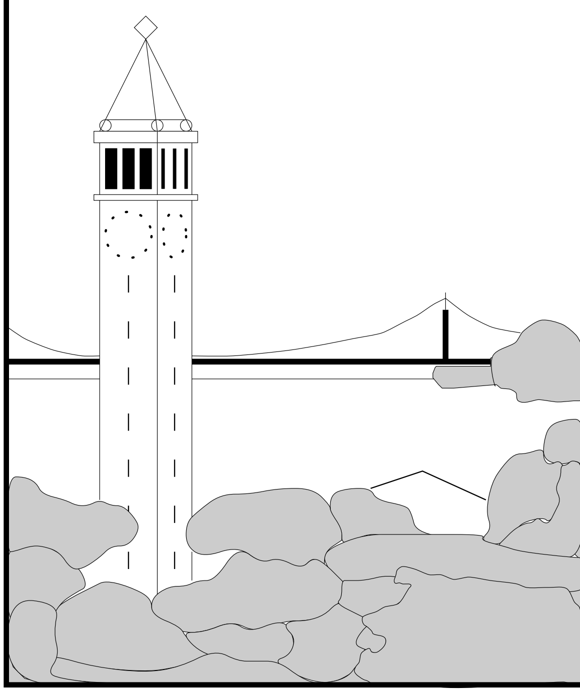
# Silverback: A Global-Scale Archival System

*Hakim Weatherspoon, Chris Wells, Patrick R. Eaton,*
*Ben Y. Zhao, and John D. Kubiatowicz*
*Computer Science Division*
*University of California, Berkeley*
{hweather, cwells, eaton, ravenben, kubitron}@cs.berkeley.edu



**Report No. UCB/CSD-01-1139**

March 2001

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

# Silverback: A Global-Scale Archival System

Hakim Weatherspoon, Chris Wells, Patrick R. Eaton,
Ben Y. Zhao, and John D. Kubiatowicz
Computer Science Division
University of California, Berkeley
{hweather, cwells, eaton, ravenben, kubitron}@cs.berkeley.edu

March 2001

## Abstract

*The rise of ubiquitous computing has created a need for wide-area durable storage. We propose a model and interface for such an archival system, that stores data in a durable, verifiable, available, and self-maintainable manner. We argue that such a system can be created by using novel techniques of erasure codes, secure hashing, and decentralized wide-area location infrastructures to distribute fragments across the wide-area on an arbitrary set of servers. This model allows files to remain available even as servers fail. Finally, we implement Silverback, a prototype archival system using the model that we developed, and measure its performance.*

## 1 Introduction

The world is undergoing a second digital revolution. The first began with the advent of the computer, which radically changed manufacturing, information processing, and scientific endeavor. The second has been far more subtle and at the same time more pervasive – the rise of ubiquitous computing. With the break-neck pace of miniaturization and concomitant decrease in power consumption, computational devices are rapidly finding their way into the very fabric of lives: in cars, walls, clothing, and materials. Even more astonishing is the fact that gigabytes of information can now be deployed in small disposable devices. We can only guess at the ultimate ramifications of this technology.

One consequence of inexpensive storage and computation is that online digital data is rapidly replacing other forms of archival storage. This transformation is at once a great opportunity and a great liability – while digital information is far more flexible to manipulate than (say) paper, it is also easily destroyed. This problem of *information fragility* is reaching critical proportions, since ordinary people are beginning to commit precious, irreplaceable memories (such as photos) to digital form. Consequently, we assert that the most pressing question in the ubiquitous computing revolution is: *where does persistent information reside?*

This paper is about persistent information. The construction of a highly-available persistent storage infrastructure presents many complex issues, such as security, consistency, performance, availability, and archival durability. Systems such as Farsite [1] and OceanStore [7] attempt to address all of these issues simultaneously. In this paper, however, we will focus on one single aspect: version-based archival storage. We will present our musings in the context of Silverback, a global-scale archival system under construction here at Berkeley. The purpose of an archival system is to *durably* store information from many users; durable in this context may mean centuries or millennia. An ancillary desire is that of *time travel*[15], namely the ability to reconstruct the view of an archived document as it appeared at any time during its lifetime. In today's world of ubiquitous computing, we can easily consider the possibility that every person in the world may wish to archive information. Thus we wish to consider systems that scale to $10^{10}$ users and store (perhaps) one *mole* [1] of bytes ($6\times10^{23}$).

The ubiquitous computing vision suggests an *on-line* archival system, rather than something more traditional, such as tape. There are at least three reasons for this. First, a ubiquitous archival system must be able to commit information at high rates from numerous devices. This implies the collaborative effort of many servers writing to spinning storage (for bandwidth) and use of massive redundancy with continuous repair (for durability). Second, ubiquitous devices might be unreliable or possess limited storage; this suggests that devices will make frequent use of online retrieval of archival informa-

[1] A mole of bytes is not as large as it might seem; see Section 3.5.

tion. Finally, tape storage density is not keeping up with the 18-month doubling period of disk capacity. Hence, archiving information to spinning storage is rapidly becoming the *only* option for archival storage.

Stepping back for a moment, we can list several properties that we desire from a global-scale archival system:

- *durability*: Data is stored for long periods of time – decades, centuries, or even millennia.
- *verifiability*: Information should not subject to substitution attacks.
- *availability*: Data is accessible *most* of the time, where "most" is defined in many $9's$ of availability.
- *maintainability*: The system recovers from server and network failures, efficiently incorporate new resources, and adjust to changing usage patterns, all without manual intervention.
- *atomicity*: Each update is applied atomically, without interference from other pending updates.
- *performance*: Response time is bounded and deterministic – consistent with online storage.

This paper will describe how to construct a system that meets all of these requirements while maintaining scalability to billions of users and moles of bytes.

The rest of this paper is organized as follows: First, Section 2 presents essential elements of our archival model, complete with a minimal set of archival operations. Next, Section 3 explores requirements and coding techniques to achieve *deep archival storage*, *i.e.* information that remains unchanged for millennia. Section 4 discusses the design and implementation of the Silverback archival system, while Section 5 explores the performance of this system. Section 6 sets some future directions and Section 7 discusses related work. Finally, we conclude with Section 8.

# 2 Archival Model

In this paper, an *archive* is a linearly ordered sequence of *versions*, where each version is a read-only sequence of bytes. New versions may be added to the end of the version sequence through *update* operations, each of which generates a new version. Data may be read from a specific version through *read* operations. Each version is a stand-alone entity and is abstractly unrelated to any previous versions. For concreteness, an archive might be a *file*, a *directory*, or a *database record*. Archives may also contain the names of other archives.

We will assume the ability to generate globally-unique identifiers (GUIDs); we will discuss the specifics of GUID generation in Section 3.2 and Section 4.3. Archives are uniquely specified by *archive GUIDs (A-GUIDs)*. Every version of every archive will also have a unique *version GUID (V-GUID)*. While V-GUIDs are globally unique across all archives, version-IDs are only unique with respect to a specific archive.

When multiple updates are simultaneously submitted to an archive, an entity in the network, called a *serializer*, must provide atomicity. This serializer takes each update, atomically applies it to the archive (including any operations required to make this update durable), then generates a new V-GUID. Consequently, when a client seeks the most recent version of an archive, a request is sent to the serializer to obtain the V-GUID of this latest version. More generally, the system provides a mapping such that, given an A-GUID and some version information (for instance, a timestamp), the GUID of a particular version can be retrieved.

A global-scale archival system must include a routing infrastructure capable of forwarding requests to appropriate servers. Requests for different types of GUIDs are handled differently, so all requests will be tagged with their type. While the nature of the routing layer is an implementation detail, a good implementation can significantly improve the performance of the archival system. Additionally, a *caching* layer masked by the routing layer can greatly improve the latency to data; note that cache consistency is greatly simplified since all requests for information are against *specific* versions of an archive.

## 2.1 Archival Interface

In this section, we will list the operations that must be present in an archival system. First, to generate a new archive, a user must specify a human-readable name for the archive, the user's identity as a public key, and a public/private key pair for the signing of commits:

```
create(name, identity, keys) ⇒ A-GUID
```

An empty first version is produced as well. We assume that users keep the A-GUID of *one* "root" archive with them at all times. This can be used to construct an arbitrary, hierarchical naming structure in which to store mappings between user-relative names and their associated A-GUIDs.

To read data, we assume that a client provides a V-GUID and a specification about which data to read:

```
read(V-GUID, offset, length) ⇒ data
```

This operation returns data from the specified version.

New versions can be either unique or derived from previous versions. We highlight three distinct update operations in the following: `write()`, append(), and `modify()`. Each of these operations generates a new version of an archive, returning a V-GUID in the process. First, to generate a completely new version a `write()` operation is used:

```
write(A-GUID, data) ⇒ V-GUID
```

This operation commits a new version of the archive. A second type of update treats the archive as a permanent log:

```
append(A-GUID, data) ⇒ V-GUID
```

This operation appends new information to the end of the most recent version. Note that a persistent log is a fundamental component of many distributed algorithms. Finally, we provide the ability to derive a new version from a previous version through the `modify()` operation:

```
modify(V-GUID, offset, data, allowbr)
  ⇒ V-GUID or nil
```

The `allowbr` flag denotes whether or not we allow this operation to generate a version branch. Branching would occur if the V-GUID is not the latest for the archive at the time this operation is serialized; if this happens and the `allowbr` flag is set to `false`, then the `modify()` operation will return nil[2]. It is up to user to place additional semantics on top of branches if they occur.

Finally, we provide `query` operations to acquire specific version information from a given archive. The first form returns the latest V-GUID from a given archive:

```
query(A-GUID) ⇒ V-GUID
```

This returns the latest V-GUID at the time that the request reaches the serializer. The second form of `query` is more general:

```
query(A-GUID, Spec) ⇒ V-GUID
```

This takes a specifier for a version (which may be a timestamp, version-ID, or other means of identifying a version) and returns an appropriate V-GUID.

## 2.2 Example

Figure 1 provides a pseudo-code description of how to use the interface of Section 2.1 to back-up a file system. Although simplistic, this example illustrates a number of important points. First, we generate a separate archive for every *directory* and every *file*. Second, directories are application-level associative mappings between user-relative names and the A-GUIDs for that name. This example provides a very primitive linear array for name resolution; use of the `modify()` operation would permit more efficient hash-tables to be constructed. Note also that this example provides no file deletion operation. Third, recognition of changes is done by the application (in this case through timestamps), not by the system. Finally, by reusing the A-GUID for a file with each change, we associate all versions of a file with one another.

```
ARCHIVEFS(dir, A-GUID)
  V-GUID ← query(A-GUID);
  foreach (file in dir)
    name = "dir/file";
    record ← SEARCH(V-GUID,name);
    if (isnull(record))
      FILEGUID ← create(name,identity,key);
      record ← { name, FILEGUID, 0 };
      append(A-GUID, record);
    else
      FILEGUID ← record.aguid;
    endif
    if (record.timestamp ≠ stattime(file))
      record.timestamp ← stattime(file);
      append(A-GUID, record);
      if (isdirectory(file))
        ARCHIVEFS(name, FILEGUID);
      else
        write(FILEGUID, contents(file));
      endif
    endif
  endfor
```

Figure 1: *Archival File Backup:* Inputs are a top-level directory and A-GUID for that directory. We maintain a simple name → (A-GUID, timestamp) mapping as a linear structure that is traversed by the SEARCH() function. Changes to this mapping are performed by appending a new mapping for the given name; this is simple but inefficient. Mappings are inserted as records that are triplets with three fields: *name*, *aguid*, and *timestamp*.

# 3 Deep Archival Storage

Given the model in Section 2, we now describe the mechanisms which make a wide-area archival system possible. These mechanisms must provide high levels of durability and availability, while ensuring users' data in-

[2] Note that setting `allowbr` to `false` allows read-modify-write operations on low-conflict archives. Higher rates of conflict can be handled with an append-only logging methodology.

tegrity [3]. In particular, we must use explicit redundancy and geographic distribution of data to protect data against inevitable hardware failures and malicious threats, and use cryptographically secure mechanisms to guarantee the immutability of read-only data.

## 3.1 A Case for Erasure Codes

The most common methods used to achieve high durability of data are complete replication and parity schemes such as RAID [16]. The former imposes extremely high storage overhead (size in storage is several factors larger than original data), while the latter does not provide the robustness necessary to survive high rate of failures expected in the wide area. Erasure codes are an alternative to these classic mechanisms which provides extremely high durability and availability without imposing an unreasonable overhead in storage space.

Using erasure codes, a user can break up an object into $n$ *fragments* and recode them into $kn$ fragments, where $k > 1$. Such encoding increases the size of the data by a factor of $k$. We refer to $1/k$ as the *rate* of encoding. The key strength of erasure codes is that the original object can be reconstructed from *any* $n$ fragments.

There are a number of erasure codes with different performance characteristics. Some, such as Tornado Codes [8], scale linearly with the number of fragments. Tornado codes in particular can reconstruct an object very quickly, but do so only with high probability and only in the presence of slightly more than one half (for rate one-half) of the fragments. These properties make Tornado Codes appropriate only when large numbers (hundreds to thousands) of fragments are being produced. The "Reed Solomon" [11] family of erasure codes are popular, but have encoding time scaling quadratically, making them practical only for relatively small objects. Because we encode small blocks in Silverback, we chose an effiecient version of Reed Solomon called Cauchy Reed Solomon codes.

### 3.1.1 Availability

Erasure coding exploits the statistical stability of a large number of independent components. The availability of an object increases with the number of fragments and rate of encoding. As the fraction of the fragments needed to reconstruct an object decreases, probability of reaching enough fragments for reconstruction increases. Similarly, as the number of fragments for an object grows, the probability that not enough fragments are available for reconstruction due to network partitions and machine failures decreases. The availability of an object can be summarized as below:

| | |
|---|---|
| $P_o$ | probability that an object is available |
| $r_f$ | maximum safe number of unavailable fragments |
| $f$ | total number of fragments |
| $n$ | total number of machines in the world |
| $m$ | number of currently unavailable machines |

$$P_o = \sum_{i=0}^{r_f} \frac{\binom{m}{i}\binom{n-m}{f-i}}{\binom{n}{f}} \qquad (1)$$

This formula states that the probability that an object is available is equal to the number of ways in which we can arrange unavailable fragments on unreachable servers, multiplied by the number of ways in which we can arrange available fragments on reachable servers, divided by the total number of ways in which we can arrange all of the fragments on all of the servers.

For instance, with a million machines, ten percent of which are currently down, simply storing two complete replicas provides only two nines ($0.99$) of availability. A $1/2$-rate erasure coding of a document into $16$ fragments gives the document over five nines of availability($0.999994$), yet consumes the same amount of storage. With 32 fragments, the availability increases by another factor of 4000, supporting the assertion that *fragmentation increases availability*. This is a consequence of the law of large numbers.

### 3.1.2 Durability

An analysis of the MTTF of fragments and fragmented objects is also essential in motivating distributed archives. Disk failure distributions obtained from [10] and shown in Figure 2 indicate that while disks have some infant mortality, a high number of them survive the

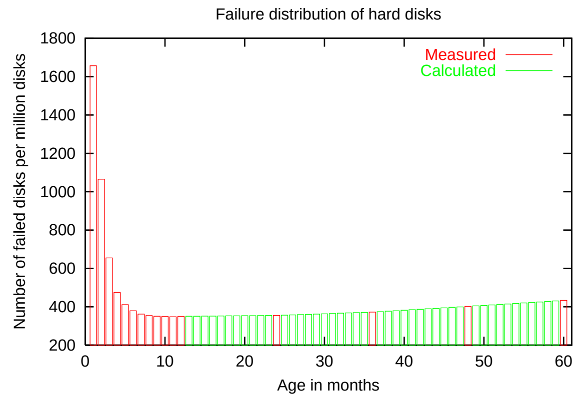



Figure 2: Disk failure distribution

[3] Privacy of data can be enforced by end-to-end encryption.

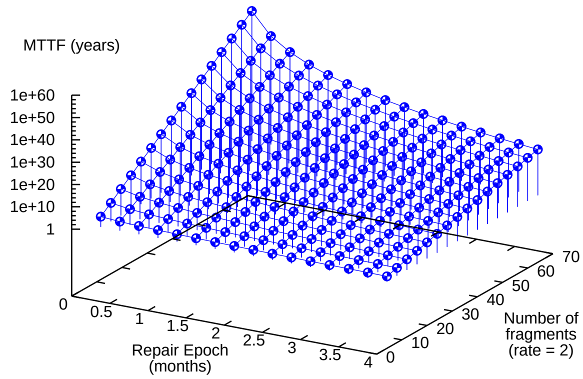



Figure 3: Mean Time to Failure of a Block

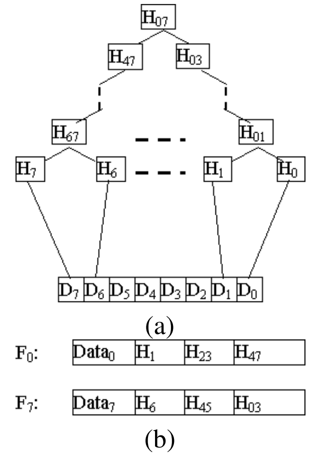



Figure 4: A *Verification Tree* is a hierarchical hash over the fragments of the blow. The top-most hash is the block's *GUID*.

duration of their service life of five years. Using these numbers, we determined that the age of a randomly selected disk was uniformly distributed from zero to sixty months. This allows us to calculate the expected lifetime of a fragment after dissemination, and ultimately to calculate the mean time to failure of an entire object. We accept the simplifying assumption that all fragments would fail independently, no servers behave maliciously, and that the repair mechanism would (if the object was still alive), periodically reconstruct and re-disseminate every fragment. Our parameters include the rate of encoding ($1/2$), the number of fragments (varying from $4$ to $64$ in increments of $4$), and the length of the repair epoch (varying from $1/4$ months to $4$ months in increments of $1/4$ month).

Figure 3 shows the results of our calculations. The scale of the MTTF axis is exponential, indicating that the MTTF of objects scales super-linearly with the inverse of the repair epoch. A more exciting result is that the MTTF of objects scales exponentially with the number of fragments. With twelve fragments and a repair time of two weeks, we see that an object has an MTTF of over one hundred billion years.

## 3.2 Verification Scheme

Erasure coding requires the precise identification of failed or corrupted fragments. As a result, the system needs to detect when a fragment has been corrupted and throw it away. We therefore introduce a secure verification scheme for fragments.

For each encoded block, we create a verification tree over its fragments. Figure 4(a) is a binary verification tree. The scheme works as follows: We produce a hash over each fragment, concatenate the corresponding hash with a sibling hash to produce a higher level hash, we continue the algorithm until there is a topmost hash. We then store with each fragment all of the sibling hashes to the topmost hash, a total of log $n$ hashes, where $n$ is the number of fragments. Figure 4(b) shows the contents of a "dissemination fragment". The hash at the root of the tree is the name or GUID of the block. To ensure that other data does not hash to the same GUID, we use the SHA-1 [9] secure hash.

On receiving a fragment for recoalescing, a client verifies it 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 until there is 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.

## 3.3 Dissemination

The serializer must store fragments in a manner that avoids correlated failures. Otherwise, the statistical advantages of erasure coding becomes greatly reduced. Correlated failures can occur, for instance, within geographic regions or administrative domains. Avoiding correlation is important enough that we devote a complete section to this in Section 4.3.1.

## 3.4 Repair

Crucial to the implementation of a durable archival system is the use of efficient, robust repair algorithms. In a distributed archival system with the previously discussed properties, there are three basic types of repair mechanisms: *local fragment maintenance*, *passive detection*, and *active sweep*. Servers can perform local fragment maintenance by periodically checking the integrity of local fragments. When servers fail, however, the system requires a distributed scheme to detect loss of fragment availability. If neighbor nodes monitor their peers, they can inform interested parties when certain fragments are no longer available. Yet even this scheme falls short in the presence of malicious servers. A periodic sweep of all the fragments by some entity is required to completely protect against a catastrophic loss of data.

Passive detection, notification, and active sweeps are simplified by the existence of some entity charged with the survival of a user's data. A *Responsible Party* is a service provider paid by users that plays such a role. Because it is a service provider, the Responsible Party can remain online continuously, and thus receive notifications of fragment failures as well as periodically sweep through users' data.

### 3.5 A Mole of Bytes

Humanity currently generates an estimated $1.5$ exabytes of data per year. An archival system should be durable on the order of $1000$ years, so a capacity of over $10^{21}$ bytes is desirable. This number is close to one *mole* ($6 \times 10^{23}$) of bytes. The mechanisms described in the preceeding sections, combined with the increasing capacity of disks and networks, make it possible for the first time to postulate the storage and maintenance of a mole of bytes. Put another way, what are the resources needed to prevent the loss of a single byte in a mole of bytes for one thousand years? Assuming that encoded objects fail independently, the analysis performed for a single object's MTTF can be extended to any number, $b$, of objects simply by taking the $b^{th}$ root of the desired probability of failure (in our case, .5).

Using the repair scheme described in Section 3.1.2, with sixty-four total fragments, a rate $1/4$ erasure code, and a repair epoch of ten months, a mole of bytes (broken up into 4kB blocks, can be expected to fail after twenty-seven thousand years. The repair mechanism for a mole of bytes requires that one billion billion bits be transferred per second. If we assume that there are ten billion machines in the world, the bandwidth required per machine is therefore one hundred Mbs. This number is within one order of magnitude of today's network capacity, indicating that a wide-area archival system can successfully scale to service one mole of bytes. Scalability becomes even more feasible when more efficient repair schemes are used — schemes which only transfer fragments which require reconstitution. Additionally, as network bandwidth grows with Moore's Law, increasing numbers of bytes will become maintainable.

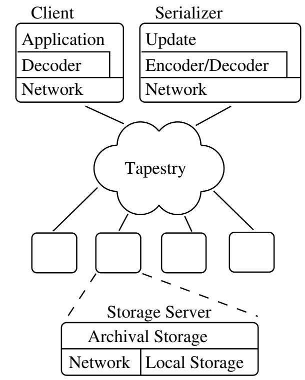



Figure 5: Archive Architecture

## 4 Implementation

Figure 5 shows the architecture of the prototype Silverback archival system. The wide-area location and routing infrastructure shown in the middle of the architecture is Tapestry [17], a wide-area routing and location infrastructure discussed in greater depth in Section 4.2. All nodes in our implementation communicate through Tapestry, so that the set of storage servers (shown at the bottom of the figure) which store fragments from a single object can all be contacted by a single message. Each node in the network can serve as a client, a serializer, a storage server, or as any combination of these roles. We have shown nodes taking on single roles for simplicity.

In this section we discuss our implementations of the components and interfaces outlined in 2 and 3. We begin our explaination with the archival object structure in 4.1, then Tapestryin 4.2, next the serializer in 4.3, and finally maintaining the system in 4.4.

### 4.1 Archival Object Structure

Figure 6 presents graphically the central data structure used by the Silverback archival layer. At the heart of this structure is the *data B-tree*, a conventional B-tree with blocks of data stored at the leaves. This structure is shown in the figure inside the dashed box. As in databases, all blocks of the B-trees need not be colocated at all times. To ensure the integrity of blocks prior to archival, the data B-tree uses secure SHA-1 hashes, organized as in Section 3.2, to refer to other nodes in the

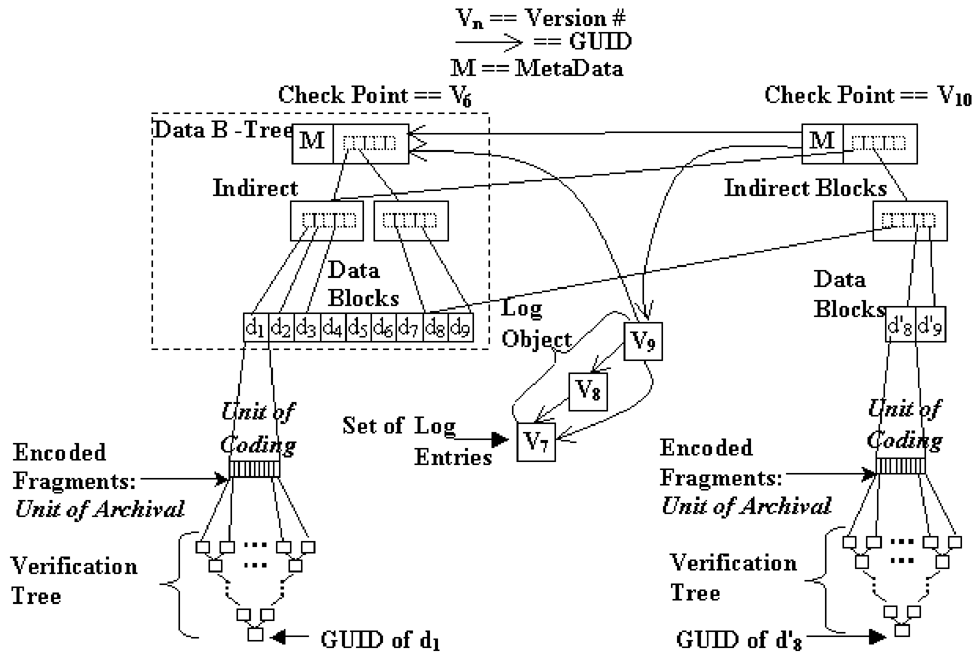



Figure 6: The data object structure.

tree.

The data B-tree uses a copy-on-write mechanism. The serializer operates on the data B-tree, utilizing the copy-on-write mechanism, to transform the object from one consistent state to the next. Because of the properties of the copy-on-write mechanism, a small update only requires changing a number of blocks equal to the height of the tree and both the old version and the new version are accessible by referencing their distinct root nodes.

As the serializer produces new versions of the data B-tree, it passes the new version to the archival layer for *checkpointing*. A *checkpoint* is a self-contained object in the archive; that is, it can be reconstructed without reference to other versions or the log (to be discussed below). To create a self-contained archive, the archival layer descends the tree archiving blocks as in a pre-order traversal; that is, all child blocks are archived before their parent. For archive, a block is erasure encoded to produce a number of *encoded fragments*. These encoded fragments are hashed using the SHA-1 algorithm to produce a verification tree, as described in Section 3.2. The hashes needed to verify an encoded fragment are combined with the encoded fragment to produce a *dissemination fragment*, or simply *fragment*. The root hash produced by the verification tree becomes the name, or GUID, for the block, and each dissemination fragment of that block is named by that GUID. This GUID is stored in the parent block so that a valid, permanent reference is archived when the parent is archived. In Figure 6, versions $V_6$ and $V_{10}$ have been checkpointed and the full encoding process has been shown for blocks $d_1$ and $d'_8$.

Even with the copy-on-write optimization, any change to a data object requires encoding and archiving at least a number of blocks equal to the height of the tree. For small updates, even this seems inordinantly expensive. To avoid the overhead of archiving blocks for each update, most of which will be small, the archival layer does not archive every version with a checkpoint as described above. For versions which are not checkpointed, the archival layer inserts a log entry. Log entries describe how to alter a checkpointed version of a data object to restore an intermediate version. In Figure 6, the archive can reconstruct version $V_7$ by applying log entry $V_7$ to checkpointed version $V_6$; version $V_8$ is recovered by applying log entry $V_8$ to version $V_7$, and so on. Version $V_{10}$ is recovered by accessing the checkpoint for version $V_{10}$ directly, without any reference to the log. The frequency of checkpoint is variable and could even be introspected upon, based on the frequency and size of updates.

## 4.2 Tapestry

Objects in Silverback are free to reside an *any* server. While this provides tremendous flexibility for replication, caching, and migration policies, it makes the task of finding object much more difficult.

This task falls to Tapestry, Silverback's routing and location subsystem. Tapestry is an IP overlay infrastruc-

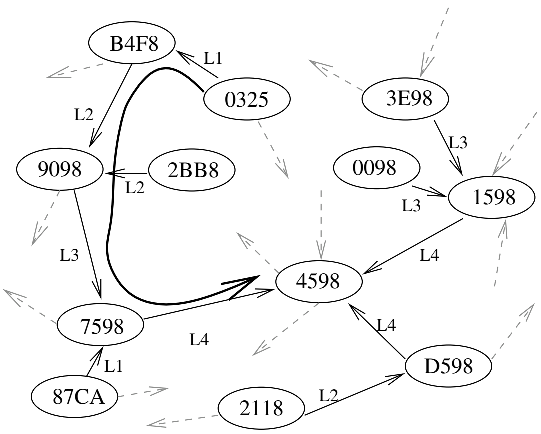



Figure 7: *Tapestry routing example.* This figure shows a route that might be taken by a message originating from Node 0325 destined for Node 4598.

ture that uses a distributed, fault-tolerant data structure to explicity track, by GUID, the location of all objects in the network. Each GUID maps deterministically to its unique *root* node in the network. A storage server *publishes* an object's by routing via Tapestry (as described below) from itself to the root node, depositing pointers to the object's location at each Tapestry hop along the way. To find an object, a client maps the object's GUID to its root node and routes to that root node. Tapestry routing is hierarchical, and multiple routes to a destination form a tree rooted at the destination. This provides locality properties, since the path taken to root by any client searching for an object nearby will with high probability intersect the path taken by the storage server at publish time. The Tapestry location client routes directly to the storage server when it finds a location pointer for the object it is looking for. If such a crossing does not occur, the route will eventually reach the object's root which also holds a pointer to the object.

The Tapestry routing scheme is based on the hashed-suffix routing structure originally presented by Plaxton, et. al. [12]. It uses local neighbor maps to incrementally route message to the destination address digit by digit. For example, a node with address 0325 searching for a node with address 4598 would follow a route along nodes with address $* * *8 \Longrightarrow * * 98 \Longrightarrow *598 \Longrightarrow 4598$, where $*$'s represent wildcards. This process is illustrated in Figure 7.

A key property of Tapestry location that the archival layer utilizes is that of locality. Since all fragments belonging to a block are named by the block GUID, a search for fragments for reconstruction routes to the "root node" of the block, returning the closest $N$ blocks to satisfy the threshold.

## 4.3 Serializer

The *serializer* is the only node in the system which is capable of encoding objects for storage. Since it is responsible for ordering updates to an archive, it must possess a copy (or a slice of a copy) which it alters according to writes it receives from the client. Also, there is a *unique* serializer on a per archival object basis. Therefore, updates to different archival objects will communicate with different archives. The seperations allows the archival system to scale to the wide area.

The client application code communicates through the network to the serializer update mechanism to get V-GUIDs and to write changes to archives. Note that the decode layer of the client is capable of communicating directly with the archive; it can send requests for versions through Tapestry and is therefore capable of accessing read-only copies of data without contacting the serializer, so long as it has the versions' V-GUIDs.

### 4.3.1 Dissemination

Once the serializer receives an update request the archive system must store fragments in a manner which achieves independence of fragment failures. Correlated failures can occur as a result of similar geographic location or administrative domain of the storage servers. Independent failures will not pose difficulty to a *random* dissemenation scheme if the pool of storage servers is *large*. For example, it is extremely unlikely that twenty-four randomly placed fragments of thirty-two fragment set will all be located on the West Coast.

We define a simple randomized algorithm that ensures independence of fragment placement by avoiding a *catastrophic collision* with extremely high probability. In the rare case that the algorithm fails, only the co-located fragments need redissemination.

**Definition:** A *catastrophic collision* is where $(k-1)n$ or more fragments are co-located.

**Input:** $kn$ fragments $\{0,\ldots,kn-1\}$.

**Output:** Disseminate fragments to servers on our network s.t. $kn$ fragments *satisfy* a system of $m$ *constraints*, where a *constraint* is a rule that states a *catastrophic* (i.e. $(k-1)n$) amount of fragments cannot share a given *property* (i.e. geographic region, domain, admin, etc...).

**Algorithm:**

1. $\forall_i$ pick a server $\pi(i)$ *u.a.r.* and disseminate $i$ to $\pi(i)$.

2. $\forall_i$, $\pi(i)$ sends back an $ack_i$ with its *properties*.

3. Analyze each $ack_i$
   If all $m$ *constraints* are *satisfied* - Done.
   Else pick a *maximal* subset of $kn$ fragments s.t. all $m$ *constraints* are *satisfied* and redisseminate fragments not in the subset.

**Analysis:**

**Claim:** $Pr[\textit{catastrophic collision}] \leq \frac{1}{\ell^{(k-1)n}}$.

**Proof:** We generalize our analysis to a system with only one constraint $\rightarrow$ property, but the property has many values $l$. Given a *property*, distribution can be made uniform. Simply stated, the probability that a fragment $i$ is stored on a server with property value $\ell_j$ is $\frac{1}{\ell}$. The probability that a second fragment $i+1$ is stored on the same server with property value $\ell_j$ is $\frac{1}{\ell} \times \frac{1}{\ell} = \frac{1}{\ell^2}$. Hence the fragments $(k-1)n$ fragments are stored on the same server with property value $\ell_j$ is $\frac{1}{\ell^{(k-1)n}}$. ■

**Example:**

As an example, given *geography* as property with 32 unique cities or 32 values, $n = 16$ and $k = 2$, where a *rate* $1/k$ encoding requires a minimum set of $n$ fragments to reconstruct the original object. The $Pr[\textit{catastrophic collision}] \leq \frac{1}{32^{16}}$.

Given $10^{10}$ users and $10^4$ objects per user, the probability of a catastrophic collision in the system is $\frac{10^{14}}{32^{16}} \approx \frac{1}{10^{10}}$

#### 4.3.2 Tombstones

Our system allows serializers to be taken offline after long periods of no use. When a serializer is taken down, it first stores the mappings from its A-GUIDs to its V-GUIDs in *tombstones*, so named because the serializer puts them in place in the event of its own death. A tombstone for a particular archive is named and located by that archive's A-GUID, and contains the public key of the serializer, the name of the archive, and the latest V-GUID of the archive. It also contains a signature over this information which is produced by the serializer's private key. Thus, a tombstone is verifiable by its archive's A-GUID: one need simply hash over the concatenation of the public key and human-readable name to verify the public key against the A-GUID, and then use the public key to verify the tombstone's signature. When the serializer produces new tombstones for an archive, it routes them to the storage servers containing the old tombstones for that archive. These servers verify the new tombstones and then overwrite their older counterparts.

In the presence of a responsible party, the user is able request to send a request to the archival system for a file even if no serializer is currently active. The request will be routed to the tombstones for the archive, which in turn are sent to the responsible party. The responsible party spawns a new serializer which begins servicing requests.

### 4.4 Maintenance

In order to provide long term availability in a dynamic environment, Silverbackshould survive changes to the physical infrastructure over time with minimal external management. During normal operation, new nodes regularly become available to the network, while other nodes exit the system for maintenance or due to failure. Silverbackprovides mechanisms that seamlessly integrate new nodes, extract exiting nodes, and recovers from link and node failures, all with minimal external intervention.

#### 4.4.1 Integration

To make a new server available for archival storage, it only needs a network connection and the location of one known Tapestrynode. The server then weaves itself into the routing and location layer, advertises that it is ready to store new fragments. Tapestry includes a set of distributed algorithms to support automatic server integration and removal without human administration. To integrate into the network, a server populates its neighbor maps by copying and optimizing neighbor maps from nearby nodes that share portions of its address. The node completes integration by notifying nearby nodes of its existence so that neighbors may include it in their own neighbor maps.

If a server cannot store new fragments due to storage constraints, it may cease this advertisement at any time, either through manual intervention or introspectively. Similarly, serializers can advertise their availability through Tapestry when they are introduced to the network.

#### 4.4.2 Removal

A server can be removed from Silverbackif it becomes obsolete, needs scheduled maintenance, or experiences component failures. When possible, the server runs an optional shutdown script which proactively informs the routing layer of its imminent departure. Neighbor nodes which receive this message can update their routing maps to eliminate references to the departing node. Location pointers to fragments and fragments themselves can be moved off the server, or regenerated after its departure. Since Tapestrynodes utilize a *soft state* fault-handling model, nodes use regular *heartbeat* beacons to inform neighbor neighbors of their existance, while object storage servers republish their objects on a regular basis. In the absence of a departure announcement, the routing layer will detect and correct for the server's absence. First, its neighbor nodes will detect its absence and update their routing maps. Nodes which depend on the server for routing will promote secondary routers and find new backups. Second, object pointers will be republished. Finally, the Tapestryroot nodes will miss regular advertisements for fragments, and if redundancy falls below acceptable levels, send notification to the objects' responsible parties or to other storage nodes, who will then ensure regeneration and redissemination of fragments.

#### 4.4.3 Fault-handling

To maintain objects such as fragments and tombstones, our archival system makes use of the repair methodology discussed in Section 3.4, and relies on built-in fault-handling mechanisms in Tapestry.

At a lower level, Tapestryattempts to recover from routine failures, and notifies Silverbackof application-level failures. For example, to tolerate routing failures, a local routing map which determines the next hop location contains several secondary routes in addition to the primary route. Failures on the primary link result in messages switching transparently over to secondary routes. To reduce the impact of location failures, Tapestry publishes an object multiple times with different names. This greatly improves the probability that a node can find an object's location, even in the presence of full network partitions.

As part of its failure detection, Silverbackstorage servers on the network periodically issue *heartbeat* beacons for each of its fragments. These beacons are propagated through Tapestry in exactly the same fashion as the initial publication of a new object (described in Section 4.2). Our analysis in Section 3.1 show that maintenance need only occur on the order of months. Consequently, heartbeats can be relatively infrequent, minimizing the overhead of the bandwidth they consume. When the root node of a particular object has failed to receive a beacon for a given fragment after several heartbeat periods, it sends notification to the object's responsible party, which will then reconstruct and redisseminate as necessary.

Finally, an active maintenance sweep must be performed to protect data against adversaries in the wide area. This task can be performed by a user's responsible party, which can be trusted, but may have limited CPU and bandwidth resources. Alternatively, the storage servers themselves can perform this task. While they have less contended resources, are not completely trustworthy. This tradeoff can be explored depending on the degree to which users trust the wide area and on resource availability. Note that the responsible party is only needed if the user desires someone to be responsible for a given operation, since storage servers are also capable of repair. Therefore, responsible parties are not essential to an archive model or to our implementation.

### 4.5 Scalability

We now attempt to analytically evaluate our design and implementation on the scalability metric. With the large scale of data storage discussed in Section 3.5, any point of centralization or potential bottleneck will hamper the scalability of the overall system.

To understand how the Silverback system scales, one needs to recognize the pervasiveness of data independence throughout our design. Our key approach is to remove any central authority which could buckle under heavy load. Each archive contains all versions of a single file, and each archive is associated with its own serializer. This implies that serializers can be distributed across all available nodes, and load-balancing can be achieved on a fine granularity. The other key mechanism, Tapestry, takes a fully decentralized approach to routing and location [17]. As a result, granularity is maintained on the order of files, and the lack of centralized mechanisms allows the system to scale up with the amount of available resources.

## 5 Performance

In this section, we evaluate our system by focusing on three metrics critical to the operation of an archival system: *data expansion*, *time to durability*, and *read performance*. By data expansion, we mean the factor by which the size of the data increases in the archive. For instance, if a system uses an erasure code with rate one-half, the expansion factor is at least two. Time to durability refers to the time it takes the system to take new bytes from a user and finish storing them in a durable fashion. Finally, read performance is the time between when a user issues a request for bytes of a version of an object and the time that he receives those bytes.

### 5.1 Initial Prototype

To explore the performance of Silverback, we constructed a prototype file system backup service on top of Silverback. Refer back to figure 5 to see where the client application level code resides in the system. We placed NFSI [6], a java-based NFS [13] server on top of our implementation of a serializer. Our serializer resides on the same machine or LAN as its clients would to limit the network latency of requests to the serializer.

Users can mount a directory stored in Silverback just like any NFS directory. The kernel processes file system calls from the user and passes them through the vnode layer to the Silverback server, which caches files and directories on a local backend directory. Silverback services requests by accessing files in its local cache or, when necessary, by accessing the archival layer.

Files and directories in Silverback are named just like files in standard UNIX file systems. Users can append version numbers or timestamps to these names when they wish to access past versions in a similar syntax to that used in the Elephant file system [14].

Finally our filesytem code is similar to the *Archival Filesystem Backup* presented in figure 1. The only difference is instead of implementing the expensive SEARCH() function, we developed a more efficient *meta-object library*, called *MLib*'s. A MLib is metadata for a directory and contains an entry for every child of the directory, and each entry, in turn, stores the version number, time of creation, and V-GUID for every version of its file.

In the context of the discussion above and in the beginning of section 5, we analyze Silverback in terms of *data expansion*, *time to durability*, and *read performance* metrics in the following sections by a combination of results from analysis, simulation, and measurement from our prototype. To drive our microbenchmark results, we used access patterns from the Andrew Benchmark [5], a standard file system benchmark which tests all major components of a file system. We ran our storage servers on a large collection of campus-sized clusters [2], and our routing infrastructure simulated wide area latency by varying the time to these nodes to up to 200ms. Running the benchmark against NFSI without Silverback took 188 seconds.

## 5.2 Storage Overhead

An important measure of the efficiency of an archival system is the data expansion, or the number of bytes stored for every byte the user commits. In Silverback, every archive version has some metadata and a B-tree for data. Data blocks are currently exactly 4kB large, but indirect blocks are no larger than necessary (they are not padded to 4kB). Root blocks are always indirect blocks, even for files smaller than 4kB, and have appended to them the metadata for the version. The Cauchy encoder we use pads the top block plus the metadata to 896 bytes (the smallest possible even multiple of 128) before generating the block's 32 fragments. Each fragment has appended to it 24 bytes of encoding metadata and 120 bytes of verification hashes (see Section 3.2). Thus, the top block for the smallest possible file is 6400 bytes.

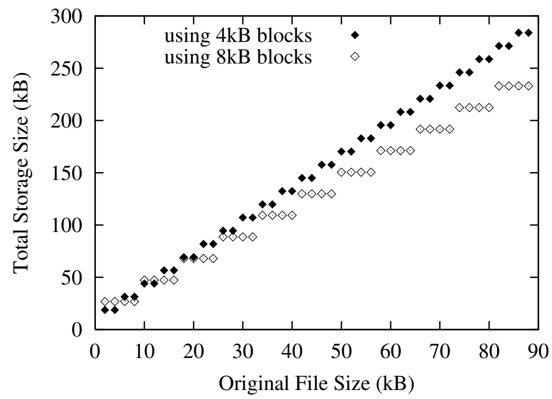



Figure 8: Storage Overhead: Total bytes stored versus original file size

The data block for a file of size $< 4$kB is padded to 4kB and fragmented into 32 encode fragments, each of which is 256 bytes large. The system appends to these fragments 24 bytes of encoding metadata plus 120 bytes of verification information, making the total number of bytes disseminated 12800 bytes. Thus, the smallest size file in Silverback is 19200 bytes. Files of size smaller than 4kB will all be of this size, and files between 4kB and 8kB will be of size 32000 bytes. The sizes continue in a stair-step fashion every 4kB, as shown in Figure 8. This figure also shows dissemination sizes for files when 8kB blocks are used. Note that in this case, smaller files are slightly larger, but that larger files — where most bytes are stored — are steadily smaller than their 4kB block equivalents.

## 5.3 Time to Durability

Users of an archival system want their data to be durably committed as quickly as possible. The time required to produce dissemination fragments is a key component of this figure. Using a stand-alone program, we measured the time it took to encode a 4kB block to be 7.04ms with a standard deviation of 0.23ms.

In our run against Andrew, we measured the time interval between when Silverback began to commit an update to the archival layer to when it put the last fragment on the network for dissemination. This number is a measurement of the load updates produce on a serializer. Figure 9 shows that the time to durability had a minimum size which grew linearly with the number of bytes being disseminated. The variability shown in the graph is a re-

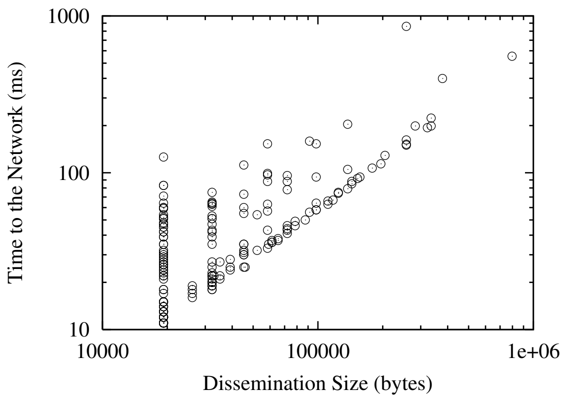



Figure 9: Time to Durability

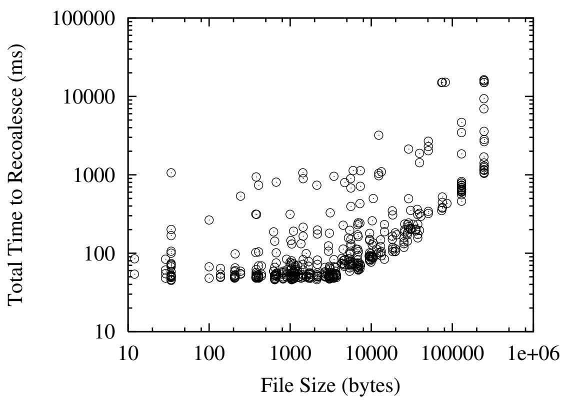



Figure 10: Recoalesce Time

sult of thread context switching and garbage collection in our Java implementation. To help mask these effects, we have removed all times which were more than two standard deviations above the mean time for a particular dissemination size.

## 5.4 Read Performance

To service a read request, an archival system needs to request multiple fragments from the network and perform computation to reconstruct the object. Because these operations impact the user-perceived latency of the read, it is important that an archival system execute these steps efficiently. The local computation required to reconstruct a 4kB block from fragments was measured using a standalone program to be 3.83ms with standard deviation of 0.77ms.

We also measured the time our run against Andrew took to service file requests which resulted in a cache miss. These numbers are shown in Figure 10. This figures shows that for files consisting of just one block, the

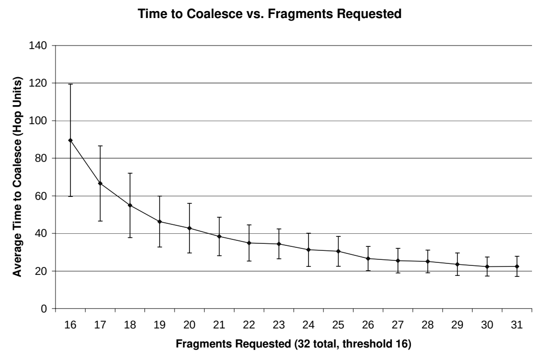



Figure 11: *Reconstruction Latency*: This chart shows latency required to receive enough fragments for reconstruction in a Transit-stub network of 4096 Tapestry nodes.

time from a request to the time that the data has been decode is independent of the file size. For files larger than 4kB, we see that the minimum time required to service a request grows linearly with the size of the file. Because we implemented our prototype in Java, we had to contend with garbage collection. Additionally, we ran our simulation at a time when the resources of the cluster that we used were being taxed by other parties. Both of these factors contributed to a high variability in the recoalesce times, so we removed all data points which were two standard deviations above the mean recoalesce time for other files of the same dissemination size.

## 5.5 Large-scale Simulations

When we receive a read request for a block no longer actively maintained, the archival layer reconstructs it by locating and requesting (via Tapestry) enough fragments of that block, and reconstructing the block from them. To better understand the latencies involved in serving such requests on a wider scale using a routing overlay such as Tapestry, we ran three simulations measuring the tradeoff between number of fragments requested and various performance metrics. These results confirm our hypothesis, that requesting a small number of blocks over the threshold drastically reduces response time in normal and failure-prone environments while incurring a low cost in additional aggregate bandwidth used.

We ran our experiments on a packet level simulator of Tapestry, running 4096 overlay nodes on several topologies, including Transit-stub networks, TIERS networks, and models of the MBone and Autonomous Systems on the Internet. The results are similar across topologies, and we show here only the Transit-stub results. For these

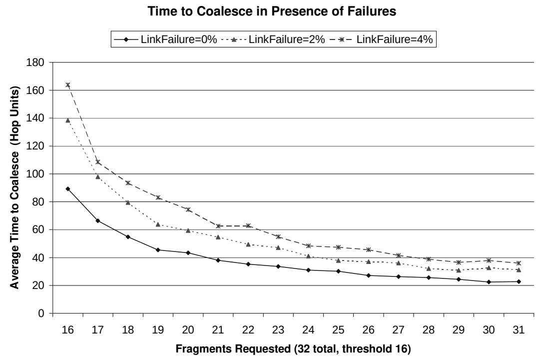



Figure 12: *Reconstruction with Failures*: This chart shows simulated time necessary for block reconstruction in a Transit-stub network of 4096 Tapestry nodes, retrying after link failures.

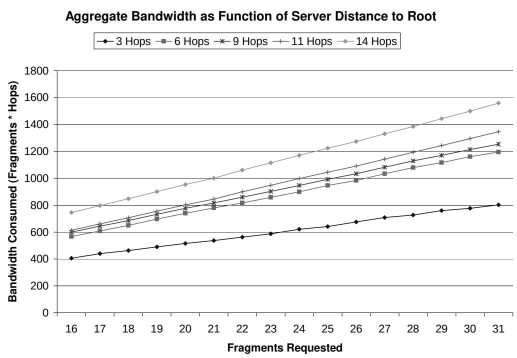



Figure 13: *Aggregate Bandwidth Used*: This chart shows the aggregate bandwidth used during block reconstruction as a function of fragments requested. A bandwidth unit is bandwidth used transmitting 1 fragment across one network hop.

simulations, we assumed a memoryless distribution applied to network hop latency, where the average hop latency was 1 "hop unit." For each run, we simulate the latency required for a randomly placed client to request and receive 16 (threshold) out of 32 (total) fragments for a block. The client needs to only issue a single request specifying the number of fragments ($N$) desired to Tapestry, which travels to the "root node" of the block, gathering the closest $N$ fragments. We also assume that 25% of the 32 randomly placed fragment storage servers are highly loaded, requiring an additional 5 hop units for queuing, where unloaded servers require only 1 hop unit for queue processing.

Figure 11 shows the time before at least threshold (i.e. 16) fragments are received by the client. The result follows a power curve, showing that increasing fragment requests gradually removes factors such as network latency variance and server load. Error bars showing standard deviation also decrease significantly as number of requests increases. Figure 12 confirms this result in the case where links fail, clients detect end-to-end failures and issues retries until success. Finally, Figure 13 shows the expected bandwidth usage, where each unit represents bandwidth required for one fragment over one network hop. We vary locality by measuring from clients at different distances from the object's root node. The result shows clients closer to the root incur a lower slope in aggregate bandwidth, meaning that they find more fragments with less hops, decreasing the penalty for higher fragment requests. Our results demonstrate that by requesting a few fragments over the threshold, we gain significant benefits both in response time and response variability, while incurring a relatively low bandwidth overhead.

# 6 Future Work

There are several unresolved security issues in our archival architecture. Chief among them is that we have not discussed means of preventing a machine from publishing false advertisements To prevent misrepresentation, an introspection layer must use some form of reputation scheme to help filter out malicious machines.

Another (unresolved) issue in Silverback's utility model is the question of billing. Clients will presumably pay a responsible party to ensure the integrity of their data, and the responsible party will in turn cooperate with and pay storage providers. Each archival fragment must therefore be tagged with a billing certificate which can be used on a regular basis to acquire payment, ultimately from the object's owner. Any billing scheme must be thorough enough that double billing (two storage nodes claiming funds for the same fragment) should be impossible.

Finally, the serializer presented in this paper is neither scalable nor fault tolerant. A set of servers on the wide area using a byzantine agreement protocol can be used to provide consistency and conflict resolution for an archive in a fault tolerant manner.

# 7 Related Work

The idea of using versioning as a means to providing time-travel was first introduced with the Postgress database [15]. The Elephant file system [14] studied the idea of time travel in a file system. Additionally, the project examined schemes for reducing the storage overhead by understanding tradeoffs between the number of versions stored and the granularity of time-travel possible.

Several other projects use the idea of distributing data of multiple machines for persistance and availability. The FarSite project [1] replicates data at multiple nodes throughout an organization-scale network. They demonstrate that such a distribution on a typical network can provide five nines availability with a replication factor of only three. The PAST project [3] provides persistant storage in their peer-to-peer system by replicating objects and distributing them throughout the system. Objects stored in PAST are immutable and thus do provide facilities for time-travel. PAST, however, suffers from large storage requirements because and does not provide any mechanisms for repair other than client scans. Most similar to the Silverback archival layer is Intermemory [4]. This system uses Cauchy Reed-Solomon erasure codes to fragment data; the fragments are then distributed among members of the service. In intermemory, repair is driven from a centralized source.

# 8 Conclusion

The most important concern in today's world of ubiquitous computing is that of information persistence. This paper describes Silverback, a global-scale, version-based archival system that is *durable*, *verifiable*, *available*, *maintainable*, and *atomic*, which scales to handle billions of users and a mole of bytes. We discussed the implementation of Silverback and explored the performance of a prototype backup system built on top of it.

Three technologies make this system possible:

- Erasure Coding: Erasure coding provides durability by exploiting the statistical stability of large numbers of independent components.
- Secure Hashing: Secure hashes permit globally-unique IDs to be unforgeably associated with archival data.
- Tapestry Routing and Data Location: Tapestry is a distributed infrastructure that routes queries directly to fragments and resources (such as serializers) using only local information.

Version-based archival systems such as Silverback are enabled by Moore's law growth in disk and storage resources. One of the most exciting consequences of such a system is the legitimate prospect of preserving digital information for 1000s of years.

# References


[1] 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).

[2] CHUN, B. N., AND CULLER., D. E. Rexec: A decentralized, secure remote execution environment for clusters. To appear in 4th Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing.

[3] DRUSCHEL, P., AND ROWSTRON, A. PAST: A persistent and anonymous store. `http://www.research.microsoft.com/~antr/PAST/`, February 2001.

[4] GOLDBERG, A., AND YIANILOS, P. Towards an archival intermemory. In *Proc. of IEEE ADL* (Apr. 1998), pp. 147–156.

[5] HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[6] KLIMOV, V. Network file system interface (NFSI). `http://www.angelfire.com/on/vkjava/`, 1999.

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

[8] LUBY, M., MITZENMACHER, M., SHOKROLLAHI, M., SPIELMAN, D., AND STEMANN, V. Analysis of low density codes and improved designs using irregular graphs. In *Proc. of ACM STOC* (May 1998).

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

[10] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Architecture: A Quantitative Approach*. Forthcoming Edition.

[11] PLANK, J. A tutorial on reed-solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience 27*, 9 (Sept. 1997), 995–1012.

[12] PLAXTON, C., RAJARAMAN, R., AND RICHA, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of ACM SPAA* (June 1997).

[13] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the Sun Network Filesystem. In *Proc. of USENIX Summer Technical Conf.* (June 1985).

[14] SANTRY, D., FEELEY, M., HUTCHINSON, N., VEITCH, A., CARTON, R., AND OFIR, J. Deciding when to forget in the Elephant file system. In *Proc. of ACM SOSP* (Dec. 1999).

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

[16] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems* (Feb. 1996), 108–136.

[17] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Submitted for publication to SIGCOMM, `http://www.cs.berkeley.edu/~ravenben/tapestry.pdf`, 2001.