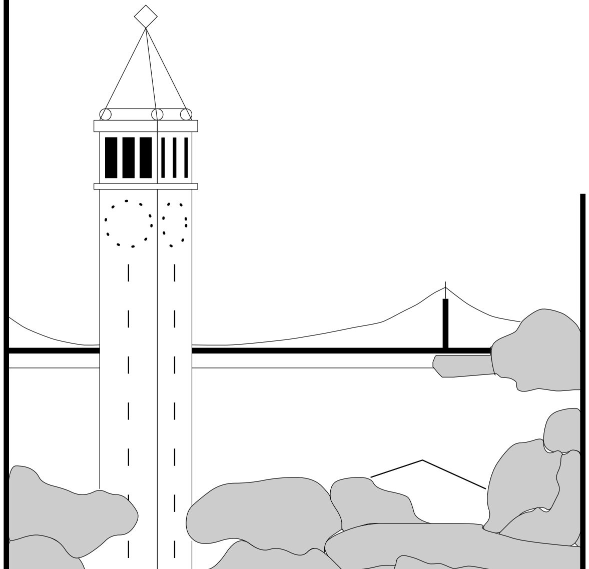
Two-Level, Self-Verifying Data for Peer-to-Peer Storage

Patrick Eaton, Hakim Weatherspoon, and John Kubiatowicz
University of California, Berkeley



Report No. UCB/CSD-05-1401

June 2005

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Two-Level, Self-Verifying Data for Peer-to-Peer Storage

Patrick Eaton, Hakim Weatherspoon, and John Kubiatowicz
University of California, Berkeley

June 2005

Abstract

First-generation peer-to-peer storage systems unnecessarily couple the unit of client data access to the unit of infrastructure data management. Designs that require all peers to operate on data at a fixed granularity lead to inefficiencies such as high query load and high per-block storage overheads. To provide variable granularity access and support more efficient peer-to-peer storage systems, we introduce two-level naming of self-verifying data. We describe how to implement two-level naming and advocate an extension to the traditional API used by peer-to-peer storage systems to support two-level naming.

1 Introduction

Self-verifying data is a fundamental building block of peer-to-peer storage systems. It allows clients to validate the integrity of data and the infrastructure to identify corrupt data. Because many peer-to-peer applications are targeted to run on mutually distrusting machines spread across the wide-area, the ability to verify data is vital. Self-verifying data guards against errors introduced by faulty peers or transmission through public networks. It also protects clients from malicious or compromised servers that attempt to deceive users by returning modified data.

Examining a variety of first-generation peer-to-peer storage systems, we observed that they share a number of design decisions with respect to self-verifying data. For example, most clients create and access many small data blocks that are linked into larger data structures. After creating data, clients store each block in the storage infrastructure as an independent objects. The storage infrastructure then manages, indexes, tracks, and repairs each small block individually.

Such a design effectively couples the infrastructure's unit of data management to the client's unit of data creation and access. This coupling engenders a challenge for creating efficient systems. For clients at the edges of the network, it is natural to work with data divided into small blocks. However, a storage infrastructure that manages small blocks sees higher indexing cost (because the infrastructure must track all replicas of each smaller block individually) and a larger query load (because clients must use the infrastructure to locate each small block). On the other hand, a storage infrastructure can reduce the overhead of tracking data and resolving queries by amortizing the costs over larger blocks of data. However, requiring clients to work with large blocks can waste precious bandwidth at the edges of the network and require significant data buffering that limits data durability. Most existing systems have elected to use relatively small data blocks, resulting in systems with high query load on the infrastructure, high storage overhead, and poor communication patterns.

To improve the efficiency of peer-to-peer storage systems, we introduce *two-level naming* for self-verifying data. Two-level naming decouples the unit of management from the unit of access by packing many small, application-level data blocks into larger containers called *extents* while maintaining the self-verifying properties. This solution allows different components to access data at different granularities—clients can access exactly the data they desire by referencing data at a fine granularity while peers in the infrastructure can operate on larger containers to amortize the cost of indexing and querying data. To support two-level naming, we advocate an extension to the traditional API used by peer-to-peer storage systems.

In Section 2, we review the concepts of self-verifying data and other related work. In Section 3, we detail the prevailing design decisions of popular, first-generation peer-to-peer storage systems and describe the consequences of these designs. In Section 4, we present two-level naming and describe an implementation based on extending the API used by traditional peer-to-peer storage systems. Section 5 shows how to use that API to build a versioning backup application. Finally, Section 6 concludes.

2 Background and Related Work

Data is said to be self-verifying if it is named in a way that allows any peer to validate the integrity of data against the name by which it was retrieved. The self-verifying property enables clients to request data from any peer in the network without concern of data corruption or substitution attack. A malicious peer cannot deceive a client with corrupt data—its attack is limited to denying a block's existence.

Data can be made self-verifying via two techniques: hashing and embedded signatures. Hash-verified data is named by a secure hash of its content. A client can verify hash-verified data by recomputing the hash of the data. Key-verified data is named by the hash of a public key that verifies a signature over a secure hash of the data. To verify the data, a client uses the public key that corresponds to the data's name to verify the signature over the data, recomputes the hash of the data, and compares the computed hash with the signed hash. These techniques were made popular by the Self-certifying Read-only File System [4].

Many systems employ Merkle's chaining technique [10] with hash-verified data to combine blocks into larger, self-verifying data structures. Such systems embed self-verifying names into other data blocks as secure, unforgeable pointers. To bootstrap the process, systems often embed secure pointers in key-verified blocks, providing an immutable name for mutable data. To update data, then, a client replaces a key-verified block with a block that references the new data. See, for example, CFS [2], Ivy [12], and Venti [13].

Additionally, Weatherspoon et al. extended the hash-based approach to name erasure code fragments in a self-verifying manner [17]. Clients can verify either individual erasure code fragments or the full block of data by the same name. Distillation codes [7] can be considered a generalization of this scheme.

The classical file systems literature provides an inspiring precedent of improving system efficiency by adapting the granularity of data access. Many file systems manage data at different granularities at different levels of the storage hierarchy to improve performance. For example, the Fast File System (FFS) [9] increased performance, in part, by increasing the unit of transfer to the disk. Also, XFS [16] managed storage in extents, or sequences of blocks, to reduce the size of the metadata and allow for fast sequential access to data. GoogleFS [5] uses extents to reduce the per-object maintenance costs. None of these systems, however, combine the concepts of aggregation and self-verifying data.

One recent system does use aggregation to improve the efficiency of content-addressable storage. To reduce the number of objects that the system must track, Glacier [6] relies on a proxy trusted by the user to aggregate many application-level objects into larger collections. An object's durability is limited while data is buffered at the proxy. Once a proxy publishes an aggregate, its content cannot be modified.

3 Analyzing Existing Systems

In this section, we identify dominant design decisions made in first-generation peer-to-peer storage systems. We then consider the consequences of these decisions.

3.1 Prevailing Design Decisions

Limited space precludes us from detailing each peer-to-peer storage system individually. Instead, we present the prevailing design decisions that we observed in popular, first-generation peer-to-peer storage systems including CFS [2], Ivy [12], OceanStore [14], Total Recall [1], and Venti [13].

The systems that we studied tended to use self-verifying data in a common manner, illustrated in Figure 1. Applications divide objects into small, hash-verified blocks ranging in size from a few tens of bytes

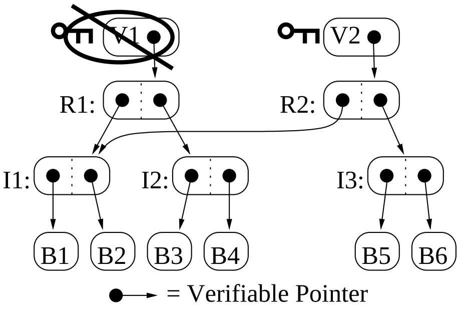


Figure 1: Applications divide data into small blocks which are combined into larger, linked data structures. A key-verified block points to the root of the structure. To update an object, an application overwrites the key-verified block to point to the new root. (V = version, R = version root, I = indirect node, B = data block)

for a log entry [12] or inode block [2] to a few kilobytes for a data block. Using the hashes that name the blocks as unforgeable references, hash-verified blocks are combined into larger data structures, like linked lists [12] or trees [2, 14]. Key-verified blocks point to the roots of these structures, providing an immutable name to mutable data.

After dividing an object into blocks, the client stores the blocks in the peer-to-peer storage infrastructure. Peers in the storage infrastructure work together to serve as a *query router*, tracking the location of data and resolving queries to a peer storing the data. In first-generation systems, the query router indexes and tracks each block of data stored by a client. That is, the unit of management in the query router is the same as the unit of creation and access by the client application. Because they access and manage data at the same granularity, clients and the query router can interact through a simple `put()/get()` interface, shown in Table 1, reminiscent of the interface to a hashtable. While we have shown `putHash()` and `putKey()` as distinct members of the interface, they are often implemented as a single `put()` function.

To update an object, applications create new blocks which reference older blocks or other new blocks and store them in the infrastructure. To make updates visible to others, applications overwrite the key-verified block of the object.

Traditional interface:		
		putHash(H(data), data);
		putKey(H(PK), data);
data	=	*get*(hash);

Table 1: The traditional `put()/get()` interface used by first-generation peer-to-peer storage systems. The `putHash()` and `putKey()` functions are often combined into a single `put()` function.

Because each new hash-verified block of data has a unique name, these systems naturally provide versioning capabilities. Some systems expose the versioning feature to the end user [14] while others do not. Using copy-on-write to provide efficient versioning has also been implemented in other systems predating the peer-to-peer systems that we describe [11].

One notable counterexample to these patterns is the PAST [15] system. PAST stores whole objects as hash-verified blocks and relies on complicated shedding and forwarding protocols to find servers that can store large replicas. PAST objects cannot be incrementally updated; they can only be wholly replaced.

3.2 Consequences of Design Decisions

These design decisions determine many characteristics of the resulting system. For example, granting clients fine granularity access to data leads to several benefits. First, it is a natural interface for many applications that consider data to be a collection of small blocks. It also allows clients to fetch data without wasting scarce bandwidth at the edges of the network retrieving data that is not needed or already cached. Finally, it allows clients to push data to the infrastructure as soon as it is created, improving durability.

Fine granularity access to data also has several disadvantages. Because each block is independently managed in the infrastructure, the client must issue a separate query to the query router for each block. To read an object of even moderate size, a client must issue many queries. The query router, then, must be able to support heavy query loads and to route many simultaneous queries efficiently. Figure 2 illustrates a query router operating in the traditional scenario where each block is managed separately.

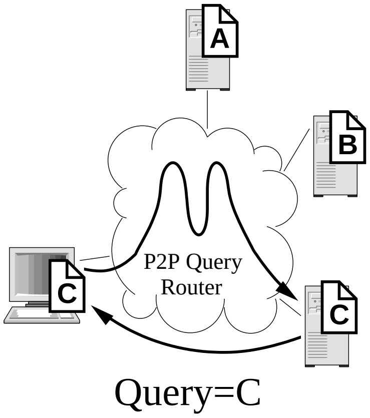


Figure 2: A system stores three blocks of data, A, B, and C. First-generation systems route queries to individual blocks.

Other consequences are more directly related to specific types of query routers. One common type of query router is the distributed hashtable (DHT) [3]. In a DHT, each participating node is responsible for a fixed portion of the namespace. When storing data, a DHT transfers the block to the peer that is responsible for the portion of the namespace containing the name of the block. Due to their design, systems that rely on DHTs for query routing must cope with widely distributed communication patterns. Because the location of data is determined solely by its name and because hashing creates effectively random names, data blocks from a single object are spread randomly across servers in the network. To read even a single object, a client's queries are routed to many different servers—often most of the servers in the system.

Decentralized object location and routing (DOLR) [3] networks are another type of query router that allow clients to select where data is placed. Systems built atop DOLR-based query routers can manage data location to improve communication patterns. To resolve queries quickly, a DOLR maintains small pointers that track the location of data. While this approach enables efficient data placement strategies, the pointer state required to track each data block and the bandwidth required to maintain that state can waste valuable storage and bandwidth.

To see these issues in practice, consider a versioning backup application storing a 1 GB filesystem. CFS, Pond, and Venti suggest that applications divide objects into blocks with a maximum size of 8 KB. Thus, the filesystem would be divided into at least 131,000 data blocks. To read the data from the system, a client would need to query for and fetch each block. Furthermore, DOLR-based systems, such as Pond and Total Recall, must maintain extra pointer state to index each of those blocks. If a pointer is 64 bytes and is replicated 8 times for availability, then the cost to store *only the index* of the filesystem is 64 MB. This storage overhead becomes even more significant, growing to 256 MB or 25%, if each block is modestly replicated 4 times to ensure durability. Even if the cost of storage is not a concern, the large index can lead to other problems. Some DOLRs maintain the index very aggressively to ensure that the query router can respond to queries [1]. As the index grows larger, the bandwidth required to maintain the index also increases. In the end, a system could use more bandwidth maintaining the indexing state than maintaining or serving the data.

4 Two-Level, Self-Verifying Data

For flexibility and control, clients must be able to access data at a fine granularity. This does not, however, mean that the infrastructure must manage data at such a fine granularity. In fact, the root cause of problems outlined in Section 3.2 is the unnecessary coupling of the unit of management in the infrastructure to the unit of access at the client.

4.1 An Alternative: Two-Level Naming

If many small, variable-sized, application-level blocks were coalesced into larger containers in the infrastructure, the system could store, index, and locate data more efficiently. This leads to a modified query process that we call *two-level naming*, shown in Figure 3.

To retrieve a block of data, a client first queries the infrastructure to find the enclosing container; it then requests a specific block from the container. Each block is identified not by a single hash, but by a tuple. The first element identifies the enclosing container; the second element names the block within the container. After identifying a server storing a container, a client can send subsequent requests directly to the server without querying the

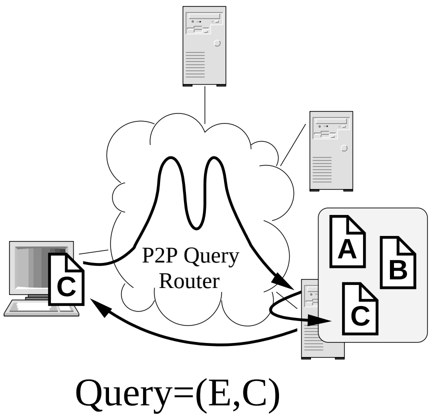


Figure 3: A system stores three blocks of data, A, B, and C. With two-level naming, small blocks are aggregated into larger extents. The query includes both an extent name and a block name.

infrastructure.

Let us return to the example backup application described previously to see the potential impact of two-level naming. Assume that the application aggregates blocks into 4 MB containers. The query routing load imposed by a client reading the 1 GB filesystem would decrease three orders of magnitude to 256 operations. For the DOLR-based systems, assuming the same pointer size and replication level, the indexing state maintained by the infrastructure drops from 64 MB to 128 KB.

4.2 An Extent-Based Interface

Next, we discuss how to implement two-level naming to aggregate many small, variable-sized *blocks* of data into containers called *extents*.

We could support two-level naming with only minor modifications to current systems by forcing clients (or their agents) to aggregate data, as is done in Glacier [6]. The consequences, however, of such an approach are unappealing. A client would buffer blocks locally until it could fill an extent. The client would then merge the set of blocks into an extent and store the container with a single `put()` operation. Implementing this scheme would require only a slight extension to the query process to allow clients to retrieve individual blocks from an extent. While easy to implement, this approach has several problems. With applications that create new data at a slow rate, a client will buffer data locally for a long time before writing it to the system. This impacts the durability and visibility of the data and can lead to lost data in the event of a client crash.

Two-Level interface:		
		create(H(PK), cert);
ext name	=	*append*(H(PK), cert, data[]);
ext name	=	*truncate*(H(PK), cert);
ext name	=	*snapshot*(H(PK), cert);
ext name	=	*put*(cert, data[]);
cert	=	*getCert*(ext name);
data[]	=	*getBlocks*(ext name, blocks[]);
extent	=	*getExtent*(ext name);

Table 2: By extending the traditional `put()/get()` interface used by first-generation systems, peer-to-peer storage systems can support data access at different granularities and allow query routers to operate more efficiently.

Instead, we seek a solution that allows clients to store data in extents incrementally and as it is created. Recall from our discussion of self-verifying data in Section 2 that key-verified data can be modified while hash-verified data is immutable. Thus, to allow the contents of an extent to change over time, we know that mutable extents must be key-verified. However, it is not feasible to store all data in key-verified extents because that would require extents to grow boundlessly large or clients to manage large numbers of key pairs. Consequently, we include a mechanism to convert key-verified extents into hash-verified extents.

With these observations, we present the interface defined in Table 2. This interface extends the traditional interface shown in Table 1 by defining additional operations for key-verified data and including the `snapshot()` operation to convert key-verified extents to hash-verified extents. Figure 4 shows how different operations relate to different types of self-verifying data. Note from Table 2 that we assume all extents—both key-verified and hash-verified—include a certificate signed by the data owner that asserts the current content of the extent. The certificate, which includes metadata about the extent and a list of blocks stored in the extent, can also be used to attribute storage to individual users for accounting purposes.

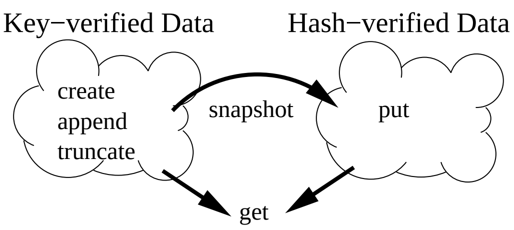


Figure 4: The expanded interface of Table 2 provides different commands to operate on different types of self-verifying data. Commands create, append, and truncate operate on key-verified extents while snapshot converts a key-verified extent into a hash-verified extent.

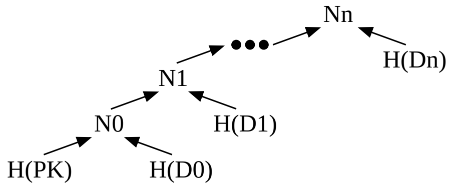


Figure 5: To compute the hash-verified name of an extent, a peer combines the blocks of the key-verified extent in a self-verifying chain. A peer can compute the eventual hash-verified name incrementally, as new blocks are added. To compute the name, a peer uses the recurrence relation $N_i = H(N_{i-1} + H(D_i)))$. $N_{-1} = H(PK)$ where PK is a public key.

Let us present an example to explain the proposed interface. A client wishing to store data first uses the `create()` interface to request that the infrastructure create an extent. New extents are key-verified objects, named by the hash of a public key that verifies the certificate stored in the extent. To create a new extent, the system must recruit a set of storage servers to host the extent. We assume that the system includes a storage set identification service that provides sets of candidate storage servers. We leave the details of this service unspecified—the service may create candidate sets using simple random assignment or using heuristics such as network location or server capacity. On a `create()` operation, the system requests a new storage set from the service and then contacts the members of that set to allocate space for the extent and participate in management of the extent.

After an extent has been created, an application can write new data to it. New data can be added to the extent using the `append()` operation. An application can append data to a key-verified extent, provided that it also supplies a new certificate that certifies those changes.

To limit the size of an extent and the number of key pairs that a client must manage, the application can periodically convert mutable, key-verified data to immutable hash-verified data using the `snapshot()` interface. Note that the `snapshot()` operation creates a new extent and thus makes use of the set identification service. Once an extent is made hash-verifiable, its contents can never change.

After saving data in an immutable format, the client can reinitialize the key-verified extent with a `truncate()` operation. The `truncate()` operation removes all blocks from the extent, leaving a key-verified extent that is equivalent to a newly created extent. While `snapshot()` and `truncate()` are typically used together, we have elected to make them separate operations for ease of implementation, especially for distributed systems with Byzantine failure modes. Each individual operation can be repeated until successful execution is assured. In Section 5, we will show how the `snapshot()` and `truncate()` operations can be used to facilitate storing streams of data.

The `append()`, `snapshot()`, and `truncate()` operations that transform mutable, key-verified extents are useful for applications that periodically write small amounts of data, allowing the system to aggregate data in a way that was not possible previously. But in situations that applications quickly write large amounts of data, using the sequence of operations can be inefficient. Instead, applications may write collections of blocks directly to hash-verified extents using the `put()` operation. The `put()` operation, of course, also relies of the set identification service.

4.3 Naming Extents

The names of hash-verified extents should be created in a way that allows any peer to reference and verify both the individual blocks and the extent as a whole. One scheme that satisfies these properties is naming blocks by chaining [8], as shown in Figure 5. Given a block of data, D_i, the block name is simply the hash of the data, $H(D_i)$. The hash-verified extent name is computed using the recurrence relation $N_i = H(N_{i-1} + H(D_i)))$, where $+$ is the concatenation operator. When snapshotting a key-verified extent that contains n blocks, the name of the resulting hash-verified extent is N_n corresponding to the last data block added to the extent. We bootstrap the process by defining N_{-1} to be a hash of the public key.

Creating extent names by chaining has several advantages. It allows us to compute names incrementally; when a block is added to an extent, we need to hash only the new data, not all data in the extent, to compute the running name. Also, chaining creates a verifiable, time-ordered log recording data modifications.

To access blocks from a key-verified extent, the client queries the system for the extent named by the hash of the public key signing the extent's certificate. To access blocks from a hash-verified extent, the client uses the name computed by hashing the chain of component blocks. In the next section, we show how an application can keep track of those names.

5 An Example Application: Versioning Backup

Finally, we present a high-level design demonstrating how one might use the interface described in Section 4 to implement a sample application, namely a versioning backup application.

Because a filesystem's version history may grow very large, the design must allow for storage to be spread across multiple extents. This design will store the version history in a series of extents. The application appends the new data from the filesystem to a key-verified extent until the extent reaches a specified maximum capacity. The application then snapshots the extent to convert it to hash-verified immutable data and truncates the key-verified extent to prepare it for more data.

We will assume that the user associates a unique key with each filesystem that she wishes to backup. To archive a filesystem—for example, the simple filesystem shown in Figure 6—the back-up application first translates the filesystem into a self-verifying Merkle tree. The form of the resulting Merkle tree for the sample filesystem is shown in Figure 7. (The details about the secure references will be explained below.) For simplicity, the file inodes are not shown in the figure. This translation process is analogous to that used in CFS [2]. Note the similarity between Figure 7 and the first version of the object shown in Figure 1.

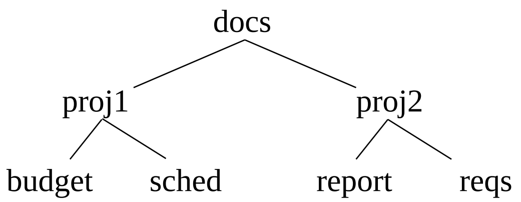


Figure 6: A simple file system used as a running example.

The key challenge when creating the Merkle tree is in determining the self-verifying block names to embed in the tree. Recall that to retrieve a block from a system that implements two-level naming, one must provide the extent name and the block name. If, however, we wish to use the snapshot functionality to convert key-verified extents to hash-verified extents, we cannot know the eventual hash-verified name of an extent as long as it is mutable. So, without knowing the eventual hash-verified name of an extent, how do we refer to a block when building the Merkle tree?

To create unique, unforgeable references for blocks stored in key-verified extents, the application assigns a sequence number, s, to each extent. New key-verified extents are assigned sequence number $s = 0$. After executing the `snapshot()` and `truncate()` operations on an extent, the application increments the sequence number and inserts it in the otherwise empty extent. When creating or updating a Merkle tree, the application embeds block references of the form $(s, H(D_i))$ where $H(D_i)$ is the hash of the data block.

The references embedded in the hash tree cannot be used to retrieve data directly because the query router does not identify extents by application-defined sequence numbers. To enable clients to retrieve data using the references embedded in the tree, the application also maintains a mapping that resolves the sequence number to the permanent, hash-verified extent name. This mapping is placed along with the sequence number as the first block in the extent. Each time the mutable extent is made hash-

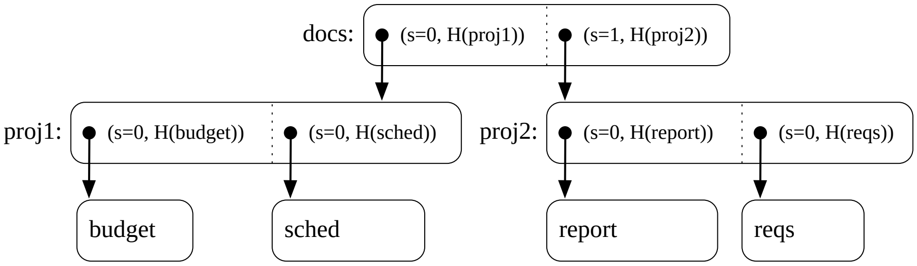


Figure 7: The Merkle tree resulting from the translation of the initial version of the filesystem into a self-verifying data structure. For simplicity, file inodes are not shown.

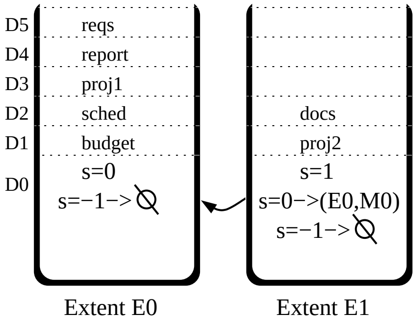


Figure 8: Two extent are used to store a back-up of the original file system. The first extent, E_0, is filled and has been converted to a hash-verified extent. The second extent, E_1, is a partially filled key-verified extent.

verified and then truncated to contain no data, the application records the mapping $s_{j-1} \rightarrow (E_{j-1}, M_{j-1})$ where E_{j-1} is the hash-verifiable name of the previous extent and M_{j-1} is the name of the metadata block that records mappings in the previous extent.

Putting all of these mechanisms together, Figure 8 shows the contents of the chain of extents after archiving the first version of the filesystem. The first block in each extent, labelled $D0$ in the figure, contains the metadata information for the application including the sequence number of the extent and the mappings between previous sequence numbers and the corresponding hash-verified names of their extent. Of course, to ensure that a parent block can record the names of all of its child blocks, the application must write blocks from the Merkle tree to extents in a depth-first, leaf-to-root fashion. In storing the initial version of the filesystem, the application completely filled one extent and partially filled another. The filled extent, $E0$, has been converted to a hash-verified extent and is immutable. The partially filled extent $E1$ is a key-verified extent and can store more data at a later time. The first block in $E1$, its metadata block includes the mapping for the previous extent $E0$. By observing the organization of data in the extents in Figure 8, the reader should now understand the derivation of the unforgeable references in the Merkle tree of Figure 7.

Figure 9 shows how the backup application handles modifications to the filesystem. Assume the user edits the files in the `proj2` directory. Figure 9(a) shows the Merkle tree resulting from these changes. The dashed pointer indicates a reference to a block from the previous version, namely block $(s = 0, H(\texttt{proj1}))$. Figure 9(b) shows the contents of the extent chain after recording the changes. The changes have filled extent $E1$ which has been converted to an immutable hash-verified extent. The key-verified extent for the filesystem is now labelled extent $E2$. Again, notice how the first block of $E2$ records the hash-verified names of previous extents and the name of the metadata block in those extents.

To recover the name of an extent corresponding to a sequence number, the application must consult the mappings that are stored in the extent. The mapping can always be found as the first data block of the key-verified extent corresponding to the object. An application can trade storage for lookup latency by storing more or fewer mappings in each extent. A client may also keep a local cache of these immutable mappings to accelerate future translations.

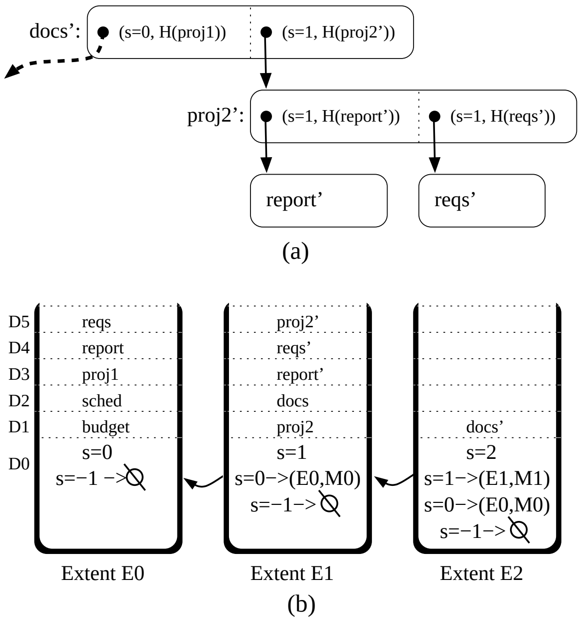


Figure 9: (a) The Merkle tree resulting from translating the updated file system. The dashed pointer indicates a reference to a block from the previous version. (b) The contents of the extents chain after storing blocks of the updated filesystem.

6 Conclusion and Future Work

We have identified the prevailing design decisions used in existing peer-to-peer storage systems focussing on the cost of coupling the unit of client data access to the unit of infrastructure data management. We proposed an alternative approach, two-level naming, to allow efficient, variable granularity access for future systems. We described an implementation and API to support two-level naming and showed how applications could use the new API.

References

[1] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Total recall: System support for automated availability management. In *Proc. of NSDI*, pages 337–350, Mar. 2004.

[2] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, pages 202–215, Oct. 2001.

[3] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proc. of International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[4] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. In *Proc. of OSDI*, Oct. 2000.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of ACM SOSP*, pages 96–108, Oct. 2003.

[6] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI*, May 2005.

[7] C. Karlof, N. Sastry, Y. Li, A. Perrig, and J. D. Tygar. Distillation codes and their application to DoS resistant multicast authentication. In *Network and Distributed System Security Conference (NDSS 2004)*, pages 37–56, Feb. 2004.

[8] J. Li, M. Krohn, D. Mazires, and D. Shasha. Secure untrusted data repository (sundr). In *Proc. of OSDI*, pages 121–136, Dec. 2004.

[9] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[10] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, pages 369–378, 1988.

[11] S. J. Mullender and A. S. Tanenbaum. A distributed file service based on optimistic concurrency control. In *Proc. of ACM SOSP*, pages 51–62, Dec. 1985.

[12] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI*, Dec. 2002.

[13] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proc. of USENIX FAST*, Jan. 2002.

[14] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. of USENIX FAST*, pages 1–14, Mar. 2003.

[15] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, pages 188–201, Oct. 2001.

[16] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proc. of USENIX Technical Conference*, pages 1–14, Jan. 1996.

[17] H. Weatherspoon, C. Wells, and J. Kubiatowicz. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proc of FuDiCo*, June 2002.