The OceanStore Archive: Goals, Structures, and Self-Repair

Chris Wells
Computer Science Division
University of California, Berkeley
cwells@cs.berkeley.edu

Abstract

The increasing amounts of digital data, originating from corporations and individuals alike, is driving the need for digital archives forward. This work discusses a system designed to meet this need, the OceanStore archive. This system stores documents in a secure manner, and provides high availability and durability. We describe the structures and algorithms used in the OceanStore archive, and present a distributed fault detection and repair scheme which makes use of OceanStore's routing and location layer, Tapestry.

1 Introduction

There is a growing demand for the automatic, online archiving of digital data. For decades, industry and other users have relied on tape to back up their critical data, but this scheme requires a human administrator to maintain the tape drives, file servers, and the tapes themselves. As the amount of digital data in the world explodes, this maintenance will become too costly to be feasible. Many different projects are currently archiving digital data, including digital libraries [20] and the Internet Archive [1]. At the same time, there are several projects dedicated to distibuting and preserving digital data for long periods of time, among them Intermemory [9], PAST [7], and OceanStore [11].

OceanStore in particular seeks to maintain its users' privacy through end-to-end encryption while simultaneously guaranteeing the integrity of their data. Intermemory and OceanStore both use erasure codes to more efficiently guarantee the long-term durability of their data, as opposed to the more traditional technique of replication like that used in PAST. The practice of distributing one's data across the wide area requires the use of a fault detection and repair scheme. OceanStore must provide such a scheme if it is to meet the challenge of long term survivability of its data. This work discusses the goals and current status of distributed digital archives. It then describes the basic mechanisms and structures used in the OceanStore. Finally, we present a fault detection and repair scheme for the OceanStore which is feasible given today's available hardware.

2 Archive Overview

An *archive* is any repository in which information is preserved. Different types of archives exist today for different types of data. For example, a bank will keep a vigilant record of all of its clients' transactions. For the past few decades, these transactions have been recorded and saved on magnetic tape. More public examples include the Library of Congress, which stores 20 Terabytes of information in the form of text in books, and the Luvre, which stores over 6000 paintings from the 13th to the 19th centuries. A more modern archive is the Internet Archive [1] , which currently contains approximately 43 Terabytes of past and present web pages.

2.1 Archival Goals

The type of data stored in archives varies, and the potential audience for each archive differs (a bank should not reveal the transactions of an individual client to the general public, but the Library of Congress is available to everyone). While the media of each archive differ as well, the underlying principles of each archive are the same. Every archive must address five basic *archival goals*.

Durability: The durability of an archive's data is the most distinguishing feature of an archive. Archives exist to preserve data "for the ages", so a good archive must provide high levels of durability for its data. Non-electronic media such as books and pictures find high durability in libraries and museums where they can be protected from the elements. Indeed, the Luvrue has well-preserved paintings which are centuries old. In more recent decades, corporations have turned from ledgers to magnetic tape for long-term data storage.

These tapes have mean time between failures rated between 200,000 and 300,000 hours — about 30 years [2] — but even these results are the optimistic results of controlled laboratory conditions. Much greater durabilities can be achieved through the use of redundancy on onling, spinning storage (see Section 3.2).

Availability: The value of information is proportional to the data's availability; if a book listed in the Library of Congress is currently on loan, it is not available to the public (and is, for the moment, worthless). Good archives must strive for high availability of their data. If data retrieval is too slow, the data may be effectively unavailable. In this way, availability is related to another archival goal, performance. Many tape archives today use robotic arms to multiplex through hundreds or thousands of tapes, with the entire setup available online. Online availability is highly desirable for digital archives, which may contain enormous amounts of data and may be required to service millions of users.

Security: Security for an archive has two meanings. First, if the archive is a private one, only authorized parties should be able to view it. A simple example of such an archive is a diary. A private electronic archive might involve a person's home vidoes. In a digital archive, end-to-end encryption can be used to preserve the privacy of documents. Security can also refer to the integrity of the documents. The integrity of a document is a similar but stronger requirement than its durability. For instance, if a book in a library were replaced with another book with the same cover, but different text, the librarian in charge of maintaining the book would not notice, but the book's data would have been altered. In a digital, online archive, the integrity of data can be assured by cryptographically tying the data's content to its name. (see Section 3.3).

Usability: If data is not usable, it should not be archived. Or, contrapositively, data should only be archived as long as it is usable. For instance, if no translation existed from ancient Hebrew into modern languages, the Dead Sea scrolls would not have been of such profound interest when they were unearthed (merely archaelogical interest). Similarly, electronic media recorded today may be untranslatable tomorrow — there are few Betamax players still in working condition. A more serious problem facing digital archivists is the lack of tape drives available to read decades-old tape. The tapes may have been perfectly preserved, but because no device exists to read them, they are unusable. These tapes are suffereing from what is known as the *media conversion problem*.

Archived data is increasingly kept online, and in a few years, it may be periodically migrated to newer media automatically, thus avoiding the media conversion problem [9]. A more subtle problem is that of preserving *archival semantics*: a Microsoft Word document may be perfectly preserved, but without Microsoft Word (or another appropriate application), it cannot be read. Solutions to this problem include preserving emulators for every type of digital document.

Finally, if a document's privacy is protected so that only a limited number of people can view it, transfer of viewing rights is necessary to prevent the document from becoming unusable. This transfer is relatively simple in the case of physical archives like books. If a document is electronic, and is cryptographically protected, the transfer is a little more complicated. The document is only usable so long as its read key survives. If an individual wishes to pass on his read key to his inheritors, a secure mechanism must exist for the transfer of the key which will not leak information about the key to other parties. One simple mechanism for an individual who wished to pass on his key would be to encrypt it using his inheritors' keys, and place the results in the care of his estate's executor, but this scheme relies on the trustworthiness of the executor.

Performance: As mentioned earlier, performance is strongly related to availability. The more quickly data can be retrieved from or deposited into an archive, the more valuable the archive becomes. An excellent example of improving the performance of an archive is a library's card catalog. This single index reduces the time to find a book from hours (or perhaps even days) to mere minutes. In the electronic world, indices are ubiquitous. Search and retrieval times are therefore dependent on the media's seek time and bandwidth. For magnetica tapes, the read and write bandwidths are as high as 15 Megabytes per second [6], but the seek time on tapes can be quite high (750 MB/s for a 25 Gigabyte tape, meaning, on average, a seek time of 33 seconds). Ideally, a digital archive should have instantaneous access. More practically, a digital archive's performance should be comparable with that of a local hard disk.

2.2 Hardware Trends

Designing archives for tomorrow requires a careful look at today's hardware trends, and how they will affect tomorrow's capabilities. The trends in modern computing are related to Moore's Law, in that most digital technology is currently improving exponentially. These trends cannot continue forever, but they can persist for the next twenty years. Three general metrics in particular are of interest, and we will see what these metrics look like today and what they will be in twenty years.

Capacity The capacity of storage devices has been growing with Moore's Law. In particular, disk capacity is growing at a phenomenal rate, doubling every 18 months to today's capacyity of roughly 100 Gigabytes. This trend has kept what is essentially one technology (magnetic platters) at the forefront of secondary storage for decades, and it shows no sign of abating. Indeed, data on conventional hard disks can become more dense, and in twenty years the 100-fold increase per decade in density will yield disks capable of storing one petabyte. If this trend continued for fifty years, a single hard disk could store one Zettabyte, or one billion trillion bytes. This number comes perilously close to violating the maximum density available were we able to control individual atoms, so the exponential increase in disk capacity will likely end before then. Still, even a petabyte of storage is immense.

While the capacity of tapes has been keeping pace with that of disks, the cost-capacity of tapes has not. In the past, the cost of a byte of disk was roughly ten times that of a byte on tape. Today, however, this ratio has been reduced from ten to three [10]. In terms of capacity, then, tapes are becomes less attractive relative to disks.

Bandwidth The transfer rate of available hardware is of critical importance when designing an archival system. As previously mentioned, today's tape systems can transfer 15 Mb/s. Disks have a roughly equivalent transfer rate (25 Mb/s). Like their capacity, hard disk's transfer rate is growing exponentially. The rate of growth, however, is much smaller than the rate of growth for capacity — doubling once every three years. The capacity / bandwidth ratio of hard disks increases ten times per decade [10].

An interesting observation made in [3] is that network bandwidth is beginning to outstrip disk bandwidth. In fact, Gigabit ethernet is on the horizon. Moore's Law for network bandwidth states that link bandwidth will double every 18 months. The deployment of links, however, is outpacing this prediction. G. Gilder predicted in 1995 that deployed network bandwidth will triple every year for the next twenty-five years [8] . According to [10], today's fiber optics have a bandwidth of 40 Gbps. In another 20 years, these single links could be as large as 400 Tbps. By the same factor, today's 100Mbps ethernet could be replaced in 20 years with Tbps lines. By comparison, disk bandwidth will be only 2.5 Gbps. It would seem, then, that an ideal archive would stripe data over a large number of disks to reduce the cost of reads, allowing the archive's bandwidth to approach network bandwidth.

	today	20 years
CPU	1 GHz	10 THz
disk capacity	100 GB	1 PB
disk bandwidth	25 Mbps	2.5 Gbps
network bandwidth	100 Mbps	1 Tbps

Figure 1: Summary of hardware capabilities today and 20 years in the future.

Finally, some forms of archival encoding may require computations which exceed those required for I/O operations. CPU speeds may therefore be critical in terms of an archive's effective bandwidth. Moore's Law for CPU speeds states that CPU speeds double every 18 months. Today's CPUs are running at 1 GHz, so in twenty years, we could see a processor capable of 10 THz, or 10,000 times faster than today's processors.

Latency Latencies are improving at a much more modest pace than are bandwidths. Indeed, network latencies are already significant fractions of the speed of light. A single message crossing the United States today takes 30ms, and barring a revolution in physics, it will take 30ms for the rest of humankind's existence. Disk seek and rotation times will fair slightly better, improving at a rate of approximately 8Today, one half a rotation takes roughly 3ms, and seek times are about 6ms. This gives us a latency of 9ms for a random seek. In twenty years, this number will be reduced to 1.7ms. Current trends in software to offset the disparity between seek times of disks and disk bandwidth and capacity include queueing (so that accesses are prioritized based on their physical locations on the disk rather than the order in which they arrived) and caching more data in an ever-growing main memory.

Summary The problems with disk seek time and bandwidth can be mitigated by observing that more disks can always be added to the network. A single machine's bandwidth to the Internet will likely be that of a single link. Further, the CPU power available to a single machine will be within a small constant factor of the CPU power of a uniprocessor. These two trends, then — network bandwidth and CPU power — will determine the available bandwidth to any distributed archive. The latency of a wide-area archive will, without caching, necessarily be on the order of tens of milliseconds due to the speed of light. The capabilities of hardware we expect now and in twenty years are summarized in Figure 2.2

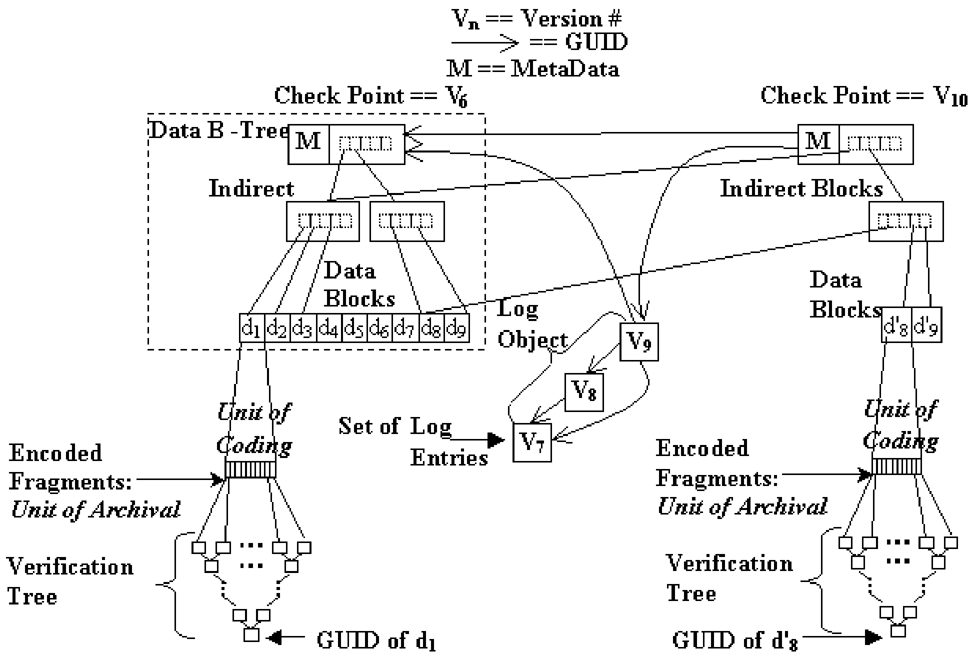

Figure 2: The data object structure.

3 Archive Data Structures

An *archive* is a linear sequence of *versions*, which in turn are linear sequences of bytes. For example, a file may be considered an archive, and its contents at a particular day and time are one version of that file. A distributed archive such as that in OceanStore requires that every object stored be located by a globally unique identifier (GUID). It also requires that there exist two basic types of GUIDs: a version GUID (V-GUID) names a specific version of an archive; an archive GUID (A-GUID) is the name of an archive, and it can be used to request from the archival system the most recent V-GUID of that archive.

This section describes the data structures used in OceanStore to store and represent the data contained in the archival system. We first describe the basic data object structure, and then move on to describe how we supplement that structure to provide strong durability and integrity guarantees as well as tight binding to GUIDs.

3.1 Version Representation

An archival system — especially a distributed archive — may contain versions so large that a typical client cannot cache them in their entirety. An archival system may also possess archives which have small updates applied to them regularly. To address both of these features, OceanStore uses the Data Object structure shown in the dashed box in Figure 2.

A version of an archive is an array of bytes. As in many filesystems and virtual memory systems, OceanStore breaks a version's array of bytes up into uniform-length *blocks*. The blocks form the leaves of a B-tree, which is shown in the dashed box. A block's parent in the tree must contain location and verification information for that block. In OceanStore, the parent block stores its children block's GUIDs, which are sufficient for both location and verification. In this way, a client can request and verify a block as long as it has the block's parent. Note that a client can cache as few or as many blocks as needed; the entire B-tree does not have to be stored as a single unit.

As mentioned in Section 3.3, the GUID of a block is the root hash in a hierarchical verification tree over the block's data and its fragments. The GUID of the top block is calculated in exactly the same manner as the GUIDs of other blocks, but it is used as the V-GUID for the data. A user of OceanStore can request any piece of data in the B-tree given its V-GUID and an offset in the data. The top block is also unique in that it has appended to it *metadata* for the archive and version, and these two entities — B-tree information and object metadata — are encoded and disseminated as a single block of information. The metadata can contain information such as owner identification, access control, and the V-GUIDs of past versions. It can also be used by application writers to solve the archival semantics problem by including in the metadata the GUID of an appropriate emulator for that object.

A client modifying a data object can use copy-on-write to prevent having to rearchive all of the old version's data. A client can change a single data block in the version, which will result in the block hashing to a new block GUID. This new GUID must be stored in the block's parent, so the parent block will change as well, producing a new GUID. The change is propagated up the B-tree, so that changing a single data block will result in a number of blocks equal to the height of the tree to be changed, and thus require redissemination. To alleviate the storage overhead inherent in this scheme, OceanStore also makes use of logging between versions.

3.2 A Case for Erasure Codes

Section 3.1 reduced our storage interface to that of a block store. To store versions of archives in the OceanStore, we therefore need only store in a distributed and durable manner the blocks of those versions. The most common methods used to achieve high durability of data are complete replication and parity schemes such as RAID [15]. 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 the high rate of failures expected in the wide area. Erasure codes are a superset of these classic mechanisms which provide extremely high durability and availability without imposing an unreasonable overhead in storage space.

Using erasure codes, a user can break up a block 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 block can be reconstructed from *any* n fragments. Figure 2 illustrates the fragments of block d1.

There are a number of erasure codes with different performance characteristics. Some, such as Tornado Codes [13], 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" [17] 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 OceanStore, we chose an effiecient version of Reed Solomon called Cauchy Reed Solomon codes.

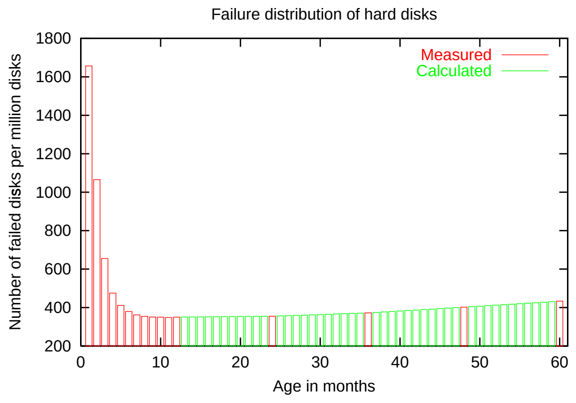

Figure 3: Disk failure distribution

3.2.1 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, the 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 follows:

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 a block 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 block over five nines of availability(0.999994), yet consumes the same amount of storage. With 32 fragments, the availability increases by another factor

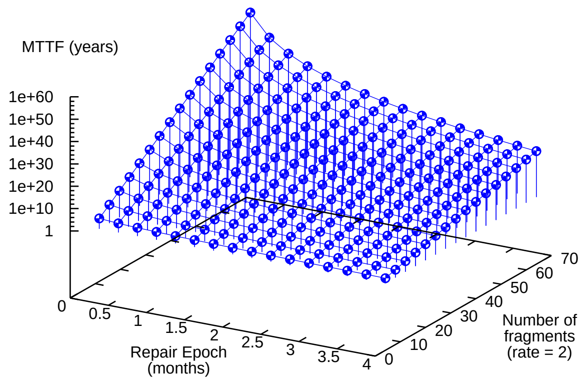

Figure 4: Mean Time to Failure of a Block

of 4000, supporting the assertion that *fragmentation increases availability*. This is a consequence of the law of large numbers.

3.2.2 Durability

An analysis of the MTTF of fragments and fragmented blocks is also essential in motivating the use of erasure codes. Disk failure distributions obtained from [16] and shown in Figure 3 indicate that while disks have some infant mortality, a high number of them survive the 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 block. 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 4 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.

3.2.3 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* (6×10^{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 3.2.2, 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.

3.3 Naming and Integrity

Now that we have a distributed, durable block store, we need a mechanism by which to name and locate individual archives, versions, and blocks. As stated in Section 3.1, a version's V-GUID is merely the block GUID of its B-tree's topmost block. Thus, only two basic types of GUIDs need be produced. This Section describes how block GUIDs and archive GUIDs are created and verified in the archival system.

3.4 Naming and Verifying Blocks

Erasure coding requires the precise identification of failed or maliciously 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.

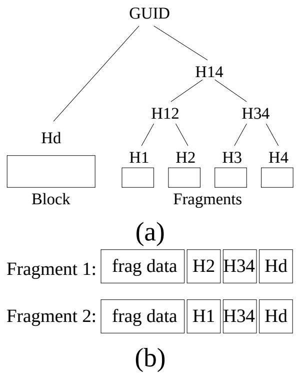
GUID
H14
H12
H34
Hd
H1
H2
H3
H4
Block
Fragments
(a)
Fragment 1: frag data H2 H34 Hd
Fragment 2: frag data H1 H34 Hd
(b)

Figure 5: The block *GUID* is the root hash in a binary verification tree of hashes over the fragments and data of a block

For each encoded block, we create a verification tree over its fragments. Figure 5(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, and 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 5(b) shows the contents of a “dissemination fragment”. The hash at the root of the tree is the GUID of the block. To ensure that other data does not hash to the same GUID, we use the SHA-1 [14] 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.

A GUID should not only verify a fragment. A block should be verifiable against its GUID independent of its fragments. A simple extension to the above scheme suffices. Once the root hash for the fragments of a block has been calculated, it can be concatenated with a hash of the block’s unencoded data, and the hash of this concatenation will then be the block’s and fragments’ GUID. Each fragment will store one additional hash (the hash over the block’s data), but the block — coupled with the root fragment hash — will be verifiable against the block GUID.

3.5 Naming and Verifying Archives

While each version of an archive possesses a GUID, the entire archive must also have a GUID (through which its

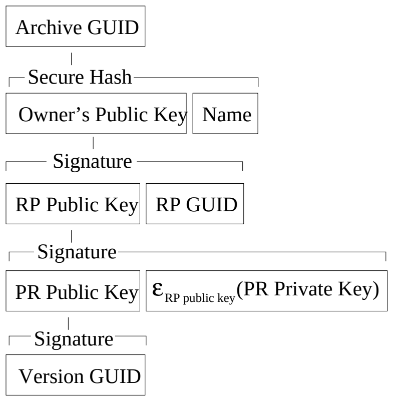

Figure 6: A *tombstone* is a secure mapping from an *A-GUID* to a *V-GUID*. The V-GUID is signed by the primary ring’s key, which in turn is signed by the responsible party’s key. The responsible party’s key is signed by the owner’s key, which is verifiable by a secure hash that produces the A-GUID.

most recent version or past versions can be requested). There must exist a secure mapping in the archival system from the archive’s A-GUID to the V-GUID of its most recent version. This mapping can exist in one of two forms in OceanStore. The archive may have an active primary ring, which is a set of servers using Byzantine Agreement protocols to maintain the A-GUID to V-GUID mapping [1]. If no primary ring exists, the mapping is stored in *tombstones*, so named because the primary ring puts them in place in the event of its death.

A tombstone for a particular archive is named and located by that archive’s A-GUID, and it contains the public key of the archive’s owner, the GUID and public key of the archive’s responsible party, the public key of the archive’s last primary ring, the human-readable name of the archive, and the latest V-GUID of the object. These items are arranged as shown in Figure 3.5, so that each of them is verifiable against the A-GUID. Thus, a tombstone is completely verifiable by its archive’s GUID: one need simply hash over the concatenation of the owner’s public key and the archive’s human-readable name to verify the owner’s public key against the A-GUID, use the owner’s public key to verify the responsible party’s public key, use the responsible party’s public key to verify the primary ring’s public key, and then use this public key to verify the tombstone’s signature of the V-GUID. When a primary ring 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 coun-

[1] Primary or Byzantine Rings are discussed in more detail in [11]

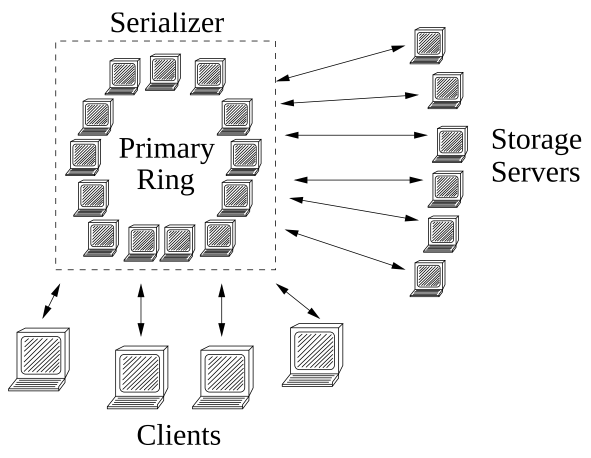

Figure 7: The OceanStore, as an archival system, is composed of clients, serializers, and storage servers. Writing clients must communicate with the serializer, but reading clients can communicate either with the serializer or with the storage servers directly. In OceanStore, the primary ring serves as the serializer.

terparts. In this way, the A-GUID to V-GUID mapping is made durable through replication. By replicating tombstones, we also enable them to be repaired in the same way as fragments (discussed later).

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

4 Interfaces

The archival layer of OceanStore has been described as a block storage system. With the addition of tombstones, it also must store mappings from A-GUIDs to V-GUIDs. This second type of storage is more complicated than merely storing read-only blocks, because it requires the archival system to synchronize and serialize versions of an archive so that the entire system considers the same distinct V-GUID as the most recent V-GUID for a given A-GUID. The archival system therefore requires a *serializer* for each archive which accepts updates from authorized clients, applies them in a sequential order, and recalls the V-GUID of the most recently created version. Figure 4 depicts the relationships among OceanStore client machines, the serializer, and OceanStore storage servers.

This section describes the interface which the archival layer must implement. This interface is used both by clients communicating with the primary ring and the storage servers and by the primary ring itself. We also describe the interface which each storage server must implement.

4.1 Archival Layer Interface

The archival layer of OceanStore is in charge of encoding and disseminating individual blocks of objects, and storing the mappings from object GUIDs to most recent version GUIDs. The interface for these mechanisms is straightforward:

```
disseminate(block) ⇒ GUID
```

The disseminate operation takes a block, erasure encodes it, disseminates the resulting fragments, and returns the GUID generated for the block from the blocks data and fragments. This routine is called by the software layer immediately above the archival layer, which handles placing the GUIDs of new children in their parent blocks, and then calling `disseminate()` for those parrents.

```
retrieve(block GUID) ⇒ block, frag-
ment hash
```

When a client desires a block which is stored in the archive, it calls the `retrieve()` routine, which locates the block's fragments with its GUID, retrieves them, and reconstructs the block. It returns the block's data as well as the top-most hash in the fragments' verification tree (recall that this hash, concatenated with the hash of the block's data, will hash to the block's GUID).

```
disseminateTombstone(object GUID, ver-
sion GUID, owner key, primrary ring
key certificate, primary ring key)
```

When a primary ring (or any other authorized client) wishes to generate a new object GUID - version GUID mapping, it calls this routine to create the tombstones and disseminate them, replacing any tombstones which currently exist for the object.

```
retrieveTombstone(object GUID) ⇒ ver-
sion GUID
```

This routine will retrieve an object's tombstones and will return the version GUID they indicate as the most recent version GUID.

4.2 Storage Server Interface

The items which the archive must disseminate, store, and retrieve are thus erasure-encoded fragments and tombstones. The two important qualities shared by these two types of data is that both are self-verifiable, and both are durable against failure by making use of redundancy. Each can be represented as a sequence of bytes, which in turn can be stored by any modern computer. For the remainder of this section, “fragment” will refer to any item which can be stored by the archival storage servers. The interface of a storage server in the OceanStore archive is as straightforward as the interface for the archive:

```
store(GUID, fragment, certificate) ⇒
success or failure
```

When a server is asked to store a fragment, it places the fragment on permanent storage so that it can be retrieved using its GUID. If the fragment is already on the server, `store` returns failure. The certificate contains information pertaining to the owner’s identity and billing information. It should securely identify the party requesting storage for the fragment.

```
contains(GUID) ⇒ success or failure
```

A server can query whether or not it has a fragment.

```
retrieve(GUID) ⇒ fragment or failure
```

A server indexes its storage medium using the input GUID, and it returns the fragment named by that GUID, or failure if the fragment could not be found.

```
delete(GUID, certificate) ⇒ success
or failure
```

A server may also be requested to remove a fragment. The fragment is located by its GUID, and the server returns either success or failure. Note that the requesting party must include a certificate enabling him to perform the deletion. Such a certificate could simply be a nonce signed by the owner or responsible party key.

With this simple interface, many possible implementations can exist. A naive implementation stores each fragment in a separate file, and names the files according to the fragments’ GUIDs. All operations of the storage server are thus reduced to file system operations. This implementation ignores a critical feature of the data in the OceanStore archive: fragments and tombstones are *small*. For example, a 4K block erasure-encoded into 32 fragments with a rate 1/2 code will produce fragments of size 404 bytes. This number includes the fragment data, the fragment’s verification information, and additional overhead such as the fragment’s GUID and its type and rate of encoding. The total size of a fragment is calculated as follows:

$$\begin{aligned}
\text{fragment data size} &= \frac{4096}{16}\,\frac{\text{bytes}}{\text{message fragment}} \\
&= 256\,\frac{\text{bytes}}{\text{message fragment}} \quad (2)\\
\text{hash overhead} &= 20\,\frac{\text{bytes}}{\text{hash}} \times 5 = \\
&\quad 100\,\frac{\text{bytes}}{\text{fragment}} \quad (3)\\
\text{additional overhead} &= 28\text{bytes} \quad (4)\\
\text{total fragment size} &= 256 + 100 + 28 = 400\text{bytes} \quad (5)
\end{aligned}$$

Tombstone are even smaller at approximately 200 bytes a piece. These small sizes are not well supported in our naive scheme, because each fragment will take up an entire block on disk (4K). A more intelligent scheme assigns multiple fragments to a single file, and maintains a mapping from GUIDs to storage files. Other implementations are possible, including databases. Because dissemination of fragments will be random, there should be no correlation between fragments on a single node. Additionally, the fragments for a block should not be accessed often, since we expect OceanStore to intelligently cache active documents. These facts seem to indicate that there is no good way avoid going to disk on every fragment retrieval. Recall from Section 2.2 that disk bandwidth today is approximately 25 Mbps, and in twenty years will be 2.5 Gbps. The number of fragments stored on a device will therefore be determined not by the device’s capacity, but by the bandwidth demanded of the archive by its users.

5 Tapestry

Fragments stored in the OceanStore archival layer are allowed to reside on any server. This property gives the archival system the ability to spread fragments for an object across the world (increasing the object’s durability), and it makes the task of removing old servers from and adding new servers to the OceanStore much simpler. To route to fragments with this property requires a sophisticated routing and location layer. OceanStore therefore makes use of Tapestry [22], an overlay network based on the hashed-suffix routing structure by Plaxton, et. al. [18].

Recall that all fragments for a given block are named by the same GUID. Each GUID maps deterministically to a single node in Tapestry whose node identifier most closely matches the GUID. This node is referred to as the

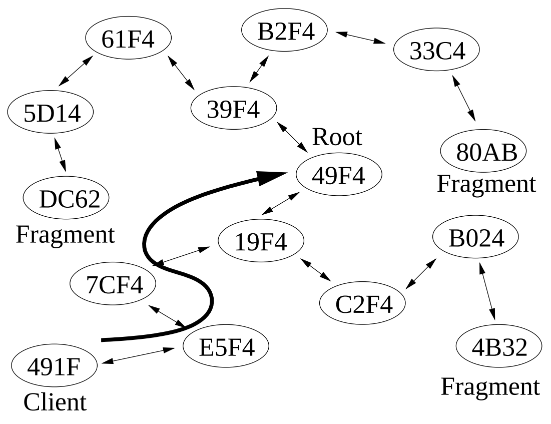

Figure 8: A single location tree in Tapestry. The three fragments publish themselves to their block's root by sending messages to nodes whose id suffixes are increasingly similar to that of the block GUID. A client requesting the fragments routes its request to the root, as shown by the thick line initiating at the Client.

object's *root* node. When a fragment is stored on a storage server, it advertises itself through Tapestry by routing a message to this root node. At each hop in Tapestry, this message deposits a pointer containing the fragment's GUID and the GUID of the node storing it. As advertisement messages for different fragments converge to the root, intermediate nodes contain increasing numbers of pointers for the object's GUID. Figure 5 illustrates three fragments for a block located in a Tapestry tree, and a client sending a request for the three fragments. The root, on receiving the client's request, will forward the request to the three nodes containing the fragments (which are themselves the roots of their own GUIDs' trees). Each of these storage nodes will then send their fragment to the client, which is also the root of its own GUID's tree.

Figure 5 summarizes the number of nodes per hop and number of fragments per node for Tapestry given a network with 10^9 nodes, a branching factor of 16, and a single object with 32 fragments. As the table indicates, there will be only 8 hops between a fragment and its root, but to retrieve 16 fragments, a request must traverse all the way to an object's root. Because the root is so critical to location in the archive, multiple location trees are used for each object. Additional details of multiple locations trees and routing in Tapestry can be found in [22].

6 Fault Detection and Repair

This section will survey the different types of repair available to the OceanStore system. These types of repair

hop	pointers per node	nodes per hop
1	1	6.25e+7
2	1	3.91e+6
3	1	244000
4	1	15300
5	1.013	954
6	1.282	59.6
7	10.67	3.73
8	32	0.233

Figure 9: The number of expected nodes at each hop in a Tapestry location tree, and the number of expected pointers per node at each hop, given a network of 10^9 nodes, a branching factor of 16 in Tapestry, and an object with 32 fragments.

include the local repair of fragments by their own storage servers, Tapestry's detection of server failure, and the distributed fault detection and repair of individual fragments. We close the section with an analysis of simulations run against our distributed fragment fault detection and repair scheme.

6.1 Local Repair

One problem in secondary and tertiary storage studied today is that of media failure. Tape can rot, and blocks can go bad on disks. Error correcting codes and other forms of redundancy are often used to prevent these kinds of failures from damaging the bits that are being stored. OceanStore archival storage servers have available to them redundancy so long as they are connected to the wide area. A server can slowly sweep through the data it stores, using its fragments' cryptographically self-verifying nature as checksums to ensure that the data has been preserved correctly. If an error is detected, the server — as a client of the OceanStore — need only request the failed fragment's block from the archive, reconstruct the block, and then fragment it to recreate the lost fragment. We do not expect storage media to fail frequently, so this sweep can be done over a fairly long period of time. In twenty years, according to Figure 2.2, a disk will store one Petabyte of data and have a bandwidth of 2.5Gbps. Using all of the disk's bandwidth, a sweep can be completed in no fewer than 37 *days*. This number is unacceptable. As mentioned in Section 2.2, the amount of data on a disk will be determined by the disk's bandwidth, and not its capacity. By replacing a 1 PB disk with 100 10 TB disks, the local sweep period of 37 days can be maintained, but will use only one percent of the disks' bandwidth.

component	server heartbeat	fragment heartbeat	MAC key update
fragment GUID	0	20	0
server GUID	20	20	20
timestamp	8	8	8
signature	20	20	20
public key	0	0	128
Total	48	68	176

Figure 10: Breakdown of Tapestry pointer heartbeat contents and MAC updates. All sizes are expressed in bytes.

6.2 Tapestry Pointer Repair

Before we discuss the repair of fragments, we must address the self-repair of the Tapestry. The pointers in Tapestry must be kept up to date, and must be kept consistent (though brief periods of incosistency are tolerable due to Tapestry's fault-tolerant nature). For the analysis of this section, we will assume a network with 6×10^9 users, one machine per user, and 10 GB of data per user. Data is broken up into 4KB blocks, each of which are encoded into 32 fragments.

6.2.1 Default Tapestry Pointer Update

The mechanism for Tapestry self-repair endorsed in [22] is a fragment heartbeat. We extend that scheme in this work to prevent malicious users from issuing heartbeats for servers that they do not own. Security can be attained by including with each heartbeat a MAC of its contents and the MAC key. This scheme requires that each server store a MAC key for each other server with whom it communicates. The size of a MAC key is 20 bytes, and the number of MAC keys which must be stored on each Tapestry node is at most the number of pointers per Tapestry node; this implies a storage overhead of less than 50%. If a machine receives a heartbeat for which it has no MAC, it can easily request the MAC key from the sender. The MAC update message from the sender includes the senders's public key, which can be verified because a Tapestry node's GUID is a hash over its public key. We assume that requests for MAC keys occur very infrequently relative to the number of messages sent.

The contents of a fragment heartbeat are enumerated in Figure 10. Once a day, an object residing on a Tapestry node will issue a heartbeat up each of its location trees. This heartbeat includes the GUID of the object as well as the GUID of the server. In this way, the location information in Tapestry is preserved through soft-state. For the OceanStore archive, however, this scheme is infeasible. Consider the total number of fragments in our example:

$$\begin{aligned} 6 \times 10^9 \text{people} \times 10 \times 1024^3 \frac{\text{bytes}}{\text{person}} \\ \times \frac{1}{4096} \frac{\text{block}}{\text{bytes}} \times 32 \frac{\text{fragments}}{\text{block}} \\ = 5.033 \times 10^{17} \text{fragments} \end{aligned} \quad (6)$$

With one node per person in the world, there will be

$$\frac{5.033 \times 10^{17}}{6 \times 10^9} = 8.39 \times 10^7 \frac{\text{fragments}}{\text{node}} \quad (7)$$

Now consider that each node will also serve as a tapestry root for fragments. By symmetry, this means that each node will store, as a root node, 83.9 million pointers. But Tapestry uses redundant location trees (let us assume five), so there are actually five roots for each fragment. Our network size will produce Tapestry location trees of height 8, and each node will serve at different heights in different location trees — a particular node will appear as often two hops away from fragments as it will eight hops away. So we end up with

$$\begin{aligned} 8.39 \times 10^7 \frac{\text{fragments}}{\text{node}} \times 5 \frac{\text{trees}}{\text{fragment node}} \\ \times 8 \frac{\text{pointer nodes}}{\text{tree}} = 3.36 \times 10^9 \frac{\text{pointers}}{\text{node}} \end{aligned} \quad (8)$$

Each pointer is 40 bytes large (20 bytes of SHA1 hash for the object GUID and 20 bytes for the storage node GUID). Thus, each server stores — and must have periodically updated — 134 GB in pointers. With MAC keys, this number increases to 201 GB. [2] In the default Tapestry scheme, each of the pointers residing on a machine will be refreshed daily. This will result in

$$8.39 \times 10^7 \times \frac{1\text{day}}{86,400\text{seconds}} = 971 \frac{\text{fragments}}{\text{second}} \quad (9)$$

and

$$3.36 \times 10^9 \times \frac{1\text{day}}{86,400\text{seconds}} = 38,900 \frac{\text{pointers}}{\text{second}} \quad (10)$$

being processed by each node. The bandwidth required by each node is therefore

[2]This number is actually larger than the $\frac{5.033 \times 10^{17}}{6 \times 10^9} \times 400 = 3.36 \times 10^{10}$ bytes stored on each node for fragments. This particular result is expected, since to provide high durability and availability of distributed information, we must necessarily use redundancy of information.

$$38,900 \frac{\text{heartbeats}}{\text{second}} \times 544 \frac{\text{bits}}{\text{heartbeat}} = 21.2\text{Mbps} \quad (11)$$

of bandwidth. If we assume 100Mbps links to each node in the system (optimistic, but not unreasonable, for today), then this is twenty percent of the available network bandwidth. The required CPU time is determined by the speed of the MACs. Each MAC can be prodcued and verified by hashing over the message contents, and then hashing over the concatenation of that hash with the MAC key. Thus, two hashes are used. The speed of the SHA-1 hash was measured in [19] as .016 ms on a 266 MHz machine. If we extraploate to a 1 GHz speed, this number becomes .00426 ms. Thus, a MAC takes .00852 ms to produce and to verify. Using this number, the required CPU time in this scheme for cryptography is

$$\begin{aligned}(3.89 \times 10^4 \frac{\text{MAC}}{\text{node second}} + 971 \frac{\text{MAC}}{\text{second}}) \\ \times .00852 \frac{\text{ms}}{\text{MAC}} \\ = 340 \frac{\text{ms}}{\text{second}} \quad (12)\end{aligned}$$

This number is over 30% of the available CPU time. We can develop a more efficient update scheme.

6.2.2 Server HeartBeats

We can significantly improve upon the default scheme by only updating server routing tables. In the Tapestry, each node has a neighbor table of size $b \times \log_b N$. With $b = 16$ and $N = 2^{160}$, this is 640 neighbors per node. If a server periodically republishes itself to each of its neighboars once an hour, the total bandwidth required will be

$$\begin{aligned}640 \frac{\text{heartbeats}}{\text{node minute}} \times 384 \frac{\text{bits}}{\text{heartbeat}} \times \frac{1\text{minute}}{60\text{seconds}} \\ = 4.096\text{Kbps} \quad (13)\end{aligned}$$

and the total required CPU time will be

$$\begin{aligned}(.00852 \frac{\text{ms}}{\text{MAC minute}} + \\ 640 \frac{\text{nodes}}{\text{node}} \times .00852 \frac{\text{ms}}{\text{MAC minute}}) \\ \times \frac{1 minute}{60\text{seconds}} \\ = .0910 \frac{\text{ms}}{\text{second}} \quad (14)\end{aligned}$$

For secure server heartbeats occuring once per minute, then, our total resource usage for network bandwidth is $4.096\text{Kbps}/100\text{Mbps} = .004\%$ and our total resource use of CPU time is .009%.

6.2.3 Notification HeartBeats

When a given node goes down, there are 640 nodes which will quickly notice. Other nodes which are pointing to critical data on that node can leverage these other nodes for *notification*. Critical data may include active objects (primary rings or cached copies), or fragmented objects for which there are only a few fragments left (so the objects' roots want to know as quickly as possible when another fragment is lost). A simulation similar to that discussed in Section **??** revealed that using a repair scheme on 32 one-half rate fragments with heartbeats occurring once a month, and with repair occuring either immediately after the loss of 8 fragments, or one week after the loss of 4 fragments, that the total number of lost fragments was only less than 8 less than 1% of the time. Thus, the proportion of objects in the archive which are critical (under this scheme) is 1 in 100.

We therefore propose that, for critical objects, root nodes in Tapestry request notification. If Server A wishes to be notified when an object on Server B is lost due to server failure, it can register with a number of B's neighbors (let's say 5). This is done deterministically in the following manner:

1. A sends a notification request to Server B. It can repeat this request several times, and if B does not send a response, A knows that B is already down.

2. B uses A's server GUID as a seed to a random number generator which produces five random numbers between 1 and 640. The first random number determines to which neighbor out of B's 640 neighbors the notification request message is sent.

3. B routes A's notification request to each selected neighbor.

4. When the notification request is received by Server C, it stores A's GUID along with B's GUID as an *interested party*

If Server B fails, Server C will detect this failure because it will not have received B's server heartbeats. It will then send a message to Server A informing it of B's failure.

This scheme issues a heartbeat containing two GUIDs (one for the server to be watched, and one for the server to be notified). These are the same size as the heartbeats in Section 6.2.1. Only one out of every forty fragments will require one of these heartbeats (which are actually issued from the Tapestry roots), but five of these heartbeats will occur per fragment. Heartbeats travel from the server wanting notification to the server to be watched, a trip which on average should take 4 hops (half the distance up the location tree until a pointer to the watched node is found). One additional hop is then required to route the notification message to a neighbor node.

If we let notification heartbeats occur once per day, the bandwidth required is

$$\begin{aligned}\frac{1}{100} \times 8.39 \times 10^7 \frac{\text{fragments}}{\text{node day}} \times \frac{\text{5 root nodes}}{\text{fragment}} \\ \times (4\frac{\text{hops}}{\text{root node}} + 5\frac{\text{notifiers}}{\text{root node}} \times \frac{\text{1hop}}{\text{notifier}}) \\ \times 544\frac{\text{bits}}{\text{hop}} \times \frac{\text{1day}}{86,400\text{seconds}} = \\ = 238\text{Kbps per node} \qquad (15)\end{aligned}$$

The CPU is used to MAC a notification heartbeat, and to verify that heartbeat five times. The number of notification heartbeats which must be sent out by an average node per second is then

$$\begin{aligned}\frac{1}{100} \times 8.39 \times 10^7 \frac{\text{fragments}}{\text{node day}} \times \\ (1\frac{\text{MAC}}{fragment} + 5\frac{\text{notifiers}}{\text{node}} \times 1\frac{\text{MAC}}{\text{notifier}}) \\ \times .00852\frac{\text{ms}}{\text{MAC}} \times \frac{\text{1day}}{86,400\text{seconds}} \\ = .496\frac{\text{ms}}{\text{second}} \qquad (16)\end{aligned}$$

The resource use of this scheme for network bandwidth and for CPU is therefore .24% and .050%, respectively.

6.3 Distributed Repair for Fragments

We have seen that servers are capable of repairing their own fragments, and we have seen that, for critical documents, Tapestry can use a combination of server heartbeats and notification to quickly detect additional failures. Each of these schemes, while beneficial, is insufficient toward maintaining a distributed object for a long period of time. The OceanStore archive requires a *distributed fragment repair scheme*.

There are four basic types of distributed repair schemes for fragments:

Untrusted passive detection A fragment periodically sends a heartbeat (see Section 6.2 for details) to an untrusted party or parties in the network. If one of these parties does not receive a heartbeat after a number of heartbeat periods, it marks the fragment as inactive.

Untrusted active detection An untrusted party in the infrastructure can also periodically sweep fragment(s) by requesting them from their storage server(s). When it receives a fragment, it checks its integrity. If a server fails to send a correct fragment after a number of sweep periods, the untrusted party marks the fragment as inactive.

Trusted passive detection Fragments periodically send heartbeats to a party in the network whom object owners entrust with the integrity (though not the contents) of their data. Again, after a number of missed heartbeats, this party marks the fragment as inactive.

Trusted active detection The trusted party periodically requests all of the fragments and verifies them. If a fragment fails to appear after a number of sweeps, the party marks it as inactive.

In this categorization, *trust* refers to whether or not the party in question will behave correctly 100% of the time. Untrusted parties may undergo Byzantine failures. *Passive* refers to fault detecting algorithms in which the detecting party notices a fault only when a fragment stops advertising itself. Finally, *active* refers to the detecting party verifying that a storage server not only claims to have a fragment, but that it can actually produce the fragment correctly upon request. Notice that each form of detection relies on a number of failures to occur before a fault is declared. This number is tunable, and in the case where it is set to one, the detecting party can maintain no fault detection state between epochs.

OceanStore has the notion of a *responsible party*, an entity that is trusted with the integrity of its clients' data. This party can perform passive detection and active sweeps of its clients' fragments. Many responsible parties may exist in the OceanStore, though there will certainly be far less of them than users or storage servers. Because it is centralized, we wish to limit the amount of work performed by and state stored at the responsible party. While the design and nature of the responsible party are beyond the scope of this paper, we discuss a possible solution to its scalability problem in Section 8.

OceanStore also assumes a network that is composed almost entirely of untrusted, online servers that can perform fault detection. In particular, the Tapestry nodes in the location trees for a fragmented object already have pointers to the object's fragments, so they are ideally suited for the task of fault detection. Tapestry offers another benefit for fault detection, in that it provides for locality of detection; nodes which are low in an object's location tree are very close to its fragments relative to nodes which are high in the tree. These servers can therefore perform fault detection more often than servers which are farther up the tree. We still want nodes very high in the tree to do fault detection occassionally, because in this manner we can insulate the system against regional outages. A node in the Tapestry performs heartbeat checks and sweeps in periods which grow exponentially with the node's distance from the fragment(s) which it is testing. Using an exponentiation factor of 1.5, if a node one hop away expects a heartbeat once a month, a node two hops away would expect a heartbeat every 1.5 months, a node three hops away would expect a heartbeat every 2.25 months, and the root (8 hops away) would expect a heartbeat every 17.09 months. Of course, nodes low in the location trees progagate detected failures up the trees to the roots, and reported faults are easy to verify — a node can request a fragment which has been declared faulty [3]. In this manner, Tapestry roots become clearing houses for fault information. Because there are five roots, this information — while not perfectly consistent among the roots — is highly reliable by means of replication.

Finally, the question of repair must be answered: when too many faults have been detected, who recreates and redisseminates the fragments? In the face of billing (discussed in Section 8), only the responsible party can recreate fragments and redisseminate them in a secure manner.

Without billing, anyone can successfully recreate fragments. In particular, the Tapestry roots are in a position to know when reconstruction is necessary. Synchronization among the roots is simple. When a root determines that is will perform repair, it sends a message to the other roots informing them that it will repair the lost fragments. It then waits one hour for a response from each other root informing it that that other root will not perform a parallel repair (which could produce too many fragments). In the event that two or more roots send these messages simultaneously, the lowest ordered [4] root's message will be honored. After a response has been received from all of the other roots, or after one hour of having sent the message, the root performs the repair. The other roots wait one day after having received the repair notification message. If, after this time, no new fragments are advertised to them, the next-highest root modulo the number of roots will perform repair. The roots cycle through the repsonsibility for repair in this manner until the fragments are regenerated. By this mechanism, exactly one root will perform the repair (unless all of the roots are corrupt, an unlikely event unless we are operating in a domain dominated by malicious servers).

In the following Sections, we argue in favor of the above scheme. Fragments send heartbeats with a period of 1 month up one hop along their trees, and send additional heartbeats further up the tree with periods exponentially smaller than 1 month (with exponential factor of 1.5). Additionally, Tapestry nodes perform sweeps every 2 months at hop one, with higher hops performing sweeps exponentially less often (again, with exponential factor of 1.5).

6.3.1 Analysis of Distributed Repair

We must ensure that the schemes proposed in the previous section will not overly tax our network and CPU resources. We expect the responsible party to perform less work in repair than the Tapestry nodes, and we are unsure of its implementation. We therefore focus our analysis on the load on the Tapestry nodes.

First, we analyze the heartbeat scheme. To use our improved signature scheme, each pointer in Tapestry must store the top hash of the hierarchical hash of the next 5 signatures, so our storage overhead increases by another hash — 50% overhead. If we assume an exponential factor of 1.5 in our heartbeat scheme, the total number of heartbeats sent by each node per second is

$$8.39 \times 10^7 \frac{\text{fragments}}{\text{node}} \times \frac{\text{5 heartbeats}}{\text{fragment month}} \times \frac{\text{1month}}{\text{30days}} \times \frac{\text{1day}}{86,400\text{seconds}} \times \sum_{i=1}^{8} \frac{1}{1.5^{j-1}} = 467 \frac{\text{heartbeats sent}}{\text{second}} \tag{17}$$

and the total number received by each node per second is

$$8.39 \times 10^7 \frac{\text{fragments}}{\text{node}} \times \frac{\text{5 heartbeats}}{\text{fragment month}} \times \frac{\text{1month}}{\text{30days}} \times \frac{\text{1day}}{86,400\text{seconds}} \times \sum_{i=1}^{8} i \sum_{j=i}^{8} \frac{1}{1.5^{j-1}} = 3063 \frac{\text{heartbeats received}}{\text{second}} \tag{18}$$

These two numbers indicate that the required bandwidth per node is

$$467 + 3063 \frac{\text{heartbeats}}{\text{second}} \times 1568 \frac{\text{bits}}{\text{heartbeat}} = 5.54\text{Mbps} \tag{19}$$

[3] When a node has reported many false faults, it may be malicious. A reputation scheme can aid in detecting, reporting, and correcting for ill-behaved nodes. The implementation of such a scheme is beyond the scope of this paper.

[4] In Tapestry, multiple roots are assigned to a single object by hashing over that object's GUID and a small integer, in our case 1 - 5. The "lowester ordered" root is therefore the root that was reached using the smallest integer.

and the required CPU time is

$$
\begin{aligned}
(467\frac{\text{sent heartbeats}}{\text{second}} \times +3063\frac{\text{received heartbeats}}{\text{second}}) \\
\times\frac{\text{MAC}}{\text{heartbeat}} \times .00852\frac{\text{ms}}{\text{MACs}} \\
= 30.1\frac{\text{ms}}{\text{second}} \quad (20)
\end{aligned}
$$

The analysis of sweeps is similar. We analyse a scheme in which sweeps happen only one half as often as heartbeat checks. MACs are still used to verify fragment requests and the origins of fragments, so that a node between a storage server and a sweeping node cannot provide a fragment to the sweeper (thus preventing it from detecting a fault).

Each server will therefore request, and by symmetry, serve

$$
\begin{aligned}
8.39 \times 10^7\frac{\text{fragments}}{\text{node period}} \times 5\frac{\text{trees}}{\text{fragment}} \\
\times\frac{\text{1period}}{\text{2months}} \times \frac{\text{1moonth}}{\text{30days}} \times \frac{\text{1day}}{86,400\text{seconds}} \\
\times\sum_{i=1}^{8}\frac{1}{1.5^{j-1}} = \frac{\text{fragment request}}{\text{tree}} \\
= 229\frac{\text{fragment requests}}{\text{node second}} \quad (21)
\end{aligned}
$$

and each server will handle on average

$$
\begin{aligned}
8.39 \times 10^7\frac{\text{fragments}}{\text{node period}} \times 5\frac{\text{trees}}{\text{fragment}} \\
\times\frac{\text{1period}}{\text{2months}} \times \frac{\text{1moonth}}{\text{30days}} \times \frac{\text{1day}}{86,400\text{seconds}} \\
\times\sum_{i=1}^{8} i \sum_{j=i}^{8}\frac{1}{1.5^{j-1}} \frac{\text{fragment request}}{\text{tree}} \\
= 1532\frac{\text{fragment requests}}{\text{node second}} \quad (22)
\end{aligned}
$$

A server handles any request which it issues or that moves through it. The above formulation assumes that the number of hops from a pointer to the node to which it points is the same as the number of hops from the node to the pointer. By symmetry, each server will have to handle 1532 fragments per second. Both requests and fragments must be supplemented with 8 bytes of timestamp and 20 bytes of MAC. Thus, the total bandwidth required by each server for this scheme is

$$
\begin{aligned}
(1532\frac{\text{fragment requests}}{\text{node second}} \times 68\frac{\text{bytes}}{\text{fragment request}} \\
+1532\frac{\text{fragments}}{\text{node second}} \times 428\frac{\text{bytes}}{\text{fragment}}) \\
\times 8\frac{\text{bits}}{\text{byte}} = 6.08\text{Mbps per node} \quad (23)
\end{aligned}
$$

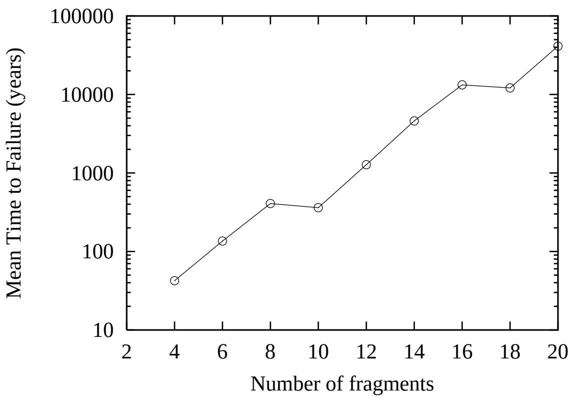

Figure 11: Mean time to failure for different numbers of fragments.

Each CPU must produce or verify four MACs for each fragment (two for the request, and two for the fragment), and must verify each fragment it receives on a sweep (5 hashes). The CPU time required for cryptography is therefore

$$
\begin{aligned}
(229\frac{\text{fragment requests}}{\text{node second}} \times 2\frac{\text{MACs}}{\text{fragment request}} + \\
229\frac{\text{fragments served}}{\text{node second}} \times 2\frac{\text{MACs}}{\text{fragments served}}) \\
\times .00852\frac{\text{ms}}{\text{MACs}} + 229\frac{\text{fragments served}}{\text{node second}} \\
\times\frac{\text{5 hashes}}{\text{fragments served}} \times .00426\frac{\text{ms}}{\text{hash}} \\
= 12.7\frac{\text{ms}}{\text{second}} \quad (24)
\end{aligned}
$$

This scheme is fairly inexpensive, in that it requires only 11.6% of the available bandwidth and 4.28% of the available CPU time.

6.3.2 Measurements of Distributed Repair

We simulated a global Tapestry with one billion nodes in order to measure the effectiveness of our scheme. We only simulated a single erasure-encoded object, because erasure encoded objects in the OceanStore will fail independently of one another. We also only simulated the distributed repair scheme – and not server heartbeats coupled with notification — because the simulation would

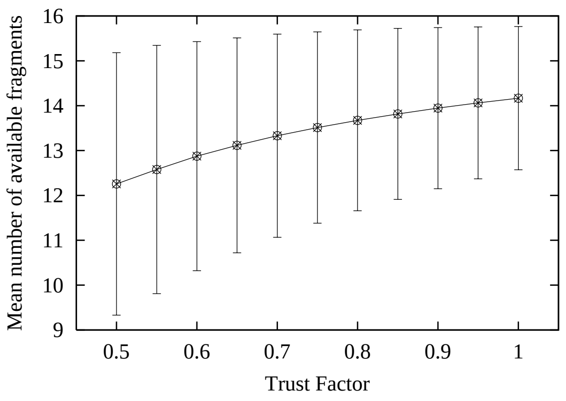

Figure 12: Number of available fragments for different trust factors.

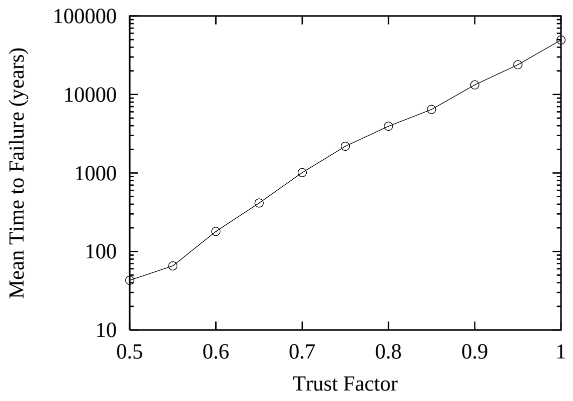

Figure 13: Mean time to failure for different trust factors.

not run to completion with notification (the mean time to failure would be too great). Each simulation used the responsible party to coordinate repair, meaning that Tapestry roots informed the repsonsible party of fragment losses. We included malicious parties in our simulation: a malicious storage server advertises a fragment, but does not store it; a malicious Tapestry node does not sweep and does not perform heartbeat checks, but does propagate false heartbeats up to its root — preventing any nodes between it and the root from passively detecting a failure. Our simulation reported two basic metrics for our scheme, namely mean time to failure and the average number of available (surviving) fragments. Each mean time to failure had an equivalent standard deviation of time to failure, a result expected because of the memoryless nature of lossy fragments and repair.

Unless otherwise stated, each simulation had the following parameters: 16 fragments with rate $\frac{1}{2}$ encoding; 5 redundant Tapestry location trees; a Tapestry tree height of 8; a Tapestry heartbeat period of one month and a sweep period of one month (with exponentiation factors of 1.5); a responsible heartbeat period of 2 months and a sweep period of 4 months; a failure trigger of 3 for heartbeat checks and sweeps (a fragment is declared dead only after 3 consecutive failures); and a repair threshold such that only after ceiling-one-quarter of the threshold number of fragments had been lost would the responsible party reconstruct.

6.3.3 Numbers of fragments

First, we measured the MTTF against different numbers of fragments. Figure 11 shows the results on a log graph. The MTTF grows exponentially with the number of fragments, but dips after 8 and 16 fragments. This dip occurs because after every 8 fragments, the number of fragment losses which must be detected before repair occurs increases. In the simulation with 8 fragments, the responsible party would only repair after one fragment loss had been detected, but with 10 fragments, it would wait until two losses had been detected. If we extraploate this graph with this trend, another dip occurs at 24 fragments, and the mean time to failure at 32 fragments is approximately 10^7 years. The mean number of available fragments in each simulation was consistently $1.7\times$ the threshold number of fragments, with very low standard deviations.

6.3.4 Trust

An important factor in our distributed repair scheme is that of trust. If we do not trust the infrastructure at all, we cannot rely on it to perform fault detection. If, on the other hand, we trust the infrastructure completely, we can rely on it alone to perform fault detection. Increasing trust increases the average number of fragments which are available at a particular instant in time and decreases the variability in the number of available fragments, as shown in Figure 12. The mean time to failure increases exponentially with increasing trust factors, as summarized in Figure 13. This result is interesting in that it indicates that Tapestry fault detection dominates the fault detection of the responsible party. Figure 14 shows the percentage of faults detected by each hop in Tapestry and by the Responsible Party. The important result here is that even for very low levels of trust, the Responsible Party detects less than one percent of all faults. This result is due in part to the reduced frequency of heartbeat checks and sweeps performed by the responsible party, and in part to the fact that each Tapestry hop is actually five nodes (to the responsible party's single node).

Figure 14 also indicates that the vast majority of faults

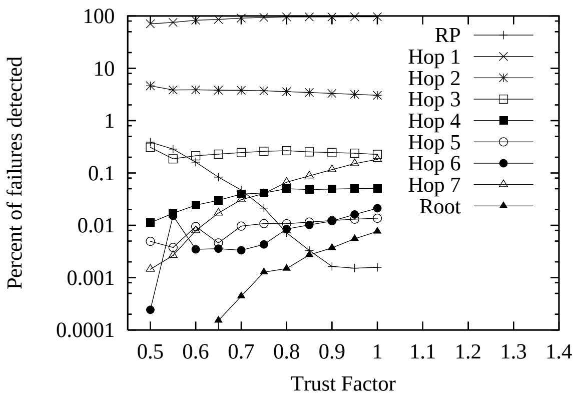

Figure 14: Percentage of faults detected by the Responsible Party and by different hops in Tapestry, as functions of the trust factor.

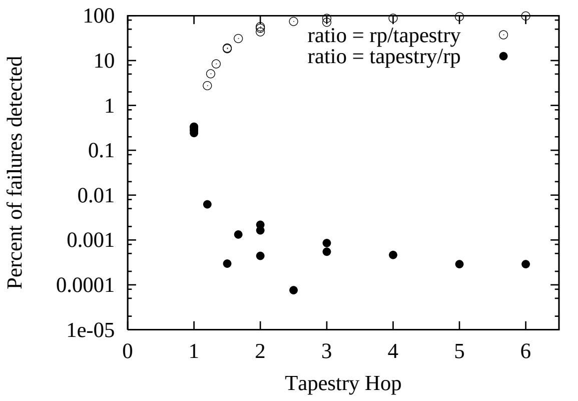

Figure 15: Percentage of faults detected by the Responsible Party versus the ratio between its detection period and the detection period of Tapestry

were detected by Tapestry nodes at hop 1. We expect this result, and it is a powerful argument in favor of our localization of repair. That other hops in Tapestry must perform periodic fault detection stems from the threat of regional outages, which we did not include in our simulations.

6.3.5 Repair Frequency

We further analyze the role of the responsible party in fault detection. We ran the simulation with sweeps always occurring only half as often as heartbeats, and we varied heartbeat checks for first-hop Tapestry nodes and the Responsible Party from 1 month to 6 months in 1 month increments. Figure 15 shows the percentage of the faults detected by the responsible party versus how often it performed sweeps and heartbeat checks relative to the frequency of Tapestry fault detections. The top line of points is plotted against the RP frequency over the Tapestry frequency, and the bottom line of points is plotted against the inverse of this ratio. The results indicate that unless the responsible party performs heartbeat checks and sweeps more often than Tapestry nodes, it is not helpful in fault detection. It is reasonable to assume that there will be fewer than one responsible party for every fifty Tapestry nodes. Recalling from Section 6.3.1 that the resource utilization of Tapestry nodes for distributed repair is 10%, it seems unlikely that we can make the responsible party perform fault detection faster than Tapestry nodes. Therefore, *the responsible party should not take part in fault detection*.

7 Related Work

Work in digital archives is not new. Since the inception of computing, programmers and administrators have been seeking better ways to preserve data for long periods of time. This Section presents a few of the projects most influential in today's digital archive research.

7.1 RAID

One of the first efforts towards making data more durable without changing the media on which it was stored was RAID [16] . The most commonly used form of RAID is Level 5, in which blocks of data across several disks share a block of parity information, and parity blocks are distributed among all of the disks in the array rather than on a single parity disk. While RAID's primary goal was to improve upon single disk cost-performance, this use of single-bit parity demonstrated that even a small amount of redundancy in spinning data could dramatically increase that data's expected lifetime. The mean time to failure of such an array was calculated to be an order of magnitude greater than data stored on a single disk.

7.2 Digital Libraries

Research in the area of digital libraries is now several years old. This field aims to take conventional information — like books, pictures, or video — and store it in an easily-used, strongly-durable format. Digital libraries attempt to address the archival goals of durability and usability. These systems strive to provide easy-to-use interfaces to vast amounts of data, not only for ease of reading and searching, but also for easy annotation of documents.

One project currently implementing a digital library is Robert Wilensky's Digital Library Project at UC Berkeley [20]. Wilensky's group uses a testbed called the

"California Environmental Digital Information System", an online repository of various documents pertaining to environmental information. This system stores its data on information servers which are implemented using a database management system. In this way, it leverages existing technologies for data storage, indexing, and retrieval. A user can enter a legacy document into this digital library by scanning it into a computer and filling in metadata about the document (such as author and title).

Users can search for information using an application implemented by Wilensky's team, or they can use a new extension to UC's Digital Library called Cheshire II [12] . Cheshire II provides a natural language search on a vast database of documents, and is capable of conventional Boolean results to searches as well as probabilistic matches to queries. Cheshire II moves the processing of queries to the servers storing the information, so that the client must only process the positive results of its queries.

UC's Digital Library Project also makes use of "multivalent documents" to [21] enable users to annotate documents with their own personal comments. One of the prime goals of this extension is to support document formats and document manipulations which have not yet been developed.

7.3 Internet Archive

A project very similar in spirit to digital library projects is the Internet Archive project [1] , begun in 1996 in order to permanently archive digital information of historical interest. The archival goals of the Internet Archive are durability and usability. The Internet Archive personnel note that much of the Internet (as well as other new forms of media like radio and television) is going largely unarchived; once a web site dies, the information on it is gone forever. They are also considering the deprecation of data formats, and are collecting emulators for data formats so that the information they store will be usable in the future. Currently, the Archive boasts 43 Terabytes of saved data. Unfortunately, it is not highly available. To use the archive, one must fill out and submit a proposal. Additionally, the Archive's users must be proficient in Unix programming. The Archive's mechanisms for durability are conventional; data is stored on a series of Linux box hard drives and tapes.

7.4 Intermemory

A work currently in progress which closely resembles the OceanStore Archive is Intermemory [9]. Concerned primarily with the archival goal of durability, this work introduced the distribution of erasure-encoded fragments into the wide area as an ideal mechanism for archival storage. It is a subscriber-based, peer-to-peer storage infrastructure. A user of an Intermemory donates for a small amount of time an amount of storage for use by other members, and in exchange receives a smaller but more permanent amount of storage in the Intermemory for his own personal use. For example, Bob could provide one Gigabyte of his own disk space to an Intermemory for one year; in exchange, he would receive two hundred Megabytes of space in that Intermemory for the rest of its existence.

By breaking objects up into 32 or even 1024 fragments, Intermemory uses the law of large numbers to help ensure the long-term durability of the data it stores. Because Intermemory deals in such large durabilities, it uses levels of indirection in its location scheme, so that a single virtual Intermemory address can be used to retrieve all of an object's fragments even after all of the original storage servers are long dead.

Intermemory's repair mechanism replaces dead storage servers with fresh Interemory daemons [5]. The fault detection in Intermemory is at a server granularity; each server has thirty-two "neighbors", each of which poll the server to determine if it is still alive. If it fails to respond after too long a period, the system will replace it with a new node. In this way, a particular server's archived data is as robust as that server's thirty-two neighbors (that is, its archived information survives so long as one-half of its neighboars survive). Additionally, by reconstructing a logical fragment on a new machine using Intermemory's protocols, the system solves the *media conversion* problem (so long as the information was converted to Intermemory's encoding format when it was originally archived). Unfortunately, this scheme places the burden of fault-detection onto the client, which may be infeasible for those clients with intermettitent or low-bandwidth connections. The authors of Intermemory also briefly mention what they call *archival semantics*, meaning that the format of an archived object may eventually become unsupported, making the document unreadable. They recommend the use of emulators to make such documents usable as long as they are durable.

7.5 PAST and FarSite

Persisten and Anonymous Storage in a Peer-to-Peer Networking Environment (PAST) [7] is a project strongly similar to OceanStore. Their archival goals are durability, availability, and, because they support storage on the untrusted wide area, security. They seek to achieve these goals by replicating files stored in PAST on several machines, locatable by their overlay-network routing layer, Pastry [5]. No repair scheme has yet been described for documents stored in PAST, implying that their current durability is comparable to that of today's disk drives (an approximate MTTF of five years).

Another project out of Microsoft Research is FarSite [4]. FarSite is similar to PAST in that it ensures file durability through replication. Unlike PAST, FarSite is not intended to scale globally, but security is still one of FarSite's goals, since they assume the presence of incompletely trusted clients. While FarSite does not describe a fault-detection or repair scheme, they do focus on the availability metrics of their system. In particular, they observe that a document in FarSite is highly available because all of the machines storing it must be down for it to be inaccessible.

8 Future Work

Two key aspects of the OceanStore which are strongly related to repair still require discussion. First, there is the problem of billing. OceanStore is intended to be a storage utility. Therefore, storage servers must be able to charge users for the data they store. The responsible party is ideally suited to this task, becaue it is financially charged with the survival of its clients data. Thus, storage servers charge fragments' responsible parties, and therefore fragments must carry with them the GUIDs of their responsible parties. A security problem is also tightly related to both billing and repair. A malicious server could potentially store all of an object's fragments, advertise them to Tapestry, and produce them when asked by a sweep. This server would thus prevent Tapestry and the responsible party from detecting any failing fragment. Presumably, once enough of the fragments on the legitimate storage servers had died, the malicious server would stop storing the fragments, effectively killing the document.

A solution to both of these problems must use a method of authenticating storage servers. That is, there must be a key which the responsible party stores in addition to the GUID of an erasure-encoded object, and this key must validate the GUID of any server claiming to store a fragment. A server billing the responsible party for a fragment includes in the bill the fragment's GUID and additional information which will produce the billing key for the fragment. One simple way to produce such a key is to use a hierarchical hash over the storage servers' GUIDs; each server would include in its bill the sibling hashes necessary to hash from its server GUID to the object's key. Whenever repair occurs, this key must be updated to reflect the server GUIDs of new storage servers. The key can also be used by Tapestry nodes to verify that a particular server is a legitimate storage server for a fragment.

[5]For a thorough description of the mechanisms and algorithms used in Pastry, see http://www.cs.berkeley.edu/ ravenben/publications/CSD-01-1141.pdf

9 Conclusion

There is a growing demand for distributed data archives. OceanStore seeks to meet most of the application demands for *preserving the bits*, and leaves the problem of archival semantics to application developers. OceanStore ensures the integrity of its data by means of cryptograhpic hashes, and it provides high durability and availability of its data using erasure codes. Essential to the use of erasure codes is the presence of fault detection and repair. We have seen that, using Tapestry, the system can efficiently detect the failures of fragments through pointer heartbeats and by periodically sweeping the data. We have also seen that it is unnecessary to use the responsible parties in OceanStore for fault detection. The durability and availability of objects in the distributed repair schemes presented in this work are sufficient for maintaining documents for thousands of years.

References

[1] Internet archive. http://www.archive.org/xterabytes.htm.

[2] Summary on linear vs. helical recording technologies in entry-level to mid-range tape backup products. 1998.

[3] ANDERSON, T., CULLER, D., AND PATTERSON, D. A case for now (networks of workstations), 1995.

[4] 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).

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

[6] CHERVENAK, A. L. Tertiary storage: An evaluation of new applications, 1994.

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

[8] GILDER, G. Fiber keeps its promise: Get ready. bandwidth will triple each year for the next 25. *Forbes* (Apr. 1997).

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

[10] GRAY, J., AND SHENOY, P. Rules of thumb in data engineering. Tech. Rep. MS-TR-99-100, Microsoft Research, Mar. 2000.

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

[12] LARSON, R., AND CARSON, C. Information access for a digital library: Cheshire ii and the berkeley environmental digital library. In *Proceedings of the 62nd ASIS Annual Meeting* (Nov. 1999).

[13] 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).

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

[15] PATTERSON, D., GIBSON, G., AND KATZ, R. A case for redundant arrays of inexpensive disks(raid). In *Proceedings of 1988 ACM SIGMOD International Conference on Management of Data* (1988).

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

[17] 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.

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

[19] ROE, M. Performance of protocols. In *Proceedings of Security Protocols Workshop* (1999).

[20] WILENSKY, R. Toward work-centered digital information services. *IEEE Computer Special Issue on Digital Libraries* (May 1996).

[21] WILENSKY, R. Digital libraries resources as basis for collaborative work. *Journal of the American Society for Information* (Feb. 2000).

[22] 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.