Naming and Integrity: Self-Verifying Data in Peer-to-Peer Systems

Hakim Weatherspoon, Chris Wells, and John D. Kubiatowicz
Computer Science Division
University of California, Berkeley
{hweather, kubitron}@cs.berkeley.edu
http://oceanstore.cs.berkeley.edu

Abstract

Peer-to-peer systems are positioned to take advantage of gains in network bandwidth, storage capacity, and computational resources to provide long-term durable storage infrastructures. In this paper, we contribute a naming technique to allow an erasure encoded document to be self-verified by the client or any other component in the system.

1 Introduction

Today's exponential growth in network bandwidth, storage capacity, and computational resources has inspired a whole new class of distributed, peer-to-peer storage infrastructures. Systems such as Farsite[3], Freenet[5], Intermemory[4], OceanStore[8], CFS[6], and PAST[7] seek to capitalize on the rapid growth of resources to provide inexpensive, highly-available storage without centralized servers. The designers of these systems propose to achieve high availability and long-term durability, in the face of individual component failures, through replication and coding techniques.

Although wide-scale replication has the potential to increase availability and durability, it introduces two important challenges to system architects. First, system architects must increase the number of replicas to achieve high durability for large systems. Second, the increase in the number of replicas increases the bandwidth and storage requirements of the system. Erasure Coding vs. Replication[15] showed that systems that employ *erasure-resilient codes*[2] have mean time to failures many orders of magnitude higher than replicated systems with similar storage and bandwidth requirements. More importantly, erasure-resilient systems use an order of magnitude less bandwidth and storage to provide similar system durability as replicated systems.

This paper makes the following contributions: First, we discuss some problems with data integrity associated with erasure codes. Second, we contribute a naming technique to allow an erasure encoded document to be self-verified by the client.

2 Background

Two common methods used to achieve high data durability are replication[3, 7] and parity schemes such as RAID[11]. The former imposes high bandwidth and storage overhead, while the latter fails to provide sufficient robustness for the high rate of failures expected in the wide area.

An *erasure code* provides redundancy without the overhead of strict replication. Erasure codes divide an object into m fragments and recode them into n *fragments*, where $n > m$. We call $r = \frac{m}{n} < 1$ the *rate* of encoding. A rate r code increases the storage cost by a factor of $\frac{1}{r}$. The key property of erasure codes is that the original object can be reconstructed from *any* m fragments. For example, using a $r = \frac{1}{4}$ encoding on a block divides the block into $m = 16$ fragments and encodes the original m fragments into $n = 64$ fragments; increasing the storage cost by a factor of *four*. Erasure codes are a superset of replicated and RAID systems. For example, a system that creates four replicas for each block can be described by an ($m = 1$, $n = 4$) erasure code. RAID level 5 can be described by ($m = 4$, $n = 5$).

Identifying Erasures: When reconstructing information from fragments, we must discard failed or corrupted fragments (called *erasures*). In traditional applications, such as RAID storage servers, failed fragments are identified by failed devices or uncorrectable read errors. In a malicious environment, however, we must be able to prevent adversaries from presenting corrupted blocks as valid. This suggests cryptographic techniques to permit the verification of data fragments; assuming that such a scheme exists, we can utilize any any m *correctly verified* fragments to reconstruct a block of data. In the best case, we could start by requesting m fragments, then incrementally requesting more as necessary. Without the ability to identify corrupted fragments directly, we could still request fragments incrementally, but might be forced to try a factorial combination of all returned fragments to find a set of m that reconstructs our data; that is, $\binom{n}{m}$ combinations.

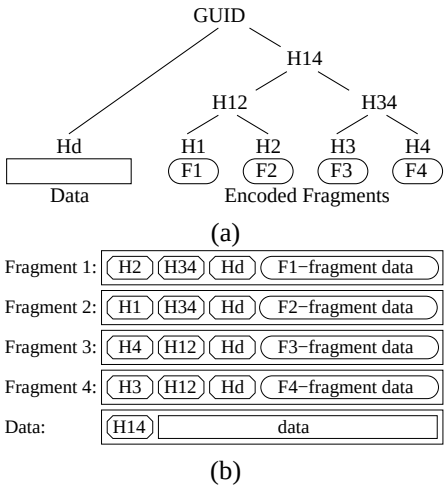

Figure 1: *A* Verification Tree: *is a hierarchical hash over the fragments and data of a block. The top-most hash is the block's* GUID. *(b)* Verification Fragments: *hashes required to verify the integrity of a particular fragment.*

Naming and Verification: A dual issue is the *naming* of data and fragments. Within a trusted LAN storage system, local identifiers consisting of tuples of server, track, and block ID can be used to uniquely identify data to an underlying system. In fact, the inode structure of a typical UNIX file system relies on such data identification techniques. In a distributed and untrusted system, however, some other technique must be used to identify blocks for retrieval and verify that the correct blocks have been returned. In this paper, we will argue that a secure hashing scheme can serve the dual purpose of identifying and verifying both data and fragments. We will illustrate how data in both its fragmented and reconstructed forms can be identified with the *same secure hash value*.

3 The Integrity Scheme

In this section, we show how a cryptographically-secure hash, such SHA-1[10], can be used to generate a *single, verifiable name* for a piece of data and all of its encoded fragments. We may utilize this name as a query to location services or remote servers, then verify that we have received the proper information simply by recomputing the name from the returned information.

Verification Tree: For each encoded block, we create a binary verification tree[9] over its fragments and data as shown in Figure 1a. The scheme works as follows: We produce a hash over each fragment, concatenate the corresponding hash with a sibling hash and hashing again to produce a higher level hash, *etc.*. We continue until we reach a topmost hash (H14 in the figure). This hash is concatenated with a hash of the data, then hashed one final time to produce a *globally-unique identifier (GUID)*. The GUID is a permanent pointer that serves the dual purpose of identifying and verifying a block. Figure 1b shows the contents of each *verification fragment*. We store with each fragment all of the sibling hashes to the topmost hash, a total of $(\log n) + 1$ hashes, where n is the number of fragments.

Verification: On receiving a fragment for re-coalescing (i.e. reconstructing a block), a client verifies the fragment by hashing over the data of the fragment, concatenating that hash with the sibling hash stored in the fragment, hashing over the concatenation, and continuing this algorithm to compute a topmost hash. If the final hash matches the GUID for the block, then the fragment has been verified; otherwise, the fragment is corrupt and should be discarded. Should the infrastructure return a complete data block instead of fragments (say, from a *cache*), we can verify this by concatenating the hash of the data with the top hash of the fragment hash tree (hash H14 in Figure 1) to get the GUID. As a result, data supplemented with hashes as above may be considered *self-verifying*.

More Complex Objects: We can create more complex objects by constructing hierarchies, i.e. placing GUIDs into blocks and encoding them. The resulting structure is a tree of blocks, with original data at the leaves. The GUID of the topmost block serves much the same purpose as an inode in a file system, and is a verifiable name for the whole complex. We verify an individual data block (a leaf) by verifying all of the blocks on the path from the root to the leaf. Although composed of many blocks, such a complex is immune to substitution attacks because the integrity and position of each block can be checked by verifying hashes. Complex objects can serve in a variety of rolls, such as documents, directories, logs, etc.

4 Discussion

Self-verifying data adds an interesting property to large-scale distributed storage infrastructures targeted for the untrusted wide area. Specifically, self-verifying data enhances location-independent routing infrastructures (such as, CAN[12], Chord[13], Pastry[7], and Tapestry[16]) by cryptographically binding the *name* of objects to their *content*. Consequently, any node that has a *cached* copy of an object or piece of an object can return it on a query; the receiving node does not have to trust the responder, merely check that the returned data is correct.

In the following paragraphs, we explore additional issues with erasure-coding data as we have advocated here:

Human-Readable Name Resolution: Up to this point in the document, we have used the word "name" and

"GUID" interchangeably. Of course, what people typically call a "name" is somewhat different: a human-readable ASCII string. ASCII names can be accommodated easily by constructing objects that serve as *directories* or *indexes*. These objects can map human readable names into GUIDs. By constructing a hierarchy of directory objects, we easily recover a hierarchical name-space such as present in most file systems. The GUID of a top-level directory becomes the *root inode*, with all self-verifying objects available along a path from this root.

Mutable Data: One obvious issue with the scheme presented in this paper is that data is fundamentally *read-only*, since we compute the name of an object from its contents; if the contents change, then the name will change, or alternatively, a new object is formed. This latter viewpoint is essentially *versioning* [14], namely the idea that every change creates a new and independent version[1]. As a result, any use of this verification scheme for writable data must be supplemented with some technique to associate a fixed name with a changing set of version GUIDs. Unfortunately, this binding can no longer be verified via hashing, but must instead involve other cryptographic techniques such as signatures. This is the approach taken in OceanStore [8].

Encoding Overhead: Each client in an erasure-resilient system sends messages to a larger number of *distinct* servers than in a replicated system. Further, the erasure-resilient system sends smaller "logical" blocks to servers than the replicated system. Both of these issues could be considered enough of a liability to outweigh the results of the last section. However, we do not view it this way. First, we assume that the storage servers are utilized by a number of clients; this means that the additional servers are simply spread over a larger client base. Second, we assume intelligent buffering and message aggregation; that is, temporarily storying fragments in memory and writing them to disk in clusters. Although the outgoing fragments are "smaller", we simply aggregate them together into larger messages and larger disk blocks, thereby nullifying the consequences of fragment size. These assumptions are implicit in the exploration via metrics of total bandwidth and number of disk blocks in [15].

Retrieval Latency: Another concern about erasure-resilient systems is that the time and server overhead to perform a read has increased, since multiple servers must be contacted to read a single block. The simplest answer to such a concern is that mechanisms for *durability* should be separated from mechanisms for *latency reduction*. Consequently, we assume that erasure-resilient coding will be utilized for durability, while replicas (i.e. caching) will be utilized for latency reduction. The nice thing about this organization is that replicas utilized for caching are *soft-state* and can be constructed and destroyed as necessary to meet the needs of temporal locality. Further, prefetching can be used to reconstruct replicas from fragments in advance of their use. Such a hybrid architecture is illustrated in Figure 2. This is similar to what is provided by OceanStore [8].

[1] Whether all past versions are kept around is an orthogonal issue.

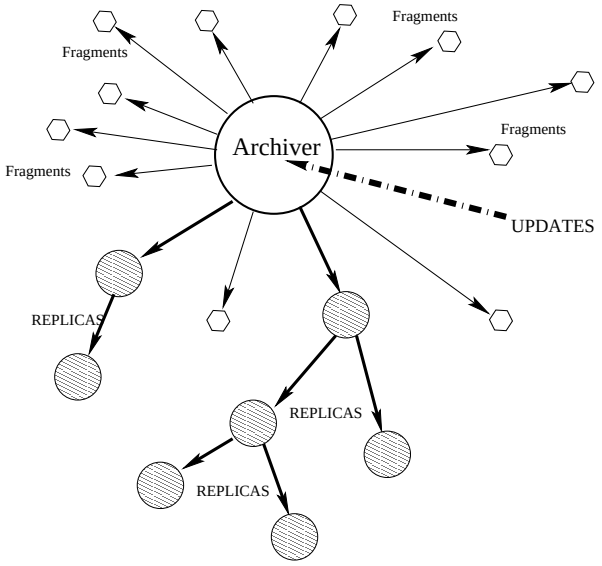

Figure 2: Hybrid Update Architecture: *Updates are sent to a central "Archiver", which produces archival fragments at the same time that it updates live replicas. Clients can achieve low-latency read access by utilizing replicas directly.*

5 Related and Future Directions

In this paper, we described the availability and durability gains provided by an erasure-resilient system. More importantly, we contributed a naming technique to allow an erasure encoded document to be self-verified.

The idea of a global-scale, distributed, persistent storage infrastructure was first motivated by Ross Anderson in his proposal for the Eternity Service[1]. A discussion of the durability gained from building a system from erasure codes first appeared in Intermemory[4]. The authors describe how their technique increases an object's resilience to node failure, but the system does not incorporate a checksum technique since the target environment is trusted.

Some open research issues deal with data structures for documents that take advantage of the peer-to-peer networks and self-verifying data. That is, replicating the document as whole, sequence of bytes, higher level structure, such as a B-tree, etc. Another issue with Peer-to-peer storage infrastructures is that they will fail if no

repair mechanisms are in place. Self-verifying documents lend themselves well to distributed repair techniques. That is, the integrity of a replica or fragment can be checked locally or in a distributed fashion.

References

[1] ANDERSON, R. The eternity service. In *Proceedings of Pragocrypt* (1996).

[2] BLOEMER, J., KALFANE, M., KARPINSKI, M., KARP, R., LUBY, M., AND ZUCKERMAN, D. An XOR-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, The International Computer Science Institute, Berkeley, CA, 1995.

[3] BOLOSKY, W., DOUCEUR, J., ELY, D., AND THEIMER, M. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. of Sigmetrics* (June 2000).

[4] CHEN, Y., EDLER, J., GOLDBERG, A., GOTTLIEB, A., SOBTI, S., AND YIANILOS, P. Prototype implementation of archival intermemory. In *Proc. of IEEE ICDE* (Feb. 1996), pp. 485–495.

[5] CLARK, I., SANDBERG, O., WILEY, B., AND HONG, T. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, CA, July 2000), pp. 311–320.

[6] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP* (October 2001).

[7] DRUSCHEL, P., AND ROWSTRON, A. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP* (2001).

[8] KUBIATOWICZ, J., ET AL. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS* (Nov. 2000), ACM.

[9] MERKLE, R. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO* (1988), C. Pomerance, Ed., Springer-Verlag, pp. 369–378.

[10] NIST. FIPS 186 digital signature standard. May 1994.

[11] PATTERSON, D., GIBSON, G., AND KATZ, R. The case for raid: Redundant arrays of inexpensive disks, May 1988.

[12] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A scalable content-addressable network. In *Proceedings of SIGCOMM* (August 2001), ACM.

[13] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM* (August 2001), ACM.

[14] STONEBRAKER, M. The design of the Postgres storage system. In *Proc. of Intl. Conf. on VLDB* (Sept. 1987).

[15] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *Proc. of International Workshop on Peer-to-Peer Systems* (2002).

[16] ZHAO, B., JOSEPH, A., AND KUBIATOWICZ, J. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB//CSD-01-1141, University of California, Berkeley Computer Science Division, April 2001.