some time has passed; in practice, we set this parameter to two months.

3.6. Correctness sketch

Due to space limitations, we provide only a rough sketch of the correctness of the Pahoehoe protocols. Pahoehoe provides eventual consistency [29] with regular semantics [15] that permits aborts (i.e., put and get operations may abort, which is similar to, but somewhat weaker than, *pseudo-regular* [18]). Consider a single key, a get operation for that key, put operations for that key, and the following definitions. The get operation *begins* the moment a proxy begins the get protocol; it *completes* when the proxy sends a reply to the client (or aborts). A put operation *begins* the moment a proxy begins the put protocol; it *completes* when the object version being put is AMR. The *latest AMR version* is the object version of the complete put with the highest timestamp at the moment the get begins. A *durable* object version is an object version for which k fragments are durably stored on FSs, as specified in the object version's policy. A *recent version* is any durable object version with a timestamp later than the latest AMR version. Assuming sufficiently synchronized clocks, a put that begins after another put completes yields a recent version.

In Pahoehoe, *regular semantics with aborts* means that the get returns a recent version, or the latest AMR version, or aborts. Note that AMR is a stable property, that is, once an object version is AMR, it remains so forever, because (1) none of the protocols ever delete metadata or fragments, and (2) a server that crashes and recovers retains its persistent state (i.e., metadata and fragments). By the definition of AMR, every KLS returns the timestamp with metadata for all AMR object versions, including the latest AMR version. Also by the definition of AMR, every sibling FS has the appropriate sibling fragment and returns it in response to a retrieve fragment request. The get operation attempts to retrieve object versions from latest to earliest by timestamp. Therefore the get operation may return a recent version, which is allowed by regular semantics. The get operation will not return an object version prior to the latest AMR version though since **can_try_earlier** always returns false if the latest AMR version is being retrieved. It returns false because all KLSs return complete metadata and all sibling FSs return the appropriate sibling fragment for the latest AMR version.

In Pahoehoe, *eventual consistency* means that if, at some point, no additional puts are issued, the recent object version with the latest timestamp will become the latest AMR version. All durable object versions eventually achieve AMR because of our assumptions of partial synchrony, fair lossy channels, and server recoveries.

4. Efficient convergence

In naïve convergence, each FS independently determines whether an object version is AMR. Such a decentralized approach, though correct and robust to failures, is inefficient. In this section we discuss extensions to convergence to make common cases more efficient. Note that we elide some simple optimizations which are not as substantial as the ones below (e.g., an FS does not actually send **converge** messages to itself in *fs* line 11 of Figure 4).

4.1. AMR Indications

In the naïve protocol, FSs may start convergence on an object version even before the put operation completes; further, every single FS individually must complete convergence, contacting every other FS. This approach is obviously wasteful; we modify the protocol to eliminate convergence steps in the common case and, when convergence steps are needed, to minimize the number of FSs that perform such steps.

In the common, failure-free case, it is likely that a proxy will completely finish a put operation and will know that the object version is AMR due to the replies it gets. To keep the FSs from doing any work during a converge step for this object version, the proxy sends an *AMR Indication* to all the sibling FSs. Upon receiving such an indication, an FS removes the object version from its $store_{meta}$ and will not perform any further convergence work. To allow the put to complete, an FS only initiates convergence on non-AMR object versions over a minimum age (currently 300 seconds). If a proxy AMR Indication is lost, the only effect is that some sibling FSs will perform an unnecessary round of convergence. However, because the failure-free case is common, this optimization significantly reduces the number of messages needed to ensure AMR.

Since sibling FSs are sent fragments for an object version at the same time, they are likely to run the corresponding convergence steps simultaneously. To improve the chance of only one sibling FS performing a sibling convergence step, convergence rounds are scheduled uniformly randomly between every 30 and 90 seconds. This randomness encourages sibling convergence steps to be unsynchronized. When an FS does determine that the object version is AMR, it sends an AMR Indication to all of its sibling FSs, so that they do not initiate convergence steps for it. Again, this indication is not necessary for correctness, and so the protocol can tolerate its loss.

4.2. Sibling fragment recovery

One of the most common failures is a network partition. In the case of a WAN partition, one entire data center will need to perform recovery after the partition heals. However, having each of the sibling FSs in that data center retrieve k fragments across the WAN to decode the object version and encode its fragment is expensive. Any FS that retrieves k fragments can recover **all** of the sibling fragments simultaneously. Having only one FS recover the missing fragments and then share the fragments over the LAN can significantly reduce WAN traffic.

In the more general case, we can have any FS that performs recovery help its siblings. In *sibling fragment recovery*, an FS that needs to recover a fragment sends converge messages to its sibling FSs with a flag set indicating its intentions. A sibling FS includes which, if any, fragments it needs recovered in its reply to the converge message. The FS performing sibling fragment recovery waits some time to accumulate replies and then recovers all of the missing fragments. The FS then sends store messages with the recovered sibling fragments to all the appropriate sibling FSs.

This optimization creates the risk of FSs performing even more work to recover fragments: if multiple FSs perform sibling fragment recovery simultaneously then they could each retrieve k fragments and store m fragments in the worst case. To prevent this from happening, FSs track whether they are currently attempting sibling fragment recovery of an object version. If they are, and they receive a converge message from a sibling FS that indicates it too intends to attempt sibling fragment recovery, then they may backoff. An FS only backs off if its unique server id is lower than the other sibling FS's unique id. This backoff extends the exponential random backoff mentioned previously in Section 3.5.

5. Evaluation

To evaluate convergence in Pahoehoe, we setup experiments, in simulation, that measure the merit of the various optimizations we added to naïve convergence, that measure the work convergence does in the face of specific failures, and that confirm object versions that achieve sufficient durability eventually become AMR. The main failures we consider are network failures: either some nodes are unreachable or some percentage of messages are lost. Such network failures can be used to simulate a server crashing (and subsequently recovering), or a proxy failing after completing only some portion of a put operation.

5.1. Experimental setup

We are primarily interested in the behavior of the protocols and how the various optimizations affect them. Hence, we measure our protocols under two criteria: the message bytes sent and the number of messages sent to reach AMR, including all activity from the proxy's put and all convergence activity. We do not measure response time or throughput because these measures depend on workload, capacity, and implementation quality, none of which we consider in this paper. We evaluate our protocols by running the Pahoehoe implementation in a simulated network environment. We use a simple performance model for the network: each message has a latency between 10 to 30 ms chosen uniformly randomly.

Our experimental workload consists of 100 puts of 100×2^{10} B objects, using the default durability policy described in Section 2: a $(k = 4, n = 12)$ erasure code with six fragments per data center and up to two sibling fragments per server. For the cases where we inject failures, we either 1) simulate a 10 minute node crash and recovery by dropping all messages in and out of that simulated node for 10 minutes, or 2) we drop (system wide) a percentage of the messages, randomly. We run all experiments until all object versions that can achieve AMR do so. Our simulated hardware consists of two data centers, with two replicated KLSs each and three FSs each. We intentionally choose a small system configuration so that, given the durability policy, any server failure affects all object versions.

We run most experiments 50 times with different random seeds and report the mean; we run the lossy network experiments 150 times. We measure the 95^{th} confidence interval to verify that all reported results are statistically significant.

5.2. Convergence in the absence of failures

In this first experiment, we evaluate the impact of optimizations in Section 4 when there are no failures. Figure 5 illustrates the results. In this experiment, we compare to an *Idealized* implementation of our protocols, i.e., one that knows this is a failure-free execution and so can send the absolute minimum number of messages to reach AMR. The Idealized results are calculated analytically as follows: one KLS per data center receives a locations request which elicits one response; the proxy sends each of the four KLSs the chosen locations to which each sends one response; it also sends each of the six FSs two store fragment requests (one for each sibling fragment) for which each FS sends one response and receives an AMR indication. In naïve convergence, every sibling FS runs a complete convergence step to determine that each object version is AMR, as a result, six times more messages are generated compared to an Idealized implementation.

Our first optimization involves each FS sending an AMR

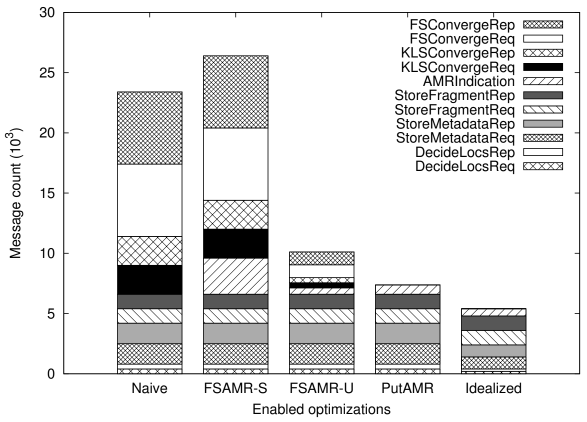

Figure 5. Failure-free execution.

indication at the end of convergence to avoid redundant work. However, when all FSs start convergence at the same time, this optimization (FSAMR-S) results in a 13% increase in the number of messages because each FS runs sibling convergence steps simultaneously **and** sends an indication to other FSs at the end. The effectiveness of this optimization hinges on each FS starting convergence at slightly different times so that the indications of the first FS executing convergence has a chance to prevent a full convergence step on the sibling FSs. FS AMR indications combined with unsynchronized start times (FSAMR-U) results in a 57% drop in message count compared to naïve convergence.

Our second optimization (PutAMR) includes the proxy sending an AMR indication at the end of the put. In the failure-free case, this avoids running any convergence steps at all, since each FS learns the AMR status from the proxy. This optimization results in 68% fewer messages compared to naïve convergence. However, it still falls short of the Idealized implementation because the proxy immediately forwards chosen locations upon receipt from a KLS in each data center, resulting in two sets of location messages and two location updates instead of one.

5.3. Convergence and server recovery

In this section, we investigate the effects of server unavailability on the performance of convergence. This failure scenario occurs either due to a server crash or due to a network partition during a put operation. We simulate both of these scenarios by using a network model that drops all the messages between designated servers during a fixed period followed by a full recovery.

First, we consider from zero to four FS failures. When there are more than one failure, the failures are roughly balanced between data centers. Figures 6 and 7 show message counts and byte counts respectively during convergence as varying numbers of FSs are unavailable during a put operation. The label below each column indicates the number

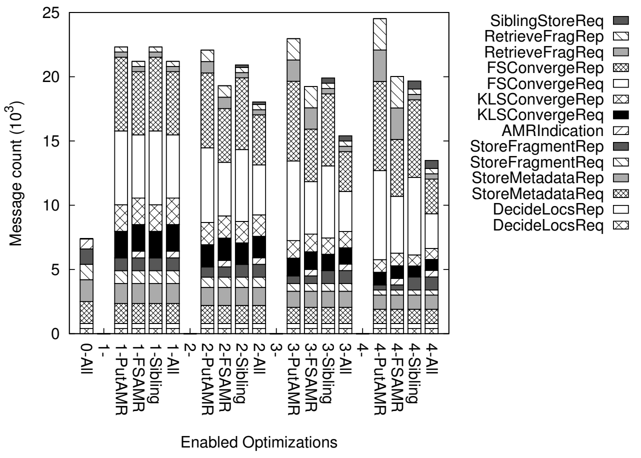

Figure 6. FS failures and message count.

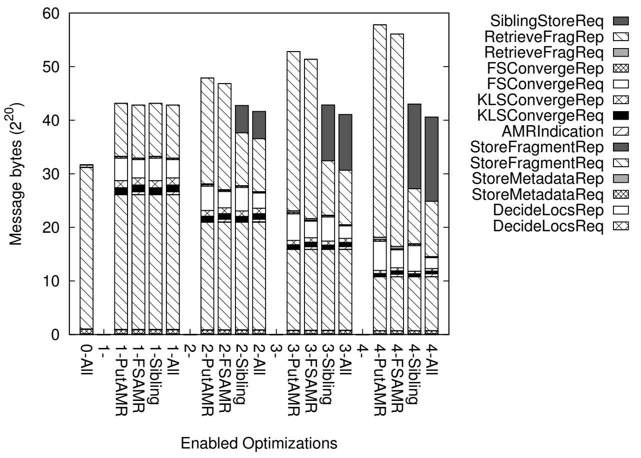

Figure 7. FS failures and message bytes.

of FSs that are unavailable (from 0 to 4) and which convergence optimizations are enabled. We consider four different optimization settings for this experiment: *PutAMR* uses the Put AMR Indication optimization; *Sibling* uses the unsynchronized sibling fragment recovery optimization; *FSAMR* uses the FS AMR Indication optimization; and *All* uses all of the convergence optimizations. The experiment *0-All* is actually the same result as PutAMR in the previous experiment (i.e., Figure 5), and is included as a reference point.

Figure 6 shows that FS failures greatly increase the number of messages exchanged during convergence. In each convergence step, an FS sends each sibling FS a converge message (FSConvergeReq) and each KLS a converge message (KLSConvergeReq), and each of these requests garners a reply. Figure 6 also shows that both the FS AMR indication and the sibling fragment recovery reduce the number of convergence messages during FS failures. The effectiveness of sibling fragment recovery increases with more FSs being unavailable as the sibling fragment recovery does a good job of preventing duplicated effort when rebuilding fragments. Note that the number of messages for convergence during FS failures depends on two factors: 1) the duration of the FS failures, and 2) the number of available FSs trying to make progress on convergence. The latter is the primary reason why the total number of messages drops as the number of unavailable FSs increase in Figure 6. Interestingly, the optimizations have a cumulative impact on message counts. This effect is good as the common recovery scenario during convergence is for a single FS to recover all needed sibling fragments, store them on the appropriate sibling FSs, and indicate to all sibling FSs that the object version is AMR.

The sibling fragment recovery optimization significantly reduces the total amount of data exchanged for convergence during FS failures. This is due to the properties of erasure codes: to recover any fragment, at least k sibling fragments must be retrieved. To use the minimal network capacity, the sibling fragment recovery optimization amortizes the cost of retrieving k fragments over recovering all missing sibling fragments. Since we are using $(k = 4, n = 12)$ erasure coding, the sibling fragment recovery uses approximately one third more network capacity compared to the no-failure case because it must retrieve 4 fragments in order to reconstruct the missing fragments (Figure 7).

For the KLS failures, we consider the same convergence optimizations as in the FS experiments. We do not show the effect of these optimizations on the number of messages for the KLS failures, as they are similar to the FS failure scenario. We separate 2 KLS failures into two cases: one KLS per data center and so the network remains connected (2C) versus two KLSs in one data center and so the network is effectively partitioned (2P). The KLS-2P failure scenario mimics a WAN partition because the proxy cannot access any KLSs in one of the data centers and so does not identify any locations in that data center.

Figure 8 shows the amount of data exchanged during convergence for varying number of KLSs being unavailable. The amount of data exchanged during convergence is dominated by the fragments, so the KLS failures add only a little overhead so long as the both data centers remain connected. If there is a WAN partition, then all the FSs on one side of the partition need to recover fragments during convergence. The sibling fragment recovery optimization prevents all FSs from independently transferring fragments needed for their recovery over the WAN, instead, only one of the FSs performs this recovery on behalf of the others. This optimization reduces the usage of the limited WAN bandwidth.

5.4. Lossy network

Figure 9 shows the performance of Pahoehoe under a lossy network that drops messages with a uniformly random drop rate ranging from 0% up to 15%. In practice, we

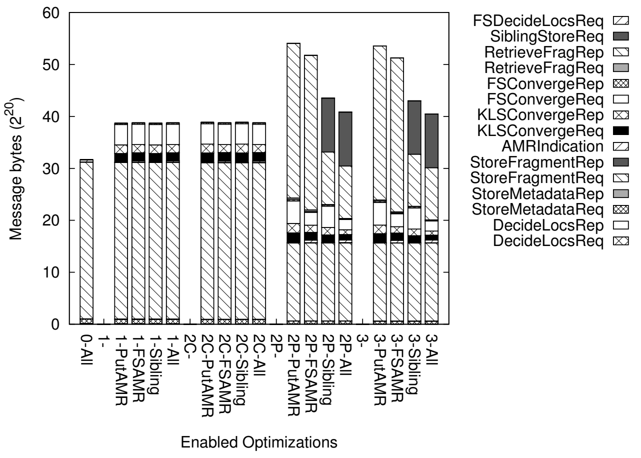

Figure 8. KLS failures and message bytes.

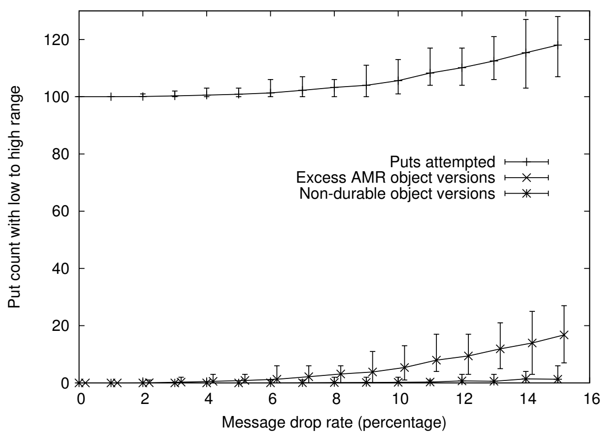

Figure 9. Convergence and a lossy network.

do not expect networks to exhibit such egregious behavior since transport mechanisms such as TCP effectively mask packet losses. This experiment is thus more of a thought experiment to understand how the cost of convergence could grow under extraordinary circumstances. This experiment also exercises some code paths that may occur due to substantial delays in the network or a proxy failures. All convergence optimizations are enabled for this experiment.

Notice that as the message drop rate increases, the number of put operations the proxy has to attempt to receive 100 success replies increases. Further note that most of these additional put attempts lead to *excess AMR object versions*, that is, even though the proxy did not receive a success reply for many puts, these puts eventually resulted in AMR object versions. Also shown on the graph is the number of *non-durable object versions*, that is object versions that never durably stored sufficient fragments to achieve AMR. The rate at which non-durable object versions occurs is very low, given the extraordinary failure scenario.

6. Related work

A number of distributed key-value storage systems [5, 7, 28] have been implemented in the past few years, all using data replication for availability, unlike Pahoehoe, which supports both erasure codes and replication. Dynamo [7], Amazon's highly available key-value store uses *sloppy quorums* and object versioning to provide weak/probabilistic consistency of replicated data. Dynamo uses *hinted handoffs* to propagate data to nodes that ought to host the data, but were unavailable when the data was updated. Hinted handoffs, like convergence in Pahoehoe, ensure that eventually the "right" nodes host the right data.

Data replication has also been widely used in distributed file systems, including Ficus [20], Coda [25], Pangaea [24], Farsite [8], WheelFS [26] and the Google File System (GFS) [11]. GFS and Farsite, like most file systems, provide fairly strong consistency and so are not suited for deployment across a WAN. Ficus, Coda, and Pangaea allow reads and writes even when some servers are disconnected; each guarantees a version of eventual consistency, but differs in how and when update conflicts are resolved. WheelFS allows the user to supply semantic cues to indicate access requirements: for example, the user can specify a maximum access time or the level of consistency expected ("eventual consistency" or "close-to-open") for a file update. Bayou [27] is a replicated, distributed, eventually causally consistent relational database system that allows disconnected operations and can tolerate network partitions. It uses an anti-entropy protocol to propagate updates between pairs of storage replicas. Baldoni et al. [3] demonstrate a replication protocol where replicas eventually achieve consistency by using gossiping, but are never aware that consistency has been achieved. None consider eventual consistency of erasure-coded data.

There are relatively few distributed storage systems that use erasure codes. Goodson et al. [14] and Cachin & Tessaro [4] have designed erasure-coded distributed atomic registers. The Federated Array of Bricks (FAB) [23] is an erasure-coded distributed block store. Ursa Minor [1] is an erasure-coded distributed object store. Pond [21] is an erasure-coded distributed file system. All such erasure-coded distributed storage systems of which we are aware, provide strong consistency and so cannot be available during a network partition. Pahoehoe, however, provides eventual consistency and can be available during a network partition.

Peer-to-peer systems use a globally consistent protocol to locate stored objects. For example, PAST [22] is an example of a distributed storage system that uses DHTs for object location. Pahoehoe uses the more traditional approach of storing location information in metadata servers (i.e., KLSs), but could use DHT techniques to scale the KLSs.

7. Conclusions

In this paper, we presented the Pahoehoe key-value cloud storage system and studied the efficiency of its *convergence* protocol. To achieve the high availability and partition-tolerance needed for the cloud, Pahoehoe provides eventual consistency. Unlike previous eventually consistent systems that use replication, Pahoehoe uses erasure codes to durably store values.

Our key contribution is to show how a distributed erasure-coded storage system can provide eventual consistency without incurring excessive network overheads. Traditional gossip-based mechanisms are not well-suited because they require $k \geq 1$ fragments to recover a single erasure-coded fragment, thereby increasing network traffic significantly to achieve eventual consistency. In Pahoehoe, each Fragment Server (FS) independently runs convergence to ensure eventual consistency under failures. We present optimizations that reduce network traffic—both messages sent and bytes sent—while still maintaining the robust, decentralized nature of convergence. These optimizations include allowing proxies and FSs to send indications that convergence is not needed (i.e., the value has achieved eventual consistency) and allowing a single FS to recover fragments on behalf of its sibling FSs to reduce or eliminate much network traffic. Our experiments show that in the failure-free case, the Pahoehoe implementation achieves network efficiency close to what we believe an ideal implementation could achieve. Our experiments also show that the optimizations taken together reduce both message count and message bytes across a broad range of failure cases.

References

[1] M. Abd-El-Malek, et al. Ursa Minor: Versatile cluster-based storage. In *FAST'05*, pages 59–72, December 2005.

[2] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[3] R. Baldoni, et al. Unconscious eventual consistency with gossips. In *Proceedings of the Eighth International Symposium on Stabilization, Safety and Security of Distributed Systems*, pages 65–81. Springer, 2006.

[4] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *DSN'06*, pages 115–124, June 2006.

[5] Cassandra. Available at `http://incubator.apache.org/cassandra/`. Accessed December 2009.

[6] J. Dean. Designs, lessons and advice from building large distributed systems. Keynote slides at `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf` Accessed December 2009.

[7] G. DeCandia, et al. Dynamo: Amazon's highly available key-value store. In *SOSP'07*, pages 205–220, October 2007.

[8] J. R. Douceur and J. Howell. Distributed directory service in the farsite file system. In *OSDI'06*, pages 321–334, November 2006.

[9] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[10] R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP'03*, pages 29–43, October 2003.

[12] G. A. Gibson, et al. A cost-effective, high-bandwidth storage architecture. In *ASPLOS'98*, pages 92–103, October 1998.

[13] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[14] G. Goodson, et al. Efficient Byzantine-tolerant erasure-coded storage. In *DSN'04*, pages 135–144, June 2004.

[15] L. Lamport. On interprocess communication, Part I: Basic formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, June 1986.

[16] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North Holland, Amsterdam, 1978.

[17] D. L. Mills. Network time synchronization research project. Available at `http://www.cis.udel.edu/~mills/ntp.html`. Accessed December 2009.

[18] E. Pierce and L. Alvisi. A framework for semantic reasoning about byzantine quorum systems. In *PODC'01*, pages 317–319, August 2001.

[19] J. S. Plank, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST'09*, February 2009.

[20] G. J. Popek, et al. Replication in Ficus distributed file systems. In *Proceedings of the Workshop on Management of Replicated Data*, pages 20–25. IEEE, November 1990.

[21] S. Rhea, et al. Pond: The OceanStore prototype. In *FAST'03*, pages 1–14, March 2003.

[22] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP'01*, pages 188–201, October 2001.

[23] Y. Saito, et al. FAB: Building distributed enterprise disk arrays from commodity components. In *ASPLOS'04*, pages 48–58, October 2004.

[24] Y. Saito, et al. Taming aggressive replication in the Pangaea wide-area file system. In *OSDI'02*, pages 15–30, December 2002.

[25] M. Satyanarayanan, et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[26] J. Stribling, et al. Flexible, wide-area storage for distributed systems with WheelFS. In *NSDI'09*, pages 43–58, April 2009.

[27] D. B. Terry, et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP'95*, pages 172–183, 1995.

[28] Tokyo cabinet. Available at `http://1978th.net/tokyocabinet/`. Accessed December 2009.

[29] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.