Efficiently Binding Data to Owners in Distributed Content-Addressable Storage Systems

Patrick Eaton, Hakim Weatherspoon, and John Kubiatowicz
University of California, Berkeley
{eaton, hweather, kubitron}@cs.berkeley.edu

Abstract

Distributed content-addressable storage systems use self-verifying data to protect data integrity and to enable graceful scaling. One feature commonly missing from these systems, however, is the ability to identify the owner of a piece of data in a non-repudiable manner. While a solution that associates a certificate with each block of data is conceptually simple, researchers have traditionally claimed that the cost of creating and maintaining certificates is too great. In this paper, we demonstrate that systems can, in fact, efficiently map data to its owner in a secure and non-repudiable fashion. To reduce the cost of creating and maintaining certificates, we extend the traditional content-addressable interface to allow the aggregation of many small data blocks into larger containers. The aggregation is performed in a way that also supports self-verifying data at the granularity of the block and container, fine-granularity access, and incremental updates. We describe two prototype implementations and present preliminary performance results from deployments on PlanetLab and a local cluster.

1 Introduction

In content-addressable storage (CAS) systems, data is addressed not by its physical location but by a name that is derived from the content of that data. In recent years, the CAS interface, similar to the traditional `put()`/`get()` interface of a hashtable, has proven to be a solid foundation upon which to build wide-area distributed storage systems (e.g. [2] [13] [12]).

Distributed storage systems derive several favorable properties from the CAS interface. First, a CAS interface helps ensure data integrity. If a system carefully selects the method used to derive names from data, clients can validate the integrity of data retrieved from the system against the name by which it was accessed. With *self-verifying* data, clients can detect data altered by faulty or compromised components and re-fetch from alternate sources. Also, a CAS interface promotes system scalability. Because the interface does not expose physical addresses to applications, the system can replicate and transfer data freely to add hardware resources or upgrade internal protocols.

One feature commonly missing from distributed content-addressable storage systems, however, is the ability to determine the owner of data stored in the system. Identifying the owner of a piece of data is critical for any system that wishes to monitor per-user storage consumption or compute usage-based fees. In this paper, we consider how to implement efficiently this feature—the ability to identify the owner of each piece of data—in distributed CAS systems.

The approach most commonly proposed for implementing such functionality is to include a certificate with each block of data. The certificate provides a secure, non-repudiable binding between the data and its owner. The certificate remains co-located with the data, even as the data is replicated or transferred.

While this solution is conceptually simple, an efficient implementation has proved elusive. One impediment is the time it takes the client to sign all of the certificates. In fact, some designers have rejected the solution by reasoning that the cost of producing certificates is prohibitively expensive [4]. To illustrate this problem, assume an application running on a 3 GHz processor wishes to store 1 TB of data. Dividing the data into 8 KB blocks and using 1024-bit RSA cryptography, it would take more than *six days* to create certificates for the data [1].

One approach to reduce the cost of creating certificates is to allow the system to aggregate blocks into larger container objects [6]. Then, a client needs only to provide a certificate for each container, not each individual block. Aggregation can reduce the cost of producing signatures dramatically. Consider an application storing 1 TB of data into a system that aggregates data into 4 MB containers. A client machine with a 3 GHz processor could create the certificates in *17*

[1] A 3 GHz Pentium-class processor can create a signature in 4 ms, as measured with the command `openssl speed rsa1024`.

minutes, a reduction of three orders of magnitude over a system that implements the certificate-per-block approach.

Aggregation, however, introduces other problems. For example, because the current CAS interface provides no support for aggregating data, clients must perform aggregation locally. This requires clients to buffer data until it can fill a container; buffering limits data durability. Also, if the system requires a client to retrieve an entire container when accessing only a single small block, it unnecessarily wastes bandwidth which can be especially costly at the edges of the network where bandwidth may be scarce or expensive.

In this paper, we present a design that allows members of a distributed content-addressable storage system to identify, in a secure and non-repudiable fashion, the owner of each piece of data stored in the system. To address the efficiency challenges described above, the design exploits aggregation in a novel way. The proposal aggregates small blocks of data into larger containers (to amortize the cost of creating and managing certificates) while simultaneously supporting incremental updates (to obviate data buffering at clients) and fine-granularity access (to allow clients to retrieve exactly the data they need). The design protects data integrity by maintaining self-verifiability at both the block and container granularity. We also describe prototype implementations of the interface and present preliminary performance from deployments running on PlanetLab and a local cluster.

In Section 2, we review some background concepts and early distributed CAS systems. In Section 3, we present a solution that allows systems to identify in a non-repudiable fashion the owner of each piece of stored data while simultaneously supporting incremental updates, fine-granularity access, and self-verifiability. Section 4 shows how to use the interface to build a versioning back-up application. Section 5 describes prototype implementations that we are building and preliminary performance evaluations. Finally, Section 6 concludes.

2 Background and prior work

In this section, we begin by reviewing the concepts behind self-verifying data. We then study the designs of popular, first-generation distributed CAS systems, focusing on their similarities and the consequences of those decisions.

2.1 Self-verifying data

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

Traditionally, data is made self-verifying via one of two techniques: hashing and embedded signatures. These techniques were made popular by the Self-certifying Read-only File System [4]. *Hash-verified data* is named by a secure hash of its content. A client can verify hash-verified data by computing the hash of the returned data and comparing it to the name used to fetch the data. Hash-verified data is immutable—if the data changes, the hash-verified name of the data changes too.

Key-verified data is verified by a certificate that is signed by a user's public key. The certificate contains some token, such as a secure hash of the content, that securely describes the data. To verify key-verified data, a client checks the signature on the certificate and compares the data against the verifier in the certificate. Commonly, key-verified data is named by a hash of the public key that signs the data's certificate. With this approach, each key pair can be associated with only a single object. To allow the system to associate multiple objects with a single key pair, other schemes hash a combination of data, such as the public key and a human-readable name, to create the name for the data. Key-verified data can be mutable—a client can associate new data with a key by creating a new certificate.

Many systems employ Merkle's chaining technique [9] 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 store the name of the root of the data structure in a key-verified block, providing an immutable name for mutable data. To update data, a client replaces the key-verified block. See, for example, CFS [2], Ivy [11], and Venti [12].

2.2 Distributed CAS systems

Recently, researchers have used self-verifying data and the content-addressable interface as a foundation for building distributed storage systems. Despite their independent development, many systems share important design features. In identifying common design features, we have considered a number of popular, first-generation content-addressable storage systems in the research literature including CFS [2], Ivy [11], OceanStore [13], Total Recall [1], and Venti [12].

First-generation distributed content-addressable storage systems provide a simple interface for clients to interact with the storage system. The interface, shown in Table 1, is often called a `put()/get()` interface due to its similarity to the interface of a hashtable. Note, while we have shown `put_hash()` and `put_key()` as distinct members of the

Traditional interface:		
		put_hash(H(data), data);
		put_key(H(PK), data);
data	=	*get*(h);

Table 1. First-generation distributed CAS systems use a simple `put()`/`get()` interface. The `put_hash()` and `put_key()` functions are often combined into a single `put()` function. $H()$ is a secure, one-way hash function; h is a secure hash, as output from $H()$.

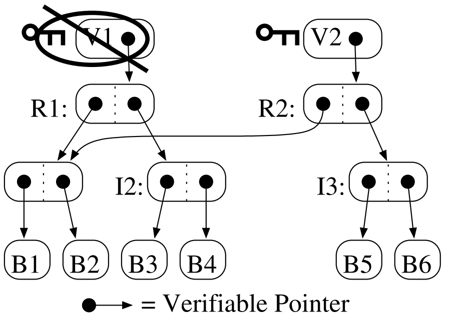


Figure 1. Clients divide data into small blocks that are combined into Merkle trees. A key-verified block points to the root of the structure. To update an object, a client overwrites the key-verified block to point to the new root. (V = version, R = version root, I = indirect node, B = data block)

interface, they are often implemented as a single `put()` function.

Systems tend to use self-verifying data and the `put()`/`get()` interface in a common manner, illustrated in Figure 1. A client divides data into small blocks, typically 4–8 KB or less. It computes the hash-verifiable name of each block and links the blocks together, using the names as unforgeable references, to create a Merkle tree. Finally, the client stores all blocks of the tree in the CAS system using the `put_hash()` interface. If the system supports mutable data, the client will typically use the `put_key()` function to store a key-verified block that points to the root of the Merkle tree, providing an immutable name to the mutable data.

To read data, a client first retrieves and validates the key-verified root block of the data structure using the `get()` function. It can then iteratively fetch and verify the other hash-verified blocks by following the chain of hash-verified names embedded in the tree.

Because each new hash-verified block of data has a unique name, CAS systems naturally provide versioning capabilities. Some systems expose the versioning feature to the end user [13] while others do not. Using copy-on-write to provide efficient versioning has also been implemented in other systems predating the distributed CAS systems that we describe [10].

One notable counterexample to these design patterns is the PAST [15] system. PAST uses the `put_hash()` call to store whole objects as hash-verified blocks. As a result, PAST cannot incrementally update objects; instead, it stores new versions of an object as a single block using the `put_hash()` interface.

The design features shared among these implementations have a significant impact on the behavior of the resulting systems. For example, the `put()`/`get()` interface forces the storage infrastructure to manage data at the same granularity as the client. While some applications, like off-line data processing, handle data in large chunks, many interactive and user-oriented applications tend to create and access relatively small blocks of data. By supporting fine granularity access, these systems allow applications to fetch data without wasting scarce bandwidth at the edges of the network retrieving data that is not needed or already cached. It allows applications to push data to the infrastructure as soon as it is created, improving durability.

Coupling the infrastructure's unit of management with the client's unit of access, however, has several disadvantages. Most relevant to our work, because each block is managed independently in the infrastructure, to provide non-repudiable binding of owner to data, a client must create a certificate for each block. We have already detailed the performance cost of such an approach.

Other consequences, though secondary to our work, also impact the efficiency of the system. For example, some of the indexing, management, and maintenance costs in the infrastructure are independent of the block size. Thus, managing the small blocks created at the client increases the load on the infrastructure. Also, because each application-level block is indexed independently, clients must issue separate requests for each block they wish to read. Reading an object of even moderate size can flood the storage system with requests.

3 Using aggregation for efficient owner identification

Our goal in this paper is to allow components of a distributed CAS system to identify the owner of each block of data in the system. The solution we present is based on the common practice of associating with each block of data a certificate that asserts the owner of the block. Recognizing, however, the prohibitive cost of providing a certificate for each application-level block, we apply the lessons of previous storage systems research to our domain, using aggregation to improve system efficiency.

The classical filesystems literature demonstrates repeatedly how aggregation can improve efficiency of storage systems. For example, the Fast File System (FFS) [8] increases system performance, in part, by aggregating disk sectors into larger blocks for more efficient transfer to and from disk. XFS [16] further aggregates data into extents, or sequences of blocks, to reduce the size of the metadata and allow for fast sequential access to data. GoogleFS [5] aggregates data from a single file further still into 64 MB chunks, improving performance and per-object maintenance costs for large files typical of their application domain.

More recently, the Glacier [6] distributed CAS system, has shown how aggregation can reduce the number of objects that the system must index and manage. Glacier [6] relies on a proxy trusted by the user to aggregate application-level objects into larger collections. All collections in Glacier are hash-verified and thus cannot be modified after they are created.

3.1 System goals

While Glacier pioneered the use of aggregation in a distributed CAS systems, our work follows from a unique set of goals. We list below the goals that guide our design for using aggregation in content-addressable storage systems.

- **Identification of data owners:** The system must be able to identify the owner of any piece of stored data in a secure and non-repudiable fashion without reference to other blocks. This allows individual nodes in the system to monitor consumption and compute usage fees on a per-user basis locally, without contacting other machines in the system.

- **Self-verifiability:** To protect data integrity, all data must be self-verifying at both the fine granularity of the block and the coarse granularity of the container. Self-verifiability allows clients to verify data locally without rely on secure servers.

- **Incremental update:** For data durability, a client must be able to write data to the system as it is created, without local buffering. Thus, the system must allow clients to add blocks to containers that are already stored in the system.

- **Fine-granularity access:** To conserve bandwidth at the edges of the network, the system must allow clients to access individual blocks without retrieving the entire enclosing container.

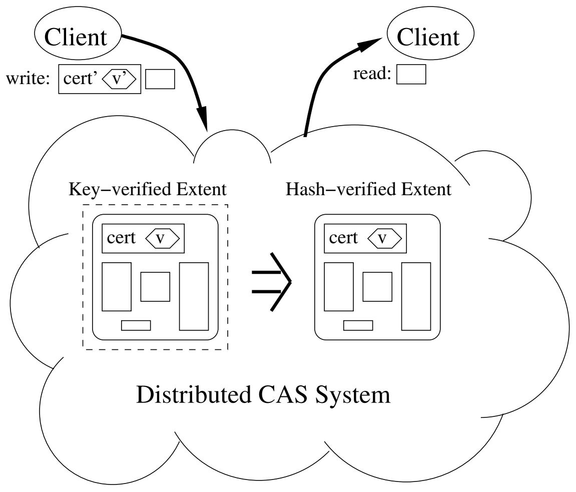


Figure 2. At a high-level, the design aggregates collections of variable-sized blocks into extents. Clients can add data blocks to key-verified extents and read individual blocks from any extent. With each extent, the system stores a certificate that identifies the owner of the data and includes a verifier that describes the data in the extent.

- **Low infrastructural overhead:** The system should not require the infrastructure to index and maintain data at the block level. The design should allow the infrastructure to amortize the cost of maintaining data and verifying certificates over the larger containers.

3.2 Design overview

Figure 2 depicts a high-level design that meets the goals of Section 3.1. The storage system maintains a collection of containers filled with variable-sized application-level blocks called *extents*. Extents are either mutable key-verified objects or immutable hash-verified objects. All data in an extent is owned by a single principal. Associated with each extent is a certificate signed by a client that includes the client's identify. The certificate serves our primary goal of allowing components to identify the owner of the data. The certificate also contains a token that represents the current contents of the extent. The token is a cryptographically-secure *verifier* (e.g. secure hash) that summarizes the contents of the extent.

The system supports incremental update by allowing a client to add data to a key-verified extent. The client submits a write request to the storage infrastructure that includes the data to be written and a new certificate. The certificate contains a new verifier, updated to reflect the data to be added.

To allow different components in the system to manage

Two-Level interface:

status	=	*create*(H(PK), cert);
status	=	*append*(H(PK), cert, data[ ]);
status	=	*snapshot*(H(PK), cert);
status	=	*truncate*(H(PK), cert);
status	=	*put*(cert, data[ ]);
cert	=	*get_cert*(ext_name);
data[]	=	*get_blocks*(ext_name, block_name[ ]);
extent	=	*get_extent*(ext_name);

Table 2. By extending the traditional `put()/get()` interface, CAS systems can support extents and two-level naming.

Certificate contents:

verifier	token that verifies extent contents
num_blocks	the number of blocks in the container
size	the size of data stored in the container
timestamp	creation time of certificate
ttl	time the certificate remains valid

Table 3. The certificate stored with each extent includes fields to bind the data to its owner and other metadata fields.

data at different granularities, the system supports *two-level naming*. In two-level naming, each block is identified not by a single name, but by a tuple. The first element of the tuple identifies the enclosing extent; the second element names the block with the extent. Retrieving data from the system is a two-step process. The system first locates the enclosing extent; then, it extracts individual application-level blocks from the extent. Two-level naming reduces the management overhead incurred by the infrastructure by decoupling the infrastructure's unit of management from the client's unit of access. The infrastructure needs only to track data at the extent level, and the client can still address individual blocks.

3.3 An API for extent-based storage

Table 2 presents an interface for an extent-based storage system like the one described above. All operations that modify data stored in the system require the client to provide a certificate authenticating the change; Table 3 lists the contents of a certificate. To support the extent-based model, the interface extends the traditional `put()/get()` interface, shown in Table 1, defining additional operations for key-verified data.

The new operations allow the system to support extents and two-level naming. Figure 3 shows how the

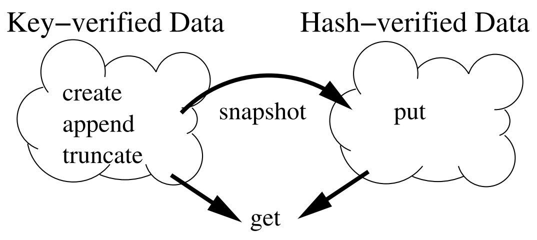


Figure 3. The expanded interface of Table 2 provides different commands to operate on different types of self-verifying data. Commands `create()`, `append()`, and `truncate()` operate on key-verified extents while `snapshot()` converts a key-verified extent into a hash-verified extent.

operations relate to different types of self-verifying data. The `create()`, `append()`, and `truncate()` operations allow clients to manage key-verified extents. The `create()` operation initializes a new, empty container for the client; `append()` allows the client to add data to an existing key-verified extent; and `truncate()` deletes the contents of an extent. The success of these operations depends on the client providing a certificate that contains a valid verifier. To produce a valid verifier on an `append()` operation, the client must know the current contents of an extent. To simplify the task, the system constructs the verifier in a special manner (described in Section 3.4).

Another extension in the interface is the `snapshot()` operation. The `snapshot()` operation converts mutable key-verified extents to immutable hash-verified extents. While the interface derives much of its power from the ability to append data to key-verified extents, it is not feasible to store all data in key-verified extents. Doing so would require clients to manage a large number of key pairs (which is a difficult key management challenge) or extents to grow boundlessly large (which is undesirable because it limits how the storage system can replicate, manage, and transfer data). Depending on the implementation, the system may copy the data to a new set of servers during the `snapshot()` operation. Since the verifier is a cryptographically-secure digest of the contents of an extent, it is a natural name for the new hash-verified extent. We envision that a client-side library responsible for communicating with the storage system will call the `snapshot()` when an extent reaches a specified maximum capacity.

The interface also provides three functions to access data stored in the system. The `get_blocks()` operation is the primary function for reading data, allowing applications to retrieve one or more application level blocks from the system. The `get_cert()` operation returns the certificate associated with an extent, allowing an application to determine the state of data stored in the system. Finally, the `get_extent()` operation supports bulk transfer and retrieval; it returns the entire container including all blocks and the certificate.

Consider the following example to illustrate a typical use of the proposed interface. Upon joining the system for the first time, a client uses the `create()` interface to create a new key-verified extent in the system. The storage system recruits a set of storage servers to host the extent. (For the current discussion, assume the existence of a black box that can identify candidate servers. We call this the *set identification service*.) The set of servers allocate space for the extent and agree to participate in its management.

After an extent has been created, the client can add data using the `append()` operation. A client can append data to any key-verified extent for which it can create a certificate, signed with the proper key, that asserts those changes.

When the key-verified extent reaches a predetermined maximum size, the client (or more likely, a library working on behalf of the client) converts mutable extent to an immutable hash-verified extent using the `snapshot()` interface. Like `create()`, `snapshot()` must query the set identification service; however, the service may return a set containing some or all of the servers already hosting the key-verified extent. If the sets do intersect, the system may consume less network bandwidth in creating a hash-verified version of the extent.

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. Individually, each operation is idempotent, allowing clients to retry until successful execution is assured. In Section 4, we will show how the `snapshot()` and `truncate()` operations can be used to facilitate storing streams of data. Alternatively, an application could use `truncate()` without `snapshot()` to overwrite data stored in the system.

The `append()`, `snapshot()`, and `truncate()` operations that transform 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. A client can have multiple outstanding `put()` operations for a single key pair. The `put()` operation also relies of the set identification service.

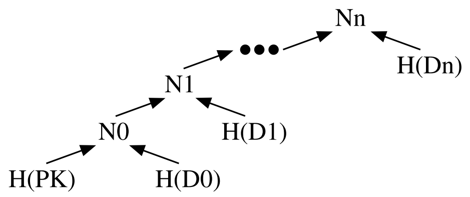


Figure 4. To compute the verifier for an extent, the system uses the recurrence relation $N_i = H(N_{i-1} + H(D_i))$. $N_{-1} = H(PK)$ where PK is a public key.

3.4 Block names, verifiers, and extent names

Central to the API presented in Section 3.3 is the protocol used to name blocks and create extent verifiers. This section describes the approach we use.

Blocks are named with a secure, one-way hash function. The key advantage of this approach is simplicity. Recall that to retrieve data from the network, a client must supply a tuple containing both an extent name and a block name. Using names derived from cryptographically secure one-way hash functions allows clients to verify data locally with information they already possess.

To create extent verifiers, we use a chaining method [7] shown in Figure 4. Assume an extent containing a sequence of data blocks, D_i, with names $H(D_i)$. The verifier is computed using the recurrence relation $N_i = H(N_{i-1} + H(D_i))$, where $+$ is the concatenation operator. We bootstrap the process by defining N_{-1} to be a hash of the public key that signs the extent's certificate. This definition ensures that the names of extents signed by different principals do not conflict.

Using chaining to create verifiers has several advantages. It allows the system to compute the verifier incrementally; when a block is added to an extent, the system must hash only the new data, not all data in the extent, to compute the running verifier. Also, chaining creates a verifiable, time-ordered log recording data modifications. Finally, when an extent is converted from a key-verified object to a hash-verified object during the `snapshot()` operation, the verifier can be used as the new hash-verified name without modification.

3.5 Other benefits

In addition to supporting incremental update and fine-granularity access, extents and two-level naming also allow distributed CAS systems to amortize some management costs. For example, two-level naming reduces the cost storing certificates by amortizing the storage overhead over a

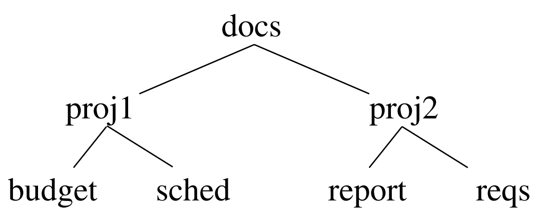


Figure 5. A simple file system used as a running example.

whole extent, allowing systems to devote a greater percentage of their resources to storing user data. Similarly, two-level naming reduces the query load on the system because clients need to query the infrastructure only once per extent, not once per block. Finally, assuming data locality—that clients tend to access multiple blocks from an extent—systems can exploit the use of connections to manage congestion in the network better.

4 Example application: versioning backup

To demonstrate how one might use the interface described in Section 3, we now present a high-level design of an application, namely a versioning filesystem back-up application.

The goal of this application is to convert the tree-based directory structure of the filesystem into a form that can be stored using the two-level API of Table 2. This design writes filesystem data into a sequence, or *chain*, of extents. In this simple design, each directory is stored as a single, variable-sized block containing the directory entries and a verifiable reference to each child; each file is stored as a single, variable-sized block containing the file metadata and data. (A more complete implementation could use the same techniques described in this section to divide large directories and files into blocks.) The application appends file and directory blocks 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. To prepare the key-verified extent for more data, the application truncates the extent. The application inserts a small block of metadata as the first block into each extent. This metadata block serves to link the many extents into a single chain.

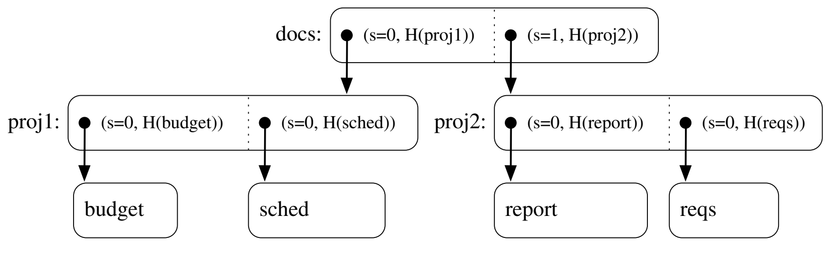
We will assume that the user associates a unique key pair with each filesystem that she wishes to backup. To archive a filesystem—for example, the simple filesystem shown in Figure 5—the back-up application first translates the filesystem into a self-verifying Merkle tree. The form of the resulting Merkle tree for the sample filesystem is shown in Figure 6(a). (We will explain the derivation of the verifiable pointers below.) The translation process is analogous to that used in CFS [2]. Note the similarity between Figure 6(a) and the first version of the object shown in Figure 1.

The key challenge when creating the Merkle tree is in determining the self-verifying 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 initialized with $s = 0$. After executing the `snapshot()` and `truncate()` operations on an extent, the application clones the metadata block from the previous extent, increments the sequence number, and inserts the block in the otherwise empty extent. When creating or updating the 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 storage system 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 in the metadata block in each extent. Each time the mutable extent is made hash-verified and then truncated, the application records the mapping $s_{j-1} \rightarrow (E_{j-1}, M_{j-1})$ where E_{j-1} is the hash-verifiable extent name of the previous extent and M_{j-1} is the block name of the metadata block in the previous extent.

Putting all of these mechanisms together, Figure 6(b) shows the contents of the chain of extents after archiving the first version of the filesystem. The first block in each extent 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. 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 $H(PK)$ is a key-verified extent and can store more data at a later time. The first block in $H(PK)$, its metadata block includes the mapping for the previous extent $E0$. By observing the organization of data in the extents in Figure 6(b), the reader should now understand the derivation of the verifiable pointers in



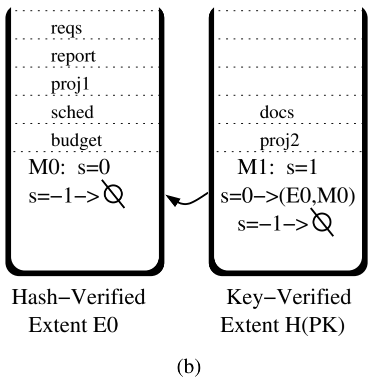
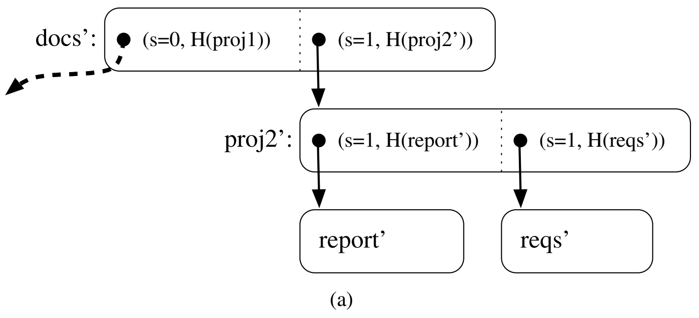


Figure 6. (a) The back-up application translates the filesystem into a Merkle tree. The verifiable pointers are of the form $(extent\_sequence\_number, block\_name)$. Verifiable pointers refer to extent by sequence number because the permanent, hash-verifiable name is not known until later in the archiving process. (b) The Merkle tree is stored in two extents. The first extent, E_0, is filled and has been converted to a hash-verified extent. The second extent, $H(PK)$, is a partially filled key-verified extent. Notice that the first block in extent $H(PK)$ contains metadata including a reference to the metadata block, M_0 of the previous extent.



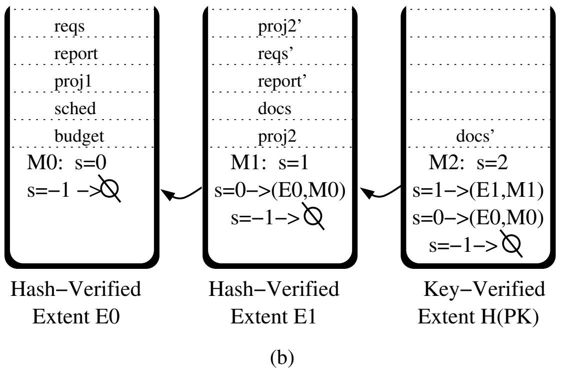


Figure 7. (a) The Merkle tree resulting from translating the updated file system. The dashed pointer indicates a reference to a block from the previous version. (b) The contents of the extent chain after storing blocks of the updated filesystem.

the Merkle tree of Figure 6(a).

Figure 7 shows how the application handles modifications to the filesystem. Assume the user edits the files in the `proj2` directory. Figure 7(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 7(b) shows the contents of the extent chain after recording the changes. The application again filled the key-verified extent which has been converted to the immutable hash-verified extent $E1$. It then re-initialized the key-verified extent, filled it with a metadata block recording the hash-verified names of previous extents and the name of the metadata block in those extents, and appended the remainder of the filesystem data.

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.

5 Prototype implementation and evaluation

To evaluate the API for content-addressable storage presented in this paper, we have built two prototype systems. In this section, we discuss the prototypes and present evaluation results from each.

5.1 Experimental environment

We use several different shared test-beds to run our experiments. The *storage cluster* is a 64-node cluster; each machine has two 3.0 GHz Pentium 4 Xeon CPUs with 3.0 GB of memory, and two 147 GB disks. Nodes are connected via a gigabit ethernet switch. Signature creation and verification routines take an average of 7.8 and 0.5 ms, respectively.

The *test cluster* is a 42-node cluster; each machine has two 1.0 GHz Pentium III CPUs with 1.0 GB of memory, and two 36 GB disks. Signature creation and verification takes an average of 24.9 and 1.4 ms, respectively. The cluster shares a 100 Mbps link to the external network.

Finally, *PlanetLab* is a distributed test-bed for research. We use 300 heterogeneous machines spread across most continents in the network. While the hardware configuration of the PlanetLab nodes varies, the minimum hardware requirements are 1.5 GHz Pentium III class CPUs with 1 GB of memory and a total disk size of 160 GB; bandwidth is limited to 10 Mbps bursts and 16 GB per day. Signature creation and verification take an average of 19.0 and 1.0 ms, respectively.

5.2 Single-server prototype

The first prototype is a single-server implementation that "stores" data in memory. The prototype implements the full API presented in Table 2. We used this prototype to test the completeness of the API by building several small applications, including the back-up application of Section 4.

We also use this prototype to measure the effects of the API on the client. Because the server does not communicate with other servers (or even the local disk), clients receive responses with very low latency. This allows us to isolate the behavior of the client.

The prototype is written in Java. We use the Java implementation of 1024-bit RSA cryptography. The client and server communicate using RPC. In the measurements below, the server is hosted on the storage cluster, and the client is hosted on the test cluster.

The proposed API should improve client write throughput. When storing multiple blocks, the client needs to provide only a single signed certificate to authorize all of the changes. Because computing the signature for a certificate is a relatively expensive operation, this change should result in a significant improvement.

We measure this effect using a simple throughput microbenchmark. In this test, a single client write data as quickly as possible using the `append()` operation. When an extent contains 1 MB of data, the client calls the `snapshot()` and `truncate()` sequence. Each update contains one or more 4 KB application-level blocks; we

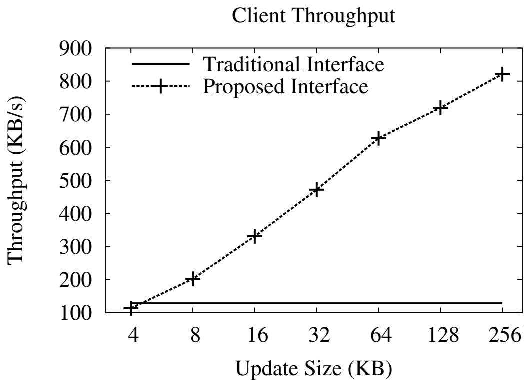


Figure 8. The proposed interface allows clients to amortize the cost of signature creation over several blocks of data. This allows applications to service writes with higher throughput than the traditional interface that must sign every block individually regardless of the size of the update.

measure the throughput of the client as we vary the size of the update. Each test lasts 60 seconds; each data point is the average of 5 tests. Figure 8 shows that clients using the proposed interface can achieve a near-linear improvement in write throughput by amortizing the cost of computing signatures over larger updates. The graph also shows the throughput of a clients using the traditional `put()/get()` interface which requires a separate signed certificate for each block.

5.3 Antiquity: a distributed prototype

We have begun work on a second prototype called Antiquity. With this prototype, we are exploring how to implement the two-level naming interface efficiently in a distributed system. At this stage of development, we can provide a high-level overview of the distributed prototype and present a preliminary performance evaluation.

Figure 9 illustrates the main components of the prototype. The primary role for most machines participating in the system is that of *storage server*. Storage servers are responsible for long-term storage of extents; they also handle requests to append data to existing extents, to convert key-verified data to hash-verified form, and to read data from the system. To ensure availability and durability in the presence of Byzantine failures, the storage servers implement replicated state machines. Storage servers rely on distributed hashtable (DHT) [3] to index and locate data.

To access the system, a client communicates with a nearby *gateway* using RPC. The gateway uses the under-

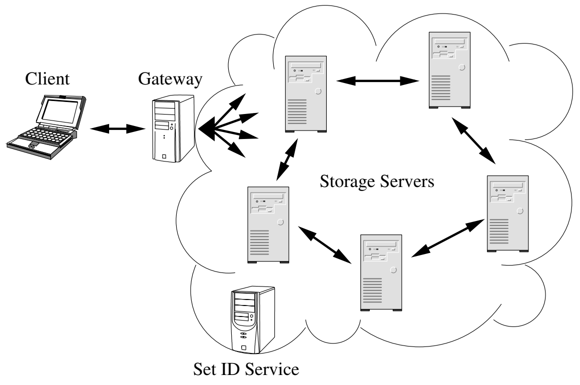


Figure 9. The components and interaction of the Antiquity distributed prototype.

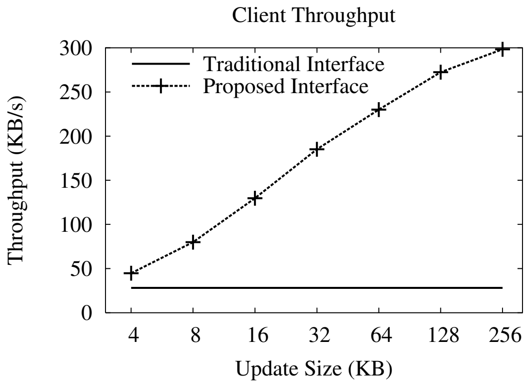


Figure 10. With the distributed prototype, as with the centralized server, client throughput increases with update size by allowing the clients to amortize the cost of producing certificates over multiple blocks of data.

lying DHT to locate the appropriate storage servers to execute the request. For operations that update the state of the storage servers, the gateway forwards the message to each replica. If a client cannot obtain a satisfactory result through a given gateway, it can retry the operation through an alternative gateway.

We described the role of the *storage set identification service* in Section 3.3. Briefly, it identifies sets of storage servers to participate in the management of an extent. The prototype implements this service as a single machine that creates storage sets based on the neighbor lists from the underlying DHT. Design of a more robust and fault-tolerant storage set identifier service is planned.

This prototype is also written in Java. It uses the Java implementation of 1024-bit RSA cryptography and makes extensive use of RPC. It uses the Bamboo DHT [14]. The prototype has been running semi-continuously in two separate installations since early July. We run a small test installation on the storage cluster and a larger installation on PlanetLab. Both installation are configured to replicate each extent on four storage servers.

5.3.1 Performance results

Though the prototype is still under development, we can present preliminary performance results from latency and throughput microbenchmarks. Note that we have not yet optimized this implementation for performance.

The throughput microbenchmark: First, we study the throughput of the system using the microbenchmark described in Section 5.2.

First, consider the throughput for a single client. Figure 10 shows the throughput of a client as the update size varies; this graph is the analog to Figure 8 for the distributed prototype and shows the same trends. The throughput of a single client increases by allowing the client to amortize the cost of producing certificates over multiple blocks of data.

Next, consider the scalability of a *single* gateway. In the distributed prototype, the gateway is responsible for accepting client requests and forwarding those requests to appropriate components in the system. For most operations, the gateway must forward the message to the four replicas that store an extent. Figure 11 plots the write throughput of a single gateway as the number of clients varies. The duration of each test was 6 minutes; we configure the system to use 1 MB extents, 32 KB updates, and 4 KB blocks. For small numbers of clients, the aggregate throughput increases with the number of clients. The aggregate throughput through the single gateway peaks at 1.6 MB/s for 16 clients performing 32 KB appends. With even more clients, the aggregate throughput starts to fall due to processing load and network contention.

The latency microbenchmark: The second microbenchmark measures the latency of individual operations. For each operation, we measure the latency as the time a client begins sending a request to the gateway until the client receives the response. In Figure 12, we report the cumulative distribution function (CDF) of the latency of each type of operation assuming 1 MB extents, 32 KB updates, and 4 KB blocks. Figure 12(a) reports on the system hosted on the cluster; Figure 12(b) reports on the system hosted on PlanetLab.

First, consider the performance on the cluster (Figure 12(a)). In general, the latency of an operation depends on how much data the operation must transfer and whether the operation requires the use of the storage set identifier service. The `append()` and `truncate()` operations are the fastest. These operations transfer little or no data and

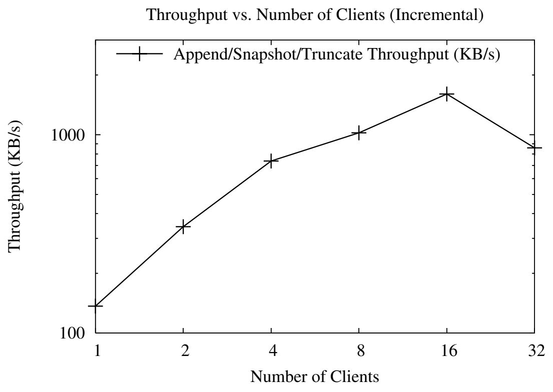
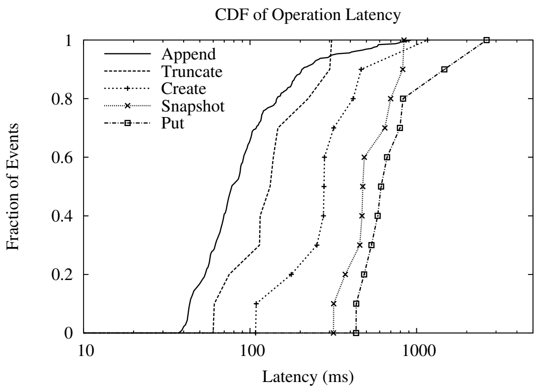
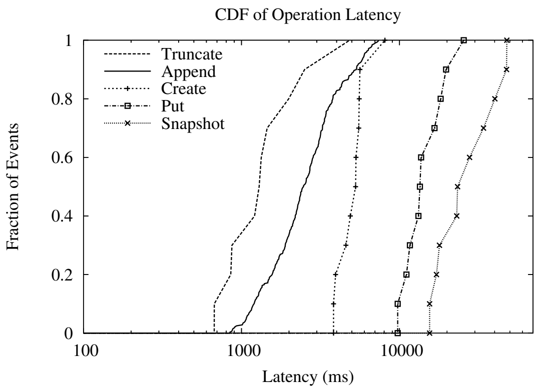


Figure 11. The throughput of update operations through a single storage cluster gateway as a function of number of clients. Extent size = 1 MB, Update Size = 32KB, Block Size = 4 KB, Test Duration = 6 minutes.



(a) Cluster



(b) PlanetLab

Figure 12. The CDF of operation latency for a single client through a single gateway on (a) the cluster and (b) PlanetLab. (Block Size = 4 KB, Update Size = 32 KB, Extent Size = 1 MB)

do not require the use of the set identifier service. The `append()` operation (median latency of 78 ms) is faster than the `truncate()` operation (median latency of 132 ms) because the former requires one sequential write to the local disk whereas the latter requires several writes to delete data that is scattered across the disk, and, in the local cluster, disk latency dominates network latency. The `create()` operation is the next fastest operation. While it does not transfer any data, it must query the set identification server for a set of servers to host the new extent. Finally, the `snapshot()` and `put()` operations are the slowest. Not only must they query the set identification service, but they must also transfer a full 1 MB extent to multiple storage servers. The `snapshot()` operation copies data to a new set of storage servers because the set identification service determines storage sets based on extent name.

The performance trends on the PlanetLab deployment mirror those on the cluster. The `append()` and `truncate()` operations are the fastest. On PlanetLab, however, the `truncate()` operation (median latency of 1293 ms) is faster than the `append()` operation (median latency of 2466 ms) because network latency dominates disk latency. Because of the cost of transferring large extents across the wide area, `snapshot()` and `put()` are very high latency operations; the median latency for `put()` operations is 13,462 ms and the median latency for `snapshot()` is 23,332 ms. We believe that `snapshot()` takes longer than `put()` because the disk is a highly-contended resource in PlanetLab and reading the key-verified extent from disk before transfer can take a significant time.

6 Conclusion

In this paper, we have examined the problem of enabling machines in distributed CAS systems to identify the owners of the data that they store. The basic solution, attaching a certificate to each block, has been proposed before. We have shown how a novel use of aggregation can reduce the costs of creating and managing the certificates and presented an API to support that technique. We are currently building a prototype implementation of this interface. We hope to report on lessons learned from this system soon.

Acknowledgments

The authors would like to thank Dennis Geels, Emil Ong, and the anonymous reviewers for comments that greatly improved the presentation of this paper.

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

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. of USENIX Technical Conference*, June 2004.

[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.