ROBUST RESOURCE ALLOCATION IN DISTRIBUTED FILESYSTEMS

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Computer Science

By

Kevan Carstensen

2011

SIGNATURE PAGE

THESIS:	ROBUST RESOURCE ALLOCATION IN DISTRIBUTED FILESYSTEMS
AUTHOR:	Kevan Carstensen
DATE SUBMITTED:	Fall, 2011
	Computer Science Department

Dr. Craig A. Rich _____
Thesis Committee Chair
Computer Science Department

Dr. Gilbert Young _____
Computer Science Department

Dr. Lan Yang _____
Computer Science Department

Abstract

Distributed filesystems are an increasingly popular technology, especially amongst cloud computing providers. Distributed filesystems aggregate the storage and processing power of multiple computers into a coherent filesystem, then allow users to store data on that filesystem. Distributed filesystems must address a number of challenges to provide the fault-tolerance and availability that users expect. We study file allocation, one such challenge. We present servers of happiness, an algorithm for file allocation in the widely used open-source Tahoe-LAFS distributed filesystem. We compare the servers of happiness algorithm to its predecessor, the shares of happiness algorithm. The servers of happiness algorithm improves on the shares in two important ways, and should make an appealing addition to Tahoe-LAFS at some point in the future.

Contents

List of Tables

List of Figures

Chapter 1

The Filesystem as an Abstraction

Informally speaking, a filesystem is the set of algorithms and storage structures responsible for low-level file management on a computer. An essential task for such a system is writing data to physical storage in such a way that it can be retrieved on-demand. Filesystems commonly have other features as well. For example, filesystems may need to associate metadata with files, or to provide a mechanism for access control. We will only consider storage and retrieval of files in our work, however; these other features are mostly orthogonal to the issue of robustness and efficiency, and we do not wish to cloud our analysis by considering them.

More formally, we consider a filesystem F as the following operations.

- A set P of allowable paths, representing possible storage locations.
- An operation $\mathsf{STORE}_{\mathsf{F}}(\mathtt{path} \in \mathsf{P}, \mathtt{data} \in \{0,1\}^*) \to \{0,1\}$ that associates the data $\mathtt{data}$ to the path $\mathtt{path}$, returning 1 if the operation was successful or 0 otherwise.
- An operation $\mathsf{RETRIEVE}_{\mathsf{F}}(\mathtt{path} \in \mathsf{P}) \to \{0,1\}^*$ that returns the data associated to $\mathtt{path}$, or nothing if no data has been stored at $\mathtt{path}$.

Each operation is parameterized with F, which we use to represent the state associated with a filesystem. In a filesystem on a single disk, F would include the single disk. For a filesystem spread over many disks or many computer systems, F would include those computer systems. To understand the need for a state variable, consider an intuitive, high

level implementation of $\mathsf{STORE}_\mathsf{F}$ and $\mathsf{RETRIEVE}_\mathsf{F}$. We expect that $\mathsf{STORE}_\mathsf{F}$ will write its `data` parameter somewhere, and we expect that $\mathsf{RETRIEVE}_\mathsf{F}$ will retrieve some data from somewhere, where somewhere is a piece of physical storage. In essence, F encapsulates and formalizes the notion of somewhere.

1.1 Centralized filesystems

For common and traditional filesystems, the state used by the STORE and RETRIEVE operations all exists on a single physical disk. We will refer to such filesystems as centralized filesystems; the motivation behind this designation is the availability characteristics that result from using a single physical disk to store data, metadata, and everything else necessary for the filesystem to function. We will discuss these characteristics later.

The ext3 filesystem

ext3 [5], an evolution of ext2 [6], is a common filesystem on GNU/Linux systems. We will briefly examine, at a high level, how $\mathsf{STORE}_{\mathsf{ext3}}$ and $\mathsf{RETRIEVE}_{\mathsf{ext3}}$ are implemented.

Generally, an ext3 filesystem is organized in a familiar tree structure corresponding to a directory hierarchy. Some of the contents of the directory hierarchy are created and modified by users, while others are managed by the operating system (e.g., the device hierarchy). Since we're considering the user-facing $\mathsf{STORE}_{\mathsf{ext3}}$ and $\mathsf{RETRIEVE}_{\mathsf{ext3}}$, we will consider only the common case of files and directories. Both files and directories are defined by inodes, special objects in the filesystem that contain metadata about files (e.g., type, permissions and size) as well as pointers to the disk blocks that store the data associated with the file. Directories are implemented as files with specifically-formatted and defined data; that is, a mapping of filenames in the directory to the inodes associated with those files. [2]

Using this information, it is easy to see how $\mathsf{RETREVE}_{\mathsf{ext3}}$ would work. Given a path, say, `/etc/passwd`, it would read the `etc` entry of the root directory node and figure out which inode corresponds to `etc`. It would then read the the `passwd` entry from the `etc` directory. Finally, it would retrieve the contents of the data blocks associated with the `passwd` inode and return them to the caller. If $\mathsf{STORE}_{\mathsf{ext3}}$ is invoked on an existing path,

then it behaves in much the same way: it traverses the directory tree until it finds a leaf node, then it writes data to the data blocks associated with the leaf node, allocating new blocks as necessary. Similarly, if a new file is created, $\mathsf{STORE}_{\mathsf{ext2}}$ must traverse the directory tree until it finds the parent directory of the new file, select an inode to store metadata about the new file, and then allocate blocks to store the data for the new file.

Evolutions of centralized filesystems

Technologies like RAID [14], ZFS [16], and LVM [17] allow conceptual filesystems to straddle disk boundaries, so that the availability of the filesystem is not necessarily tied to the availability of one disk. These, then, are not centralized filesystems in the same way as ext3. We argue that they are still conceptually centralized, though, since their availability cannot generally be decoupled from a single computer system. Their availability, in other words, is still dependent on a single and relatively small conceptual entity with a non-negligible probability of failure. Decentralized systems, as we'll see shortly, are designed not to depend on any one disk, computer, or conceptual entity with a high probability of failure.

Weaknesses of centralized filesystems

As mentioned, centralized filesystems generally rely on a single physical disk for the persistence of the state they need to operate correctly. This makes filesystem availability dependent on the functioning and availability of a single physical disk, whose failure is sufficient to render the entire filesystem unavailable. If a filesystem is stored entirely on disks within one computer, as in RAID, LVM, or ZFS, the filesystem's availability is dependent on the availability of the computer. These availability limitations make it difficult to use centralized filesystems to implement robust storage systems that are resilient to regional catastrophes, or even somewhat widespread power and network failures. Some level of robustness can be achieved, of course – we can keep offsite backups, for example – but these solutions are orthogonal to the filesystem itself and generally require human intervention and induce some level of temporary availability loss while, e.g., backups are restored or a

failover site is activated. Decentralized filesystems, as we'll see, are designed to scale beyond a single computer, which yields a pleasant and configurable level of robustness while preserving the relatively simple filesystem abstraction.

1.2 Improving availability with a decentralized filesystem

Suppose that we want to define a filesystem F with generally good availability properties in the face of common and less common failure scenarios. We will assume that the filesystem has a set C of computers that it can use. We suppose that each $\mathsf{c} \in \mathsf{C}$ supports $\mathsf{STORE}_\mathsf{c}$ and $\mathsf{RETRIEVE}_\mathsf{c}$ operations that are similar to those used in the previous section, and that F may use these as it sees fit. Intuitively, these operations might be implemented on a given computer by a local filesystem like ext3. Since computers and their supporting infrastructure can and do fail at points, we'll analyze various implementations of F in the context of failures of some of the computers in C. We'll assume that computers in C fail with some probability, and that the failure of a computer is expressed in binary terms – the computer is either working perfectly or not working at all.

How to evaluate the availability and efficiency of a filesystem

In the following sections, we will reason about the availability properties of a distributed filesystem. We will also consider the space used to store a particular bit pattern in the filesystem. We will see that there is usually a tradeoff between the availability of a piece of data and the amount of space that is used to store a particular piece of data: specifically, we typically get better availability properties if we store more bits than we were asked to store. We will first establish various ways to evaluate the performance of a filesystem in terms of availability and storage utilization. This will give us a consistent, objective way to evaluate and compare our constructs later.

As stated above, we wish to consider storage utilization. When considering storage utilization, we will consider the number of bits that must be stored on a particular filesystem in order to store a bit pattern of length $\mathfrak{n}$, ignoring constant factors due to disk and local filesystem structures. Using this measurement, we can compare two filesystems by

comparing the number of bits that each uses to store a particular bit pattern. For example, a filesystem that must write $3n$ bits to store an n-bit datum is inferior in terms of storage to a filesystem that only writes $2n$ bits to store the same datum.

We also wish to consider availability. When evaluating decentralized filesystems, we will consider three availability states.

First, we will consider the availability of a decentralized filesystem as a whole. Let F be a filesystem. We will say that F as a whole is available if there exists a path $\mathtt{p}$ in the set of valid paths such that $\mathsf{STORE}_\mathsf{F}$ and $\mathsf{RETRIEVE}_\mathsf{F}$ work correctly; that is, $\mathsf{STORE}_\mathsf{F}(\mathtt{p}, \mathtt{data})$ associates $\mathtt{data}$ with $\mathtt{p}$, and $\mathsf{RETRIEVE}_\mathsf{F}(\mathtt{p}) = \mathtt{data}$. A filesystem that is available as a whole is minimally functional or partially available; it may work correctly for some paths but may return missing data or no data for other paths, or it may work correctly for all paths.

We will also consider total availability. We say that a filesystem F is totally available if $\mathsf{STORE}_\mathsf{F}$ and $\mathsf{RETRIEVE}_\mathsf{F}$ work correctly for all possible paths. A filesystem that is totally available may be regarded as working normally.

Finally, a filesystem F may be entirely unavailabile, in which case $\mathsf{STORE}_\mathsf{F}$ and $\mathsf{RETRIEVE}_\mathsf{F}$ do not work correctly for any path.

These three availability states can be thought of as a continuum of availability whose endpoints are total unavailability and total availability. No system may be better than totally available, or worse than totally unavailable; systems in between these two are partially available. We may compare partially available filesystems F_1 and F_2 by comparing the number of paths for which they function correctly, ranking systems with more functioning paths higher than those with fewer functioning paths. When comparing filesystems, we will also reason about the difficulty of going from one state on this continuum to another. Since we will primarily be speaking of failure, most of our state transitions will be down the continuum, i.e., from totally available to partially available, or from partially available to unavailable. We will usually express this by considering the number of participating servers that must fail before availability is degraded, and will rank filesystems by comparing these numbers. For example, if F_1 will go from totally available to partially available if 3 servers fail, and F_2 will go from totally available to partially available if 5 servers fail, then we will

regard F_2 as superior to F_1 in terms of availability.

Viewing a centralized filesystem in our terminology

We can view a simple centralized filesystem F_c in these terms. Specifically, we select a $c \in C$ that is held constant for all operations in F_c. Intuitively, this c is the computer that we're using — if the centralized filesystem is on your notebook computer, then c is your notebook computer. Then we define STORE_{F_c} and $\mathsf{RETRIEVE}_{F_c}$ as follows:

$$\mathsf{STORE}_{F_c}(path, data) = \mathsf{STORE}_c(path, data)$$
$$\mathsf{RETRIEVE}_{F_c}(path) = \mathsf{RETRIEVE}_c(path)$$

Then the filesystem F_c essentially delegates all of its tasks to the underlying filesystem on c.

First, note that F_c is relatively efficient in terms of space. For an input datum of length n, it stores n bits. Short of compressing or damaging the data, F_c cannot store fewer than n bits to store an input datum whose length is n bits.

Since F_c is defined in terms of a single computer, its availability properties are very closely tied to those of a single computer. By earlier assumption, the availability of a particular computer is binary, so there is no middle ground in F_c between total availability and total unavailability. In other words, only one failure — the failure of c — is necessary to cause F_c to lose all of the data it is responsible for storing.

A simple distributed filesystem

We will now consider a filesystem that is very much like F_c, but with a simple elaboration to make it distributed. Instead of selecting one computer c to store data, we will select a set $N \subseteq C$ computers to store data on. We will call this filesystem F_s, and will define it as follows:

$$\mathsf{STORE}_{\mathsf{F_s}}(\mathtt{path}, \mathtt{data}) = \mathsf{STORE}_{\mathtt{find}_{\mathsf{N}}(\mathtt{path})}(\mathtt{path}, \mathtt{data})$$

$$\mathsf{RETRIEVE}_{\mathsf{F_s}}(\mathtt{path}) = \mathsf{RETRIEVE}_{\mathtt{find}_{\mathsf{N}}(\mathtt{path})}(\mathtt{path})$$

We have defined an internal function, $\mathtt{find}_{\mathsf{N}}$. $\mathtt{find}_{\mathsf{N}}$ selects a computer from N given a path $\mathtt{path}$, and does so consistently: i.e., $\mathtt{path}_1 = \mathtt{path}_2 \Leftrightarrow \mathtt{find}_{\mathsf{N}}(\mathtt{path}_1) = \mathtt{find}_{\mathsf{N}}(\mathtt{path}_2)$. $\mathtt{find}_{\mathsf{N}}$ is essentially a hash function whose buckets are the computers in N. Intuitively, when presented with some data to be stored at a path, $\mathsf{F_s}$ chooses a computer from the set of acceptable computers N and stores the data on that computer.

We see first that $\mathsf{F_s}$ is as efficient as the centralized filesystem, since, for each n-bit datum, n bits are stored in the filesystem.

Unlike in $\mathsf{F_c}$, there is a distinction in $\mathsf{F_s}$ between total availability and partial availability. $\mathsf{F_s}$ is partially available at time t if at least 1 computer in the initial set N is still working at time t, since, assuming an efficient $\mathtt{find}_{\mathsf{N}}$, there is some path p whose data is stored on the remaining server. All of the computers in N must be broken before $\mathsf{F_s}$ is totally unavailable. Then, assuming that $|\mathsf{N}| > 1$, $\mathsf{F_s}$ offers better resistance to total unavailability than $\mathsf{F_c}$. $\mathsf{F_c}$ and $\mathsf{F_s}$ both require only one server to fail before they degrade from a totally available state, so $\mathsf{F_s}$ is neither better nor worse than $\mathsf{F_c}$ in that regard. Then, since $\mathsf{F_s}$ can tolerate the failure of more servers than $\mathsf{F_c}$ before becoming totally unavailable, and since $\mathsf{F_s}$ can tolerate the same number of server failures as $\mathsf{F_c}$ before degrading from a totally available state, we say that $\mathsf{F_s}$ offers better availability characteristics than $\mathsf{F_c}$. $\mathsf{F_s}$ is also no worse than $\mathsf{F_c}$ in terms of storage utilization.

This construct is fairly similar to a data structure called a distributed hash table. Distributed hash tables are conceptually similar to more familiar hash tables, in that they map some key to a location. In a normal hash table, the location might be a linked list, if the hash table is a bucket chaining hash table. In a distributed hash table, the location is typically a computer on a network. Distributed hash tables that wish to avoid relying unduly on one or a few constituent computers typically include algorithms run on data-storing clients to find the computers responsible for storing those data, and other algorithms for

network maintenance, including allowing nodes to join or leave a network, and algorithms to periodically balance load between nodes. Distributed hash tables have been used to implement trackerless BitTorrent [11]. Popular distributed hash tables include Chord [15] and Kademlia [12].

Improving the simple distributed filesystem with replication

The primary weakness of F_s is that its total availability — the property that it works correctly for all paths — is dependent on the correct functioning of all of its constituent computers. F_c and F_s share this property, but the effect is more pronounced with F_s since the probability that any one computer of a large number of computers will have broken at some point in time is greater than the probability that one specific computer will have broken at some point in time. We can address this by decoupling the availability of data stored at any one path from the functioning of a particular computer. In this way, the failure of one or a few computers does not necessarily render the filesystem partially unavailable. We do this by storing multiple copies or replicas of data within the filesystem: this process is called replication.

F_r, a distributed filesystem employing replication, is very similar to F_s. We can define it as follows:

$$\mathsf{STORE}_{\mathsf{F}_\mathsf{r}}(\mathtt{path}, \mathtt{data}) = \text{for } \mathsf{p} \in \mathtt{find}_\mathsf{N}(\mathtt{path}), \mathsf{STORE}_\mathsf{p}(\mathtt{path}, \mathtt{data})$$

$$\mathsf{RETRIEVE}_{\mathsf{F}_\mathsf{r}}(\mathtt{path}) = \text{for } \mathsf{p} \in \mathtt{find}_\mathsf{N}(\mathtt{path}), \mathsf{RETRIEVE}_\mathsf{p}(\mathtt{path})$$

We've changed the internal function $\mathtt{find}_\mathsf{N}(\mathtt{path})$ to return a set of computers rather than a single computer. The size of the set returned by `find` controls the amount of replication present in the network, and might be a configuration option. We will refer to this size as r. We then store the data associated with that path on each of those computers. Retrieval operates in the expected way: each of the computers returned by the locator function is asked for the data it has stored at a given path.

We first note that F_r is not as efficient in terms of space as F_c or F_s. For some n input bits, F_r stores $r \cdot n$ bits. Recall that F_s and F_c both store n bits which, assuming $r > 1$, is smaller than $r \cdot n$.

When we consider partial availability, F_s is slightly superior to F_r. Note that F_s may still operate normally for some path if only one computer is still operating. F_r may retrieve the data associated with a particular path if only one computer is still operating, but at least r computers must be operating for it to store data at a particular path. So, in order for F_r to be partially operational by our terminology, at least r computers must be online at once. When we consider total availability, though, F_r is superior to F_s. Recall that F_s may enter a state in which it is nonfunctional for some paths if only one server goes offline. F_r can retrieve data for all paths if up to $r-1$ of its constituent systems are offline. In other words, F_r trades space efficiency for a greater margin of error before it degrades into only partial availability. Specificially, trading an additional n bits of redundancy allows us to increase by 1 the number of servers that can fail before some files become unavailable.

F_r is similar to the Dynamo construct [8], developed by Amazon.com and used internally for various pieces of infrastructure. Dynamo deals with concepts of write consistency that F_r does not.

We can view a storage operation as a number of individual requests, each directing a selected storage server to store a copy of the input data. We would like for the overall storage operation to "return" quickly, so that programs using Dynamo need not block on storage requests for a long time, but we would also like to ensure write consistency. A storage operation is consistent if each of the storage servers selected to hold a copy of some data successfully stores a copy of the data. By achieving write consistency, we can ensure that future attempts to read data stored at a path return the data that we stored, and not some other data that was there previously or no data at all. An inconsistent state can occur if the storage operation fails to successfully store a data on one of the selected servers; for example, if there are network issues, or if one of the servers is temporarily down for maintenance. To control the tradeoff between responsiveness and consistency for later reads and updates, Dynamo allows applications to tune the number of servers that must return successfully before a write can be called successful. Applications for which consistency is

very important might set this to be equal to the number of replicas, while applications in which performance is more important would set it to a smaller value.

Using erasure coding to temper the space tradeoff

In the previous section, we saw how introducing some inefficiency in terms of storage space made F_r tolerate more server failures before degrading from a state of complete availability. Specifically, storing additional replicas of data allows the system to tolerate more server failures before losing the data. Erasure coding allows us to get more bang for our space tradeoff buck. Erasure coding relies on an abstract function that takes in some arbitrary data, and outputs n pieces of that data such that $k \leq n$ distinct pieces are sufficient to restore the original input data when fed into a different (and known) function. The combination of a particular k and n is often called a k-of-n encoding, since k of the original n pieces are required to reconstruct some given input data.

Lets consider F_e. F_e is like F_r, except that a file is erasure coded before being replicated. Let n and k be the parameters of the erasure code. Note that the space tradeoff now depends on n and k. In general, we have $\frac{n}{k}$ bits of data stored on the grid for every input bit. Formally, we define STORE_{F_e} as follows:

Input: $path$, $data$

$D = ec(data, n, k)$

for all $x \in find_n(path)$ **do**

Choose a $d \in D$

$\mathsf{STORE}_x(path, d)$

$D = D \setminus \{d\}$

end for

and we define $\mathsf{RETRIEVE}_{F_e}$ as:

Input: $path$

Choose $R \subseteq find_n(path), |R| \geq k$

$D = \{\}$

for all $s \in R$ **do**

$D = D \cup \{\text{RETRIEVE}_s(path)\}$

end for

return $decode(D, n, k)$

Intuitively, the store operation takes input data, produces erasure-coded chunks of that input data, then puts one chunk on each server. The retrieve operation selects k servers from the set of servers that might have stored data for the input path, retrieves the erasure-coded chunks from those servers, then reassembles the data and returns it. We've also introduced two new private helper functions. $ec(data, n, k)$ erasure-codes some input data using the parameters n and k. $decode(D, n, k)$ takes as input a set of at least k erasure-coded chunks and decodes them to the original input data.

To understand the availability and efficiency implications of erasure coding, we must closely compare it to a system using replication. Specifically, we wish to see what happens when r, n, and k are set so that F_e and F_r store the same amount of data for a given amount of input data. Then $\frac{n}{k} = r$, and $n = k \cdot r$. If we choose n and k such that the erasure coding is not simply replication (as would be the case if $k = 1$), then we have that $n > r$.

Note that, by assumption, we have that $r = \frac{n}{k}$. For m bits of input data, F_r stores $m \cdot r$ bits and F_e stores $\frac{n}{k} \cdot m$ bits of input data, so F_r and F_e are equivalent in terms of storage utilization.

Next, we consider the number of server failures that F_r and F_e can tolerate before failing completely. In the case of F_r, at least r servers must operate in order for storage and retrieval to work for some paths. In the case of F_e, at least k such servers must operate. There is no clear winner here, since we don't have a relationship between k and r — all we know is that $k > 1$.

The big difference between F_e and F_r is in total availability. Note that F_r can tolerate the failure of $r - 1$ servers while remaining fully functional. F_e can tolerate the failure of $n - k$ servers while remaining fully functional. Note that

$$r - 1 = \frac{n}{k} - 1$$

$$(r - 1)k = n - k$$

Since $k \geq 1$, F_e can tolerate at least twice as many server failures as F_r before potentially losing data, despite the fact that it stores the same amount of data for a given input bit pattern. In other words, F_e allows us to make a distributed filesystem that can tolerate many more server failures than F_r before degrading from a state of total availability.

This construct is similar to Tahoe-LAFS [9], a distributed filesystem that offers fault-tolerance as well as confidentiality that is independent of the provider of data storage. In other words, data stored in a Tahoe-LAFS grid remains secret even from the operators of the computer systems that store it. This is achieved through pervasive use of encryption. Tahoe-LAFS also uses capabilities for access control. We will examine Tahoe-LAFS in depth later in this thesis.

An example

To better understand the interplay of the variables used in previous sections, we will now analyze a more concrete distributed filesystem construct. Suppose that we have a set of computers N such that $|N| = 20$. In other words, we have 20 computers that we can use to build a filesystem.

First, we will consider F_c, a centralized filesystem. As before, F_c is unaffected by the size of N. It will store n bits if asked to store an n-bit bit pattern, will function with total availability if its only computer is online, and will be totally unavailable if one computer fails.

Next, consider F_s, a simple distributed filesystem. Like F_c, it stores n bits if asked to store an n-bit bit pattern. It will be partially available so long as no more than 19 computers fail, but will only be totally available if all 20 computers are running.

Next, consider F_r, a distributed filesystem using replication. We'll set r to 3, which means that each bit pattern will be replicated three times on computers in N. Clearly, F_r will store $3 \cdot n$ bits when asked to store an n-bit input file. F_r can be guaranteed to be totally available if up to 2 servers fail, and will remain partially available if no more than 17 servers fail.

Finally, we consider F_e. As in the previous section, we will choose n and k such that

Table 1.1: Comparison of F_c, F_s, F_r, and F_e

Filesystem	Expansion Factor	Failures before partial loss	Failures before total loss
F_c	1	0	1
F_s	1	0	s
F_r	r	$r-1$	s
F_e	$\frac{n}{k}$	$n-k$	$s-k$

$\frac{n}{k} = r$; $n = 15$, $k = 5$ satisfies this constraint. Note that F_e, configured as such, will store the same number of bits for an input bit pattern as would F_r. F_e is partially available if no more than 15 computers have failed at a particular time, and F_e is totally functional if no more than 10 computers have failed at one time.

The following table summarizes the availability properties of the filesystem designs examined in this section. s is the number of servers. r is the replication parameter for F_r. n and k are erasure coding parameters for F_e, $k > 1$, set such that $\frac{n}{k} = r$. Recall that $n - k \geq 2(r-1)$, as demonstrated above.

1.3 Examining implicit assumptions about decentralized filesystems

In the analysis above, we examined the storage utilization and availability characteristics of four example decentralized filesystems. By their design, these examples make some assumptions about how to store data on computers that compose a decentralized filesystem. These assumptions are shared by the practical allocation strategies that we will study later. We will now state these assumptions explicitly, analyze them, discuss why we wish to make them, and discuss any downsides associated with them.

Share allocation assumptions

In our analysis, we discussed two filesystems, F_r and F_e, that can use more than one computer to store some input data. Specifically, F_r stores copies of the entire input data, while F_e stores erasure-coded shares of the input data. We'll call the copies and erasure-coded shares output data; they are the data that F_e and F_r write as output when asked to store something.

Both F_e and F_r attempt to store no more than one piece of output data on any one storage server; in other words, they attempt to spread out their output data . This is a deliberate design decision; clearly, F_e and F_r could be allowed to store more than out piece of output data on a given storage server. We will now examine the consequences of changing this assumption.

First, we consider F_r; specifically, we consider what happens when we invoke $\mathsf{STORE}_{F_r}(\mathtt{path}, \mathtt{data})$ for some $\mathtt{path}$ and $\mathtt{data}$. Suppose that r is the number of replicas placed by F_r. As above, F_r will store a copy of $\mathtt{data}$ on each of r different storage servers. This means that $\mathsf{RETREIVE}_{F_r}(\mathtt{path}) = \mathtt{data}$ as long as no more than $r - 1$ of those storage servers have failed. More generally, F_r can be said to be totally available if no more than $r - 1$ storage servers have failed. Now suppose that $\mathsf{STORE}_{F_r}(\mathtt{path}, \mathtt{data})$ is permitted to store more than one of its r copies of $\mathtt{data}$ on a given storage server. Let u be the number of distinct storage servers used by STORE_F, and assume that $u < r$. Then this modified form of F_r can tolerate up to $u - 1$ failures before $\mathsf{RETRIEVE}_F(\mathtt{path}) \neq \mathtt{data}$. In general, the modified form of F_r can tolerate $U - 1$ failures before degrading from a state of total availability, where $U = \min(\{u : p \in P\})$; the smallest number of distinct storage servers storing replicas for a particular path. In other words, allowing F_r to store multiple replicas on a single storage server reduces F_r's ability to tolerate server failures while remaining in a state of guaranteed total availability. Replication of this sort does not provide any benefits when we consider degradation from partial availability to total unavailability. Recall that F_r can be regarded as partially available if at least one storage server with a replica of the data associated with some path is online. This does not change if we allow storage servers to store multiple replicas of input data. In summary, allowing F_r to store multiple replicas on a single storage server harms the ability of the filesystem to tolerate server failures before degrading from a state of total availability, and doesn't improve F_r's ability to resist degrading into a state of total unavailability. In fact, the availability characteristics of an F_r that stores r replicas but doesn't guarantee that they will be stored on distinct storage servers are the same as the availability characteristics of an F_r that does guarantee that replicas will be stored on distinct storage servers, but which has a smaller replication constant — U — and therefore stores less data. So there is little reason to allow F_r to store more than one replica on each

storage server.

Next, we consider F_e. Let n and k be erasure coding parameters. Recall that F_e can tolerate the failure of up to $n - k$ storage servers before degrading from a state of total availability. Now suppose that $\mathsf{STORE}_{F_e}(path, data)$ is allowed to place more than one distinct erasure coded piece onto a particular storage server. As above, this reduces the ability of F_e to tolerate the failure of storage servers without degrading from a state of total availability; specifically, if u is the number of distinct storage servers that have stored a share for a particular $path$, and $u < n$, then we can no longer guarantee total availability if we lose more than $u - k$ storage servers. The tradeoff is less clear in this case because it depends on which storage servers fail. Suppose that $n = 10, k = 3, u = 9$ for some path p. Then one of the 9 storage servers has two distinct shares. We can then retrieve our file from only two surviving storage servers as long as one of them is the storage server storing two distinct shares. In other words, we can no longer guarantee that we are in a state of total availability if any more than $U - k$ storage servers fail, but we are not certain to be in a state of partial availability at that point, since some servers may still collectively have enough shares to restore the file. Similarly, we may degrade from a state of partial availability to a state of total unavailability if fewer than k servers are running, but cannot say for certain. So, for F_e, allowing storage servers to store more than one share harms the ability of F_e to tolerate server failures without degrading from a state of total availability in general, does not improve the ability of F_e to tolerate server failures without degrading into a state of total unavailability in general, but may allow some individual files to better tolerate server failures, depending on distribution characteristics. Depending on use case, there may be an argument for allowing more than one share on each storage server, but we prefer to make our general availability characteristics as good as they can be, and will favor spreading shares amongst as many storage servers as possible in this work.

Storage utilization assumptions

We also assume that F_e and F_r will not store more data than they produce; in other words, than is implied by their replication or erasure coding parameters. For example, if $r = 4$

in F_r, we are not allowed to store more than 4 replicas of the original input data on the distributed filesystem. In the case of F_e, it means that we are not allowed to store any erasure coded share more than once.

The reasoning for this requirement is less straightforward than that for spreading shares out amongst many storage servers. Note that we can achieve an availability improvement by storing more data. If $r = 4$, and we store five replicas on five distinct servers, then the file can tolerate the loss of four servers instead of five before read availability is compromised. Using selective double placement of shares in an F_e-style filesystem allows us to tolerate the failure of $n - k + 1$ or more storage servers.

This requirement is more for usability and consistency than any clear availability criteria. Space utilization in distributed filesystems is an important issue. Many commodity computing services charge based on the amount of space used [1] [4]. So, in a practical distributed system, it is important for the user to be able to reason about space usage in a precise way. Explicit erasure-coding or replication parameters provided to the user allow the user to do this. We argue that it is not appropriate for an algorithm to second-guess the user's choices, and say instead that the user will increase $n, k,$ or r if they want more data stored on the filesystem.

1.4 Analysis and conclusions

In the previous sections, we've shown how slight elaborations on intuitive distributed filesystems can change their availability and space efficiency properties. In particular, we've seen how two common distributed filesystems trade space-efficiency for stronger guarantees of availability in the presence of failed computer systems. Generally, filesystems that make this tradeoff can operate without degraded availability in the presence of more computer failures than systems that do not make such a tradeoff. We discussed erasure coding, and examined its impact on a distributed filesystem. We saw how the use of erasure coding allows a distributed filesystem to tolerate the failure of far more computers than the use of simple replication when configured to use the same amount of space to use an input file. We concluded by making some of the implicit assumptions in our examples explicit, and

analyzing their implications.

Chapter 2

Practical Constraints to Idealized Distributed Filesystems

In chapter 2, we reasoned about filesystem availability by seeing how tolerant example decentralized filesystems were to the failure of some of their constituent computer systems. Though informative, these conclusions represent an idealized viewpoint of a distributed filesystem. Specifically, they present the operation of placing data at a high level of abstraction. They also assume that there are enough constituent computers to place shares in such a way as to validate assumptions about the availability of files. Problems with this level of abstraction emerge when we wish to consider implementation decisions. In this chapter, we will discuss some problems which motivate our work.

2.1 Too few servers

Implicit in our discussion in chapter 3 is the assumption that there are enough available computer systems for each STORE operation to complete successfully. Consider F_r, and let $\mathsf{r} = 3$. Our reasoning about the availability properties of F_r holds only if there are at least 3 available servers for $\mathsf{STORE}_{\mathsf{F}_\mathsf{r}}$ to write to, since $\mathsf{STORE}_{\mathsf{F}_\mathsf{r}}$ must store each of the 3 replicas on a distinct server for our conclusions about availability to be valid. One of the challenges in designing a practical distributed filesystem is to design protocols that intelligently and consistently handle cases where there are fewer servers than a user might

have expected when configuring the distributed filesystem. Consider a simple example involving F_r. Suppose that $r = 3$, and suppose that F_r has two computers available when $\mathsf{STORE}_{\mathsf{F}_r}$ is invoked. In this case, we cannot give better availability properties than the user would get with $r = 2$.

There are a few ways that STORE can behave in situations like the one above. Most obviously, we may refuse to store the data at all. This makes it easy for the user to reason about the availability of recently-placed files, since the filesystem will never knowingly place files in a way that delivers availability characteristics other than those implied by the user's configuration. On the other hand, it does not do a good job of handling transient failures. If one or two servers are temporarily offline due to a network fault or are unable to accept data due to a disk failure or disk capacity issues, the user may wish the network to place data anyway on the assumption that some automated process will later redistribute the replicas in an optimal way: this may be seen as preferable to having, say, an automated backup process not back up data at all if one or two systems is down temporarily. We can respect the user's desire for graceful, tempered failure by separating the replication parameter (which controls how available the file can be under optimal conditions) from an availability parameter. Then the STORE operation will try to place data in a way that delivers optimal availability characteristics given the configured replication parameters, but won't fail so long as the data can be placed in such a way as to deliver availability characteristics consistent with the availability parameter. In subsequent sections and in the rest of this chapter, we will consider systems with a separate availability parameter.

2.2 Efficient distribution of erasure-coded shares

For replication, it is straightforward to allocate shares across a collection of servers even in suboptimal conditions and with separate availability and replication parameters: we simply choose some number of servers to receive a replica, and check to see whether there are enough servers to meet our goals. If we consider F_e instead of F_r, the problem becomes more complicated.

In Chapter 3, our reasoning about the availability properties of a distributed filesystem

using erasure coding relies on the assumption that no share is allocated to more than one server. Consider, for example, a 5-server distributed filesystem and an erasure code with $k = 3, n = 5$. If we store each of 5 shares on a different server, then the system can tolerate the failure of 2 servers without compromising the availability of the file. If we store shares 1 and 2 on separate servers and then store share 3 on the 3 remaining servers, then the file is unavailable if server 1 or server 2 stop operating; in other words, the filesystem cannot necessarily tolerate the failure of even one randomly selected server.

So, when allocating shares, we must ensure not only that we have enough servers but also that we aren't storing one share on more than one server, or that whatever overlap exists does not diminish the availability properties. We must also continue to do this even if there aren't enough servers available to store every erasure-coded share of the file, or else fail to store the file.

2.3 Dealing with existing shares and repair operations

In large-scale distributed systems, it is generally assumed that there is some amount of churn amongst the computers that make up the system; that is, some computers will fail and leave the system forever, while others will join the system. To account for this, distributed filesystems have a notion of "repairing" a file. We may regard a file as damaged if it is missing replicas or shares, or if those replicas or shares are allocated in an inefficient or unreliable way, or for other reasons. A damaged file is repaired by replacing the parts of it that are missing and/or by moving its replicas or shares around in order to improve availability characteristics.

Generally, a repair operation can be viewed as a special case of a STORE operation; specifically, a STORE operation begun when there are already pieces of a file in the filesystem.

Repairing is a fairly straightforward operation in the case of F_r, since we need only count the number of servers that have a replica of a particular bit pattern and compare that to the replication parameter, uploading new replicas as necessary to bridge the gap. Repairing is more difficult for F_e. We cannot simply count servers, since a group of servers

may have multiple copies of the same share, and may therefore contribute no more to overall availability than one server with that share. Nor can we count shares alone, since a single server having all of the shares does not offer very good availability. One alternative is to count the number of pairwise-disjoint $(\mathtt{server}, \mathtt{share})$ tuples in the filesystem, and, based on that, place new shares as necessary.

Consistent with our focus on storage utilization, we'd also like for storage and repair operations to require a small amount of computation and to avoid placing unnecessary data onto the filesystem. Placing any data onto a distributed filesystem involves various costs, primarily in terms of network traffic, latency, and storage space consumed on the storage servers.

2.4 Summary

When considering F_e, it is apparent that it is difficult to describe, in implementation-level terms, the $\mathsf{STORE}_{\mathsf{F}_e}$ operation while also behaving sensibly in situations in which there are not enough servers, or in which there are already shares in place. Our work is concerned with the evaluation of such algorithms.

Chapter 3

An Introduction to Tahoe-LAFS

In a previous chapter, we discussed at a simplified level the design of idealized types of distributed filesystems. Toward the end, we discussed a distributed filesystem that implements replication through the use of erasure codes, and the impact that the use of erasure codes has on the availability of the filesystem. We will now elaborate on that by presenting Tahoe-LAFS, a distributed filesystem that uses this technique.

3.1 Mojo Nation, Mnet, and Freenet

Tahoe-LAFS can be seen as a simplified version of an earlier product called Mojo Nation [20].

Mojo Nation was the product of a startup from the early 2000s [20]; it was meant to be a general content delivery network, backup system, or other tool for entities who could contribute heterogeneous computers to one network, usually on behalf of their users; an analogy from today is a BitTorrent swarm used to deliver content from a publisher like a film studio or Linux distribution to users who want that content. The enduring novelty of Mojo Nation was the use of electronic currency, called Mojo, to achieve load balancing within the network.

A Mojo Nation network is composed of three types of nodes, provided by users: a block store, which stores data for other clients, a relay, which forwards traffic into the network from nodes that are stuck behind NATs or firewalls, and a content tracker, which serves as

a sort of search engine for the network [19]. These nodes all contribute some amount of disk space, network capacity, or processor power to the grid as a whole. A trusted third party currency server, run by whoever controls the grid, keeps track of these contributions by recognizing them with Mojo. Transactions within the network, such as searching or placing a new file, involve the exchange of Mojo. By creating a market that maps closely to the contributions one makes to the network, Mojo Nation gives users who contribute a lot to the network the ability to demand more of the network than other users. For example, users who have accumulated a lot of Mojo can store more data than other users, search more quickly than other users, and cut to the front of the line for popular content by offering more Mojo to providers of these services than other clients are able to.

As a filesystem, Mojo Nation is very simple. It supports immutable files; that is, files which are not modifiable once uploaded. Files are addressed using an identifier computed from the contents of the file, and are encrypted with a Content Hash Key or CHK, which is an encryption key derived from a hash of the file contents. Files are encoded using standard erasure coding algorithms to replicate them efficiently across the network, and to enable, in conjunction with segmentation of source files, massively parallel or swarming downloads.

When the startup behind Mojo Nation failed, its code was released under the GPL as the Mnet project, which was functionally similar to Mojo Nation except for the lack of micropayments [20]. Development of Mnet has ceased. One employee of the startup went on to create BitTorrent, which uses a simpler subset of the functionality of Mojo Nation (specifically, swarming downloads, content trackers, and an implied digital currency metric that clients generally follow) to great effect in distributing files. Others went on to create Tahoe-LAFS, a simplification of Mojo Nation that has evolved toward a filesystem, though one without digital currency, swarming downloads, or relay servers.

3.2 Tahoe-LAFS

Tahoe-LAFS is essentially a key-value store. Uploading data to a Tahoe-LAFS grid returns a key associated with and derived from the data, which can then be used to retrieve the data later [22]. Tahoe-LAFS is distinguished from distributed hash tables like Chord [15] or

Kademlia [12] in its substitution of a very limited sort of centralization and simple lookup and storage protocols for more complicated algorithms that ensure efficient routing of search and store requests throughout a large decentralized network. It is, in other words, more simplistic than Chord, Kademlia, or other academic distributed hash tables. Simple systems like Tahoe-LAFS are sometimes called NoSQL databases. A NoSQL database, generally speaking, is a data store that supports a very limited notion of querying; typically asking for an object or set of objects by an identifier, and also not supporting or using tables or other relational tools in the same way as traditional SQL databases. NoSQL databases have simple maintenance protocols and are typically designed to tolerate the failure of one or more nodes that comprise them, characteristics shared by Tahoe-LAFS. However, Tahoe-LAFS operates at a lower level than most NoSQL databases, and does not natively support associations beyond associating some arbitrary data to a key. Tahoe-LAFS is most similar to systems like Dynamo [8] and Cassandra [10], though its use of erasure coding, strong cryptography, and capability-based security differentiates it from these projects.

The fundamental entity in a Tahoe-LAFS system is called a grid. A grid is defined by an introducer, a set of storage servers, and a set of clients [22]. When clients first connect to the grid, they communicate with the introducer, which returns to them a list of storage servers. When storage servers first connect to the grid, they communicate with the introducer, which informs all currently connected clients about the storage servers and adds them to its internal list of storage servers. The simplistic nature of the introducer allows the grid to operate without it if it fails; currently connected clients will not learn of any new storage servers, and new clients will not be able to connect, but the existing clients and existing storage servers will be able to communicate normally.

Security is a pervasive characteristic and design goal of Tahoe-LAFS. The most important consequence of this is the notion of provider-independent security, which, roughly, states that the security of data stored within a Tahoe-LAFS grid is not dependent on the security or maliciousness of the storage servers, introducers, or other clients within that grid. This is achieved through the use of strong cryptography on all data stored within the grid as they are uploaded by the client. Specifically, AES-256 is used to encrypt the data, and Merkle trees using SHA-256d as a hash function are used to validate their integrity

after encryption [18]. The overall key used to access stored data is called a capability; that is, it is both an identification and authorization token for a particular resource. In Tahoe-LAFS, capabilities contain the data necessary to locate stored data, verify its integrity, and to decrypt it. Capabilities in Tahoe-LAFS may also be attenuated; for example, a read capability (which delegates to its holder the ability to read the resource to which it refers) may be attenuated to a verify capability, which delegates to its holder the ability to verify the integrity of the ciphertext of the resource to which it refers without giving the authority to read the plaintext of the resource. In general, it is not possible to easily reverse the attenuation of a capability. Capabilities simplify key management by coupling access control and resource identification into one identifier.

In Tahoe-LAFS, a capability for an immutable file looks like:

URI:CHK:key:hash:k:n:size

where `key` is the encryption key used to decrypt the contents of the file, and `hash` is the root of a Merkle tree computed on the ciphertext of the file, and is used to verify the integrity of the ciphertext when it is downloaded. `k` and `n` are erasure coding parameters, discussed below. `size` is the size of the original, unencoded ciphertext.

To account for unreliable storage servers, Tahoe-LAFS replicates data. In most other decentralized storage systems, this means that whole copies of the datum being stored are placed on more than one storage server. The robustness and reliability improvements are clear: so long as one of the storage servers that initially received a copy of the datum remains online and working until the datum is retrieved, the datum is still available. One downside to replication is that the amount of space required to store a particular datum increases in direct proportion to the number of replicas that are made of the datum. In a direct replica system, to tolerate the failure of any n storage servers, it is necessary to store complete replicas of the data on at least $n + 1$ storage servers. Tahoe-LAFS addresses this cost somewhat by the use of erasure coding [22]. Tahoe-LAFS uses zfec, an efficient and fast implementation of Reed-Solomon codes [21]. In zfec, all n pieces are the same size, which is $\frac{N}{k}$ bits if the source data is N bits. As in other erasure codes, the combination of particular n and k are often called a k-of-n encoding.

Tahoe-LAFS is a good test platform for our experiments because it is designed to tolerate misbehaving or adversarial storage nodes and grid participants. At the same time, it lacks a coherent approach to addressing the sort of reliability issues that we wish to examine. Finally, it is free, open-source software with which the author is familiar.

Chapter 4

Share Allocation Strategies in Tahoe-LAFS

We will now discuss two share allocation strategies used at various points in Tahoe-LAFS, described in chapter 4. We will give a high-level overview of each strategy, then discuss algorithms to verify the availability characteristics implied by each strategy and to place shares such that the availability characteristics implied by each strategy are satisfied. These algorithms will guide our implementation when we simulate the performance of the share allocation techniques in subsequent chapters. Readers interested in viewing the source code implementation of these algorithms are encouraged to consult the appendix.

4.1 How Tahoe-LAFS controls for availability

Tahoe-LAFS follows the model described in previous chapters, in that it decouples the ideal replication state of a file, as implied by the n and k encoding parameters, from the minimum replication required in order for a file storage operation to be successful, which we will refer to as h [13]. The user may configure n, k, and h to satisfy their own availability requirements. In various versions of Tahoe-LAFS, h has meant different things. In this chapter, the abstractions and algorithms we discuss will essentially be defining h, as it was defined in various versions of Tahoe-LAFS.

4.2 Shares of happiness

Initially, ħ was a shorthand for "happiness", which in turn referred to the idea of shares of happiness [3]. In the shares-of-happiness file availability test, the file is regarded as acceptably available ("happy") if at least ħ of its erasure-coded shares are recoverable from the distributed filesystem; otherwise, the file is not sufficiently available (i.e., it is "unhappy") and the upload operation is not successful.

As discussed earlier, a test for shares of happiness is simple: we must determine how many shares were placed on computers within the distributed filesystem, then compare that number with a preconfigured threshold. If the number is smaller than the threshold, then availability will be unacceptably low and the file upload operation should be aborted. In terms of implementation, we can generate a set of shares that we expect to produce, then remove elements from that set until we either run out of cooperating computer systems (e.g., because some have failed or are full) or until all shares are placed. The size of this set at the end of share placement is what we must compare with the preconfigured threshold.

Shares of happiness uses a simple, phase-based share placement algorithm. An uploader keeps track of shares that it has to place, shares that it has successfully placed and their locations, and servers that may accept shares, sorted according to a file-specific permutation key. An uploader also distinguishes between unused storage servers, which have not yet been asked to store a share by the uploader, and already-used storage servers, which have accepted a share when asked to do so by the uploader.

Share placement occurs as a loop, generally controlled by the number of shares yet to be placed. At each iteration, an unallocated share is selected for storage on some storage server. Next, a storage server is selected. If there are unused storage servers, then an unused storage server is selected; otherwise, a previously-used storage server is used. A message is sent to the storage server directing it to allocate space for the share. If the request is successful, then the share is removed from the list of unallocated shares and the server is stored in a list of servers with allocated shares. If the request does not result in an error but is not accepted (likely due to insufficient storage space on the server), the server is removed from further consideration and the share is left in the list of unallocated shares. If the

result does result in an error (likely due to some unexpected problem or bug on the storage server), then the share is left in the list of unallocated shares, the server is removed from further consideration, and any shares previously allocated to the server are added to the list of unallocated shares, since the protocol assumes that the server has failed and cannot be trusted to honor previously accepted storage requests. Along with an answer to the share allocation request, each storage server will return a set of the shares that it already holds. If this set is nonempty, then the uploader will remove the shares in the set from the set of shares that it needs to place. This is of benefit during file repair operations, in which some number of shares already exist on the distributed filesystem; specifically, it minimizes the number of shares that the uploader must place in order to complete the repair operation.

As mentioned, the shares of happiness process operates in phases. During the first phase, the uploader attempts to store each share that it chooses on an unused storage server. The first phase ends when there are no more shares to place, in which case the upload is declared successful, or when there are no more unused storage servers to place shares on, in which case the second phase begins. The second phase attempts to place shares one-at-a-time onto storage servers, but uses used storage servers instead of unused storage servers. The second phase, like the first phase, ends either when there are no more shares to place, in which case the upload process stops, or when there are no more storage servers. In the latter case, the protocol enters the third phase, during which shares are allocated proportionally amongst all of the used storage servers that are still accepting shares. For example, if there are four shares remaining in phase 3, and 2 storage servers available to accept shares, then each step of the loop will attempt to place two shares on each storage server. The third phase continues until all shares have been placed or until there are no more servers available to accept shares; in either case, the end of the third phase is also the end of the share placement process. When the share placement process concludes, the shares of happiness algorithm is executed. If the share allocation is acceptable, then the upload is successful, otherwise, it is unsuccessful.

4.3 Servers of happiness

In current implementations of Tahoe-LAFS, a slightly more complicated availability test called servers-of-happiness is used [3]. Servers of happiness was initially motivated by shortcomings in the shares of happiness test. The name servers of happiness is due to Zooko, one of the developers of Tahoe-LAFS. The idea to use a bipartite graph as the health test for servers of happiness is due to David-Sarah Hopwood, another Tahoe-LAFS developer [7]. We will examine the servers of happiness test, and develop a share allocation algorithm that improves upon shares of happiness by using the bipartite graph idea to motivate share placements.

The shares of happiness availability test is not able to distinguish between a share allocation in which all n shares are stored on a single storage server and a share allocation in which each of the n shares are placed on a different server, even though these two allocations have completely different availability characteristics. This shortcoming is due to the simplistic design of shares of happiness; specifically, that it is concerned only with the number of shares that are placed. In our reasoning about availability characteristics in an earlier chapter, we presented availability characteristics primarily defined by the number of servers holding shares of a file. We reasoned about the number of servers that must fail before a filesystem can no longer be guaranteed to be totally available, and the number of servers that must be available in order to guarantee partial functionality. For practical reasons, we must also consider the number of distinct shares generated from a particular file that are available on the filesystem. To see why this is, consider a file with encoding parameters $k = 3, n = 10$. Note that the file is unrecoverable if one of its shares is replicated once on each of 10 storage servers, but that the same file is recoverable if each of its 10 shares is replicated once on its own storage server. The shares of happiness share placement algorithm is designed to place each share on a different storage server, yielding a share allocation in which the number of storage servers is also the number of available shares. In this case, counting the number of distinct shares available is sufficient as an availability test, since it implicitly considers the number of servers when it considers the number of shares. The problem with shares of happiness becomes apparent when the shares

of happiness share allocation algorithm fails to achieve a share allocation whose availability characteristics allow the number of shares to model the number of servers, since, as above, it cannot distinguish between situations in which the availability characteristics are acceptable and situations in which the availability of the file depends entirely on the functioning of one server. Intuitively, our solution to this problem is to explicitly consider the number of servers alongside with the number of shares.

Instead of shares, servers-of-happiness deals with the size of a bipartite matching between servers and shares. The idea of using a bipartite This has many attractive properties.

First, and most relevant to the shortcomings of the shares of happiness technique, servers of happiness takes the number of servers storing shares of a file explicitly into account when determining its availability characteristics. If $\mathtt{h} = 7$, then the file cannot be deemed acceptably available unless its pieces are stored on at least 7 distinct storage servers. Using a bipartite matching also ensures that the availability measured by the uploader is not affected by servers storing more than one share each, since only one share on each storage server is counted toward the matching. Additionally, the bipartite matching will not overcount shares stored on multiple storage servers. To see why this is important, consider a file encoded with $\mathtt{k} = 3, \mathtt{n} = 4$, and stored such that pieces 1 and 2 are each stored on one storage server and piece 3 is stored on two storage servers. Note that this layout can only tolerate the loss of one of the two servers storing piece 3 without losing read availability, which is a weaker availability guarantee than if each of the four storage servers stored a unique piece of the file (in which case we could tolerate the failure of any single storage server without losing read availability).

At its core, servers of happiness uses a bipartite graph to determine whether a proposed placement of the erasure-coded shares of an input file would yield suitable availability characteristics. Specifically, it uses a bipartite matching constructed in a bipartite graph induced by the share placement and constructed as follows:

- Let $G = (Sh \cup Sr, E)$ be a bipartite graph
- Let Sh be the set of all erasure-coded shares generated by a storage operation.
- Let Sr be the set of all servers that may accept shares for a storage operation.

- Let $\mathsf{E} = \{(\mathsf{sr}, \mathsf{sh}) : \text{ server } \mathsf{sr} \text{ stores store a copy of } \mathsf{sh} \}$

The servers of happiness configuration parameter relates to the size of a matching found in this bipartite graph. If a matching of at least h edges can be found in G, then the share placement has suitable availability characteristics; otherwise, the availability characteristics are unacceptable and the storage operation must not succeed.

The above is very nearly sufficient as a test for servers-of-happiness. The only modification we need to perform is to add an edge to E for each server that has agreed in a peer selection phase to store a particular erasure-coded share. This amends servers-of-happiness so that it tells us if the storage operation, once completed, will have suitable availability characteristics, which is exactly what we want a servers-of-happiness test to do. Intuitively, we would expect a peer selection mechanism to keep track of the shares stored by a particular server (or, equivalently, which servers store which share), an expectation which motivates our implementation. Such an implementation could consist of an object that takes in such an association, constructs a graph from the association, as described above, and then computes, using an appropriate algorithm like Edmonds-Karp, a maximum matching in the graph, returning the cardinality of the matching as a result. This remains independent of most of the mechanics of peer selection (e.g., the process of communicating over the network with storage server), allowing the peer selection phase and availability test phase to be two conceptually different parts of the program, which eases maintenance.

When first implemented, this is how servers-of-happiness was integrated into Tahoe-LAFS. The existing multi-phase share placement algorithm was left intact, except for modifications to create and maintain the necessary data structures to allow the bipartite graph to be constructed. After peer selection and share allocation, but before the uploader actually started uploading data, the test described above is performed. If the share placement yields acceptable availability properties, then the storage operation is allowed to continue; otherwise, the storage operation fails. Though this had advantages in terms of implementation (in particular, it made the change much less invasive than it could have been), it comes at the cost of a fairly significant mismatch between the share placement algorithm and the share placement availability test. Specifically, the share placement algorithm was

not designed to satisfy the servers-of-happiness upload availability test. In many situations, the existing share placement algorithm would do a good job satisfying the servers of happiness test, but there are many cases in which it doesn't. For example, consider the case of a share placement operation that comes upon a server that already holds some or all of the shares that need placing. This may happen in storage operations associated with a high-level repair, in which the file to be stored has already been stored on the filesystem. The existing share placement algorithm would mark these shares as placed. This is acceptable from the standpoint of the simpler shares of happiness share placement test, but does not do a good job of satisfying the servers of happiness test, since only one of the shares on the single storage server can contribute an edge to the matching and since the existing placement algorithm will not attempt to place any of the other shares elsewhere. Instead, we'd like the share allocation algorithm to attempt to spread the shares around to other storage servers, improving availability characteristics. These sorts of interactions show some of the benefits to a share placement strategy that is conceptually integrated with the availability test performed upon the share placement.

It is straightforward to modify the servers of happiness test to also determine share placements. First, note that an edge $(\mathtt{sh}, \mathtt{sr})$ in the bipartite matching is essentially a share assignment, in that it can be interpreted to say "store share $\mathtt{sh}$ on server $\mathtt{sr}$". This suggests that we can get more information about where to store things by going beyond a simple scalar description of the matching by also considering its constituent edges. We can modify the shares of happiness test to place shares using this observation by adding another set of edges to the edge set of the bipartite graph describe above. Specifically, we want to add an edge for each possible share placement. That is, we want to draw an edge from each share to each server that could possibly store that share. If we compute the matching in the same way as before, the edges of the matching will be the share placements that we should use if we want to get the maximum availability score, as determined by the servers of happiness availability test, and the number of assignments that we have is how available our resulting allocation will be.

More formally, we construct the bipartite graph G as follows.

- Let $G = (Sr \cup Sh, E)$, where
- Sr is the set of storage servers.
- Sh is the set of shares to be placed.
- $E = \{(sr, sh) : sr \in Sr, sh \in Sh,\ sr \text{ stores } sh\ \} \cup \{(sr, sh) : sr \in Sr, sh \in Sh,\ sr \text{ could store } sh\}$

To allocate our erasure-coded shares, we compute a maximum bipartite matching in G; the edges in the matching are our share placements, and the number of edges dictates the availability of our file according to the servers of happiness availability test. If there aren't at least h edges, we conclude that the file does not have satisfactory availability characteristics and refuse to complete the storage operation.

This behaves acceptably in scenarios where there are at least as many servers accepting shares as there are shares to place. In situations where there are fewer servers than shares to place, this has a very significant downside: it will not place all of the shares. This results from the fact that the size of a maximum matching in a bipartite graph is bounded by the size of the smaller of the two vertex sets that make up the graph. So, in a situation where the size of the server vertex set is smaller than the size of the share vertex set, the size of the matching is bounded by the number of servers available to store shares. This is addressed fairly simply, however: we simply store the unallocated shares onto servers which have been allocated a share by the bipartite matching, attempting to spread them out evenly according to the individual matching. For example, in a situation in which there were 10 erasure-coded shares but only 5 storage servers, we would place 2 shares on each of the 5 storage servers.

Another weakness of this formulation is its inability to distinguish between shares that already exist on the filesystem and shares that have not yet been placed on the filesystem. To conserve network bandwidth, we want preexisting shares to compose as much of our matching as possible, since preexisting shares don't need to be uploaded during the storage operation. We do not, however, want to have a share allocation that is less available than it could be if network bandwidth was unimportant; in other words, we must re-use

preexisting shares subject to the constraint that we may not produce a smaller matching than an algorithm that does not distinguish between preexisting shares and shares that must be uploaded. We can do this by splitting the share allocation operation into two phases: a matching on the subgraph of G formed by the edges associated with currently existing share placements, followed by a matching on the graph G without the edges and vertices used in the first matching. The union of these two matchings is the share placement, and it has the property that it is the same size as a maximum matching in G, and that it re-uses as many pre-existing share placements as is possible while still being a maximum matching.

We've now evolved an almost-complete share allocation algorithm based on the servers of happiness availability measurement. We have not yet specified, however, how to handle share placements that are rejected by storage servers. We say that a share allocation has been rejected if the storage server refuses to store the share that it was allocated. This may happen because the storage server is full, because the storage server is broken, or for a variety of other reasons. Since we purport to deal with actual, realistic systems, this is a case that we should handle.

Intuitively, we would need to recalculate the share allocation if some part of the share allocation was rejected by a storage server. To avoid cases where the exact same share allocation is retried, we must also modify the bipartite graph slightly before recalculating the share allocation. The most obvious way to do this is to remove any server that rejected its allocation from the set of servers (and delete any associated edges). This ignores information that the storage server gives us when rejecting our storage request, however. We can split the idea of rejection into two distinct actions: rejection because of insufficient space, and rejection due to an error. We may assume that servers which reject shares because they don't have enough space to store them can still serve shares that they already store. We should not make the same assumption about servers that reject share allocations due to an error, since the error condition may render them unable to serve shares that they have already stored. These distinctions help show how we can alter the bipartite graph to reflect rejection when it happens. In the case where a storage server is full, we shouldn't remove the storage server entirely from the graph; instead, we should remove any edges added to reflect the possibility of storing new shares on the storage server, leaving the edges for

shares already stored intact. This increases the likelihood that the server can contribute meaningfully to availability measurements for the storage operation. Only in the case of an error should a server be removed entirely from the graph. Note that this has a useful invariant for guaranteeing eventual termination: in every case in which the share allocation is recalculated, at least one edge is removed from the graph.

There are possible efficiency considerations, though. We expect, in cases where a storage allocation request is rejected, that most of the other allocation requests are successful. To avoid having to make redundant allocation requests in subsequent iterations of the protocol, we would like for the share allocation strategy to re-use shares that have been allocated but not yet stored as pre-existing shares are. We can do this through another application of the subgraph idea that re-uses preexisting shares. Specifically, after the matching phase for preexisting shares, we add another matching phase on the subgraph of G formed by taking servers, shares, and edges that represent shares allocated in previous iterations of the allocation procedure. As before, we remove from the graph considered by subsequent phases any edges, servers, and shares used in the matching for allocated shares. As before, the final matching is the union of the three matchings made in phases of the protocol, and, as before, this is a maximum matching in the graph. Then, subject to the constraint that we first want to maximize the number of preexisting shares used and the constraint that we want a maximum matching, we minimize our redundant storage requests by adding a separate phase for shares already allocated.

Putting it all together, we have the following algorithm outline:

- Poll storage servers for their shares. Register each share with the servers of happiness share allocator object.
- Determine which servers are able to accept new shares. Register these servers with the servers of happiness allocator object.
- Using the iterated bipartite graph matching technique described above, generate a matching and return the edges of that matching as share allocations.

- Attempt to allocate shares on servers according to the matching. If some allocations fail, register these failures with the allocator object and regenerate the matching. Register successful allocations with the allocator. Repeat this process until there are no errors or until the matching falls below the configured availability threshold.

We encourage the reader to consult the source code of our implementation in the appendix for a more accurate and thorough treatment of our algorithm.

By using the servers of happiness share placement availability test to motivate share placement, this addresses many edge cases associated with using servers of happiness as a test only. It also makes efficient re-use of shares that already exist, and handles failures in an intelligent way.

4.4 Discussion

The main advantage of shares of happiness over servers of happiness is that it is much simpler. This makes it easier for users without specialized knowledge to understand the happiness parameter and to reason about what it should be set to, given availability expectations and grid characteristics. It also means that it has fewer side effects, and side effects that are easier to understand. This simplicity contributes to the main disadvantage of shares of happiness, however, since it is not powerful enough to give us the availability guarantees we want. Servers of happiness is more complicated, requiring at least an acquaintance with elementary graph theory to understand fully. This means that it is much harder for a user without specialized training to understand the parameter and determine an appropriate setting. On the other hand, it has sufficient power to give us a good idea of the availability of files once stored on the distributed filesystem.

Chapter 5

A Comparison of Share Placement Algorithms

In the previous chapter, we introduced two availability measurements used in Tahoe-LAFS to evaluate the placement of erasure-coded shares in a distributed filesystem. We identified the shortcomings of the shares of happiness test, and showed how the more complicated servers of happiness test addresses those shortcomings. Anecdotally, users and developers are generally happy with how the servers of happiness test works relative to the shares of happiness test. In this chapter, we will present a simulation that we will use to evaluate variants of these two tests, measuring availability and bandwidth usage over time. We will compare servers of happiness and shares of happiness both to each other and to a control algorithm. We will discuss the results of this simulation. Our results show that servers of happiness offers significant improvements over shares of happiness in both file availability and efficiency. We will briefly define what we mean by file availability and efficiency, and then discuss the results of the simulation.

5.1 Design of the simulation

We will evaluate the two file share allocation strategies from the perspective of a client storing some amount of data on a filesystem. The filesystem will have a fixed number $\mathfrak{n}$ of servers that fail with probability $\mathfrak{p}$ at each time step of the simulation. At the initial

time step, the client will store its data (one or more files) on the filesystem according to the share placement algorithm and availability test being used at the time. On subsequent time steps, execution will proceed as follows:

- Each server will be prompted to fail. Each server will fail with probability `p`.
- The client will assess the health of each of the files it has stored on the filesystem. There are three possible answers:
  - Healthy: shares may be missing, but the file's availability characteristics are still consistent with the configured health parameter.
  - Unhealthy: the file's availability characteristics are no longer consistent with the configured health paramater, but the file is still recoverable.
  - Unreadable: there are not enough shares left to recover the file.

  The results of these assessments will be recorded.
- The client will repair any unhealthy files, if possible.
- Previously failed servers will be restored with probability `q`. If restored, previously failed servers return to the grid without any shares that they held at the time that they failed.

After some number of time steps, the simulation will finish. We will be able to compare the algorithms based on a few measurements collected throughout the simulation.

- Total number of shares uploaded (which models resource utilization)
- Total number of unhealthy files found
- Total number of files lost by the end of the simulation

5.2 Implementation of the simulation

We expect to divide the simulation into two types of objects: client objects and server objects.

A client object keeps track of events observed over the course of the simulation. It keeps track of as much state as is necessary to evaluate the health of files and, when acceptable, to restore them. It has a method that will cause it to assess the availability of its files, a method that will cause it to attempt to repair files, and a method that will cause it to place files.

A server object is composed primarily of file storage state. It maps file identifiers to the shares stored for that file. It also has a method that causes the server to fail with probability p, methods that allow a client to get and set storage state if the server has not failed, and a method to possibly restart the server.

An object representing the filesystem is responsible for aggregating the server objects and exposing them to the client object.

Client objects contain references to share allocation and file evaluation objects, which are the meat of the simulation. We expect to have a share allocation object for shares of happiness, a share allocation object for servers of happiness, and a random share allocation object to act as a control. The first two of these will contain simplified code snippets from Tahoe-LAFS, following the algorithms described in the previous chapter.

The interaction between these objects is managed by a coordinator object. At each time step, the coordinator object will make the servers fail, initiate file assessment and repair, and make some servers go up again. The coordinator object will also gather statistics from the clients as the simulation progresses, and then report those to the user.

5.3 The Control Algorithm

To determine how effective shares of happiness and servers of happiness are in general, we will compare them with a random peer selection algorithm. The random peer selection algorithm randomly chooses a destination server for each erasure coded share, and does not have a specific availability test.

5.4 Criteria for evaluating the simulation

We consider two criteria when evaluating the results of our simulation. One criterion is file availability. As mentioned, we distinguish between three states of file availability inside the simulation: healthy, unhealthy, and unrecoverable. Of these, we care primarily about whether a file is recoverable or not, since the health of a file is primarily an internal indicator of whether a file needs repair. The other criterion is efficiency, which we will evaluate by considering how many shares need to be placed over the course of the simulation to preserve the availability of a file. In an actual distributed filesystem, there are many more efficiency considerations than share placement, but the number of shares placed is a useful proxy for many of these. Storage, bandwidth usage, and CPU usage are all closely correlated to the number of shares generated, for example.

We wish to maximize the availability of files, and we wish to minimize the number of shares placed over the course of the simulation.

5.5 Simulation parameters

We conduct our simulation using the framework described above. We run the simulation for 5000 timesteps, and simulate the results for 100 files stored on filesystems of 6, 8, 10, and 12 servers. Our encoding parameters are $k = 3, n = 10$. The happy parameter, described earlier, is set to 7. Storage servers fail at each timestep with probability .075. Failed storage servers recover at each timestep with probability .3. For each simulation, we keep track of the number of shares placed by each client, and the number of files associated with each client that are recoverable. We will use these data to assess the various share placement algorithms in the next section. By varying the number of servers, we may also draw conclusions about how the encoding parameters k and n should relate to the number of servers in order to ensure good reliability. We also run a second simulation with these same parameters on grids of 10 and 12 servers, but in which storage servers will not store more than 280 shares each. We use this to test the behavior of each algorithm when allocations fail due to storage capacity issues.

5.6 Results and Discussion

Availability

In general, servers of happiness yields better availability outcomes than either of the other algorithms. In both unbounded and unbounded capacity tests, and for all server quantities tested, the servers of happiness algorithm manages to preserve more files for longer than the other two algorithms.

First, we consider the results for the 6 server test, shown in figure 5.1.

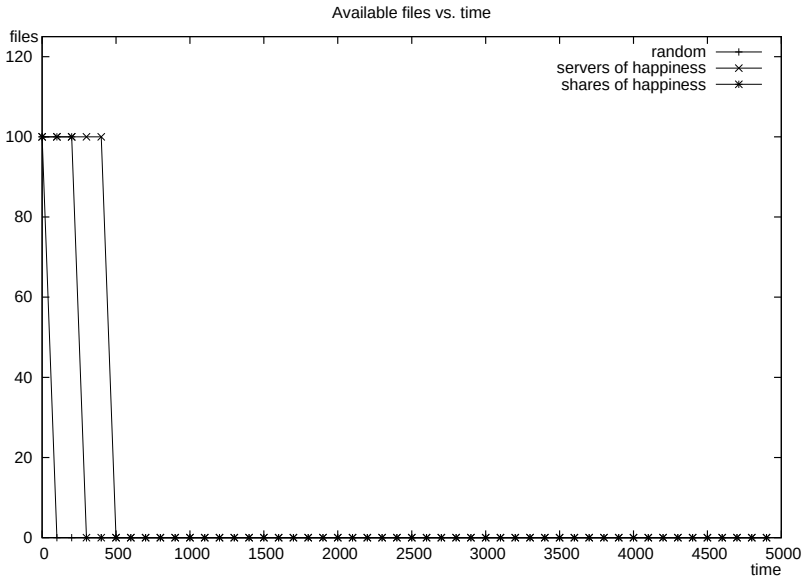

Figure 5.1: Availability versus time for a 6 server simulation

None of the algorithms performs particularly well with 6 servers. The random algorithm – the leftmost line in the graph – loses all of its shares almost immediately. By 300 timesteps, the shares of happiness algorithm has lost all of its shares, and by 500 timesteps, the servers of happiness algorithm follows suit. This, we expect, is due to the small number of servers participating in the simulation. Intuitively, files are more resilient to the failure of one or a few storage servers if they are spread out amongst a large number of storage servers, since in that case one or a few storage servers is not a majority of available storage servers. Our availability results for 8, 10, and 12 shares demonstrate this experimentally.

As mentioned, servers of happiness outperforms both the random algorithm and the

shares of happiness algorithm. In a filesystem with only 6 storage servers and in which $h = 7$, servers of happiness will never regard a share allocation as healthy, so it will always repair files when given the opportunity to do so. As long as shares of happiness places at least 7 shares, it can regard share allocations in a filesystem with only 6 servers as healthy, which means that it may not take advantage of the opportunity to repair files. This gives an advantage to servers of happiness, which is in a better position to place shares on newly activated servers and maintain a share placement that is resilient to the failure of storage servers. That said, neither algorithm offers particularly inspiring performance.

The 8 server test, shown in figure 5.2, yields slightly more interesting results.

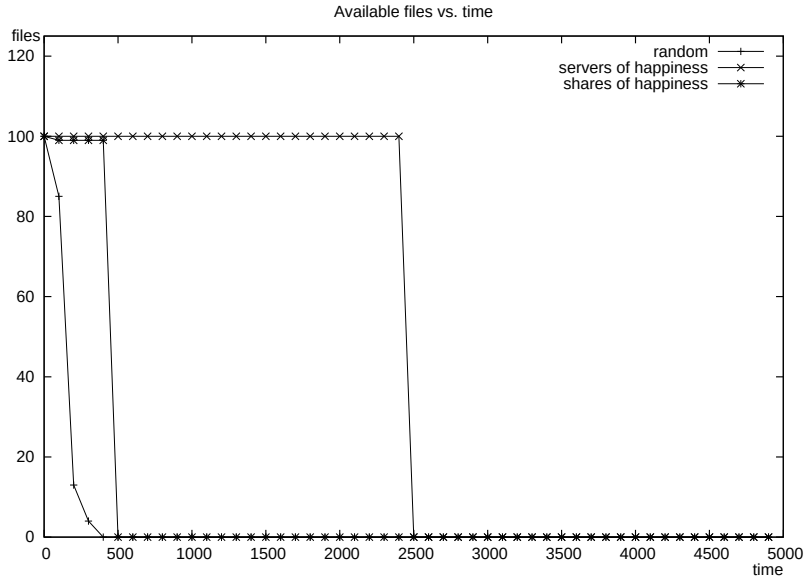

Figure 5.2: Availability versus time for an 8 server simulation

As before, the random algorithm and the shares of happiness algorithm lose their files relatively quickly; the former before 500 timesteps, the latter at 500 timesteps. The servers of happiness algorithm manages to hold on to its files for much longer, however, retaining all of them up until 2500 timesteps. We suspect that this performance is due largely to the same factors as those identified above; servers of happiness is designed to detect and react to situations in which there are too few servers to satisfy availability requirements, while shares of happiness is not designed to do this. That servers of happiness retains its shares for much longer than in the 6 server simulation (and for much longer than the other

algorithms in both simulations) is likely due to the fact that 8 servers is, given our encoding and health parameters, a small enough number of servers to pose availability challenges, but a large enough number to allow servers of happiness or an algorithm designed to take availability into account to preserve its files.

The 10 and 12 server simulations, shown in figures 5.3 and 5.4, continue this trend.

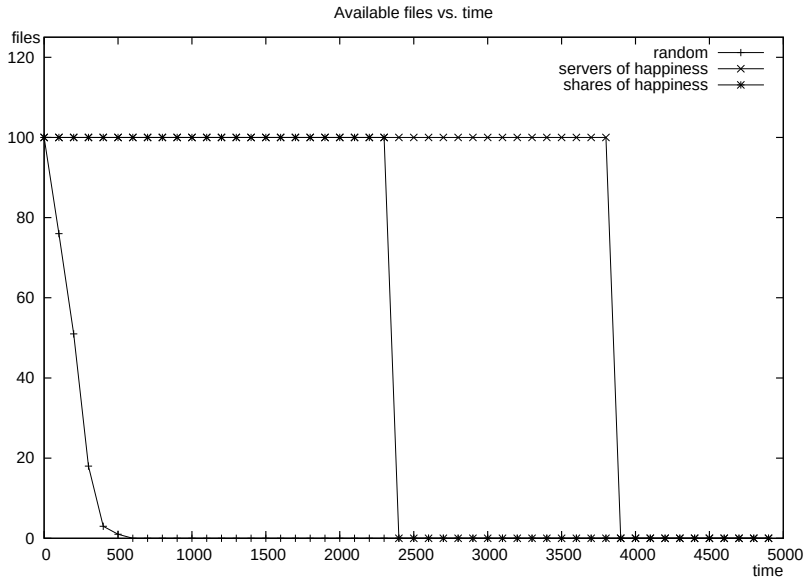

Figure 5.3: Availability versus time for a 10 server simulation

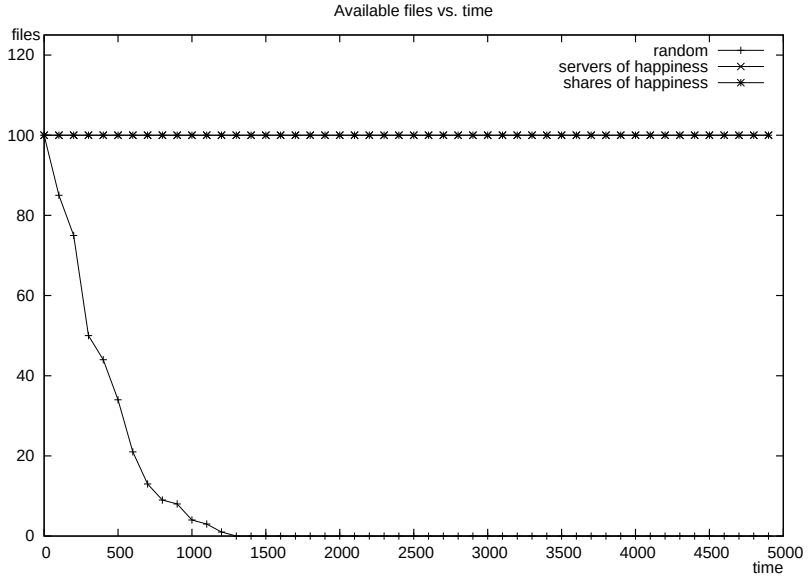

Figure 5.4: Availability versus time for a 12 server simulation

As before, the random algorithm loses its files earlier than the other algorithms; in figure 5.3, the random algorithm loses all of its files by slightly after 500 timesteps, and in figure 5.4 all files placed by the random algorithm are lost by 1500 timesteps. In each simulation, the shares of happiness algorithm outperforms the random algorithm, retaining files until 2500 timesteps in figure 5.3 and until the end of the simulation in figure 5.4. As in figures 5.1 and 5.2, the servers of happiness algorithm outperforms the shares of happiness algorithm in figure 5.3, and retains all of its files until the end of the test in 5.4, delivering availability on par with shares of happiness. Figure 5.3 is consistent with the progression seen in figures 5.2 and 5.1; the servers of happiness retains files for longer than the shares of happiness algorithm, which retains files for longer than the random algorithm in each case, but the length of time

The availability results change slightly when we impose an upper bound on the number of shares that storage servers are allowed to store. In this case, it is important not only to maximize availability but also to efficiently utilize space on the filesystem, since placing too many shares while repairing can make it harder to do repair operations in the future. Figures 5.5 and 5.6 show the results for these tests.

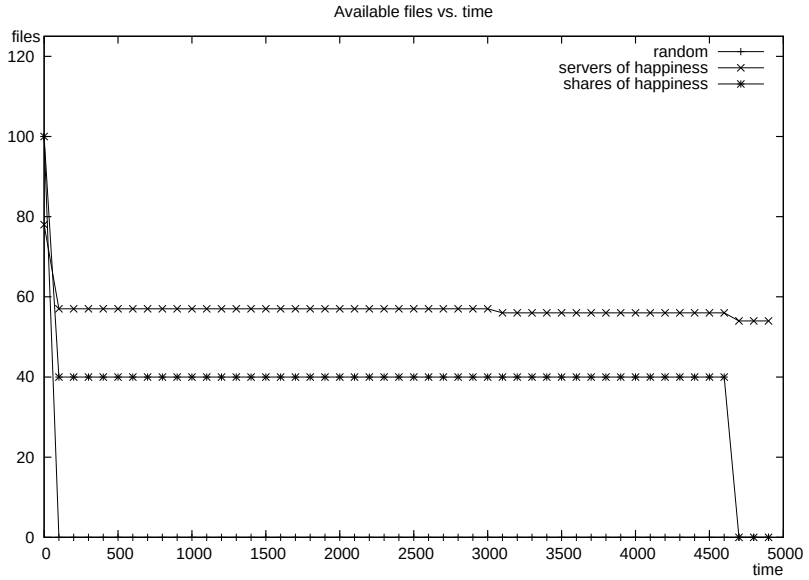

Figure 5.5: Availability versus time for a 10 server simulation with 280 share capacity

These results are generally consistent with figures 5.1, 5.2, 5.3 and 5.4, in that the

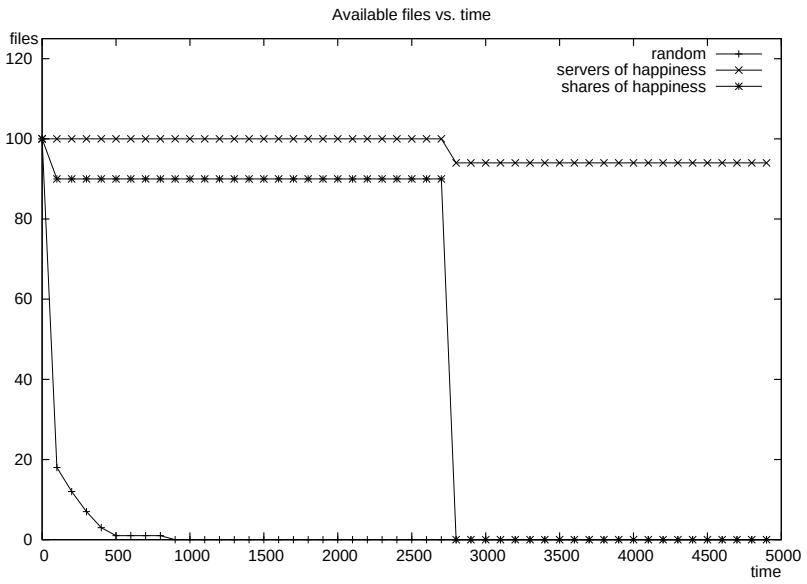

Figure 5.6: Availability versus time for a 12 server simulation with 280 share capacity

random algorithm loses its files before the other two algorithms (and relatively close to the beginning of the simulation), the shares of happiness algorithm loses its files before the servers of happiness algorithm, and the servers of happiness algorithm retains more files for as long or longer than the other two algorithms. The primary difference associated with storage capacity limits is a change to the way the servers of happiness and shares of happiness lose their files. In simulations in which no bound is placed on the storage capacity of storage servers, the servers of happiness and shares of happiness algorithms tend to either retain all of their files or retain none of their files, instead of losing some files gradually over the course of the simulation. In figures 5.5 and 5.6, we see the first partial file losses for servers of happiness and shares of happiness. In figure 5.5, both algorithms lose some shares at the start of the simulation, then continue on with most or all of the remaining shares until either the remaining shares are also lost or until the simulation ends. Figure 5.6 shows similar data. This is as expected. If servers have an upper bound on the number of shares that they will store, then not all share allocation requests to a normally functioning storage server will succeed, since the storage server may be too full to accept new shares. This can result in repair operations that fail, or that achieve file availability characteristics that aren't as good as could be had if all storage requests were accepted.

In all simulations, the availability graphs of shares of happiness and servers of happiness are similar in shape. In the unbounded simulation, they both start as a flat line, followed by a very steep drop, followed by another flat segment, corresponding to small number of timesteps in which all files are lost. The same general shape is apparent in the bounded simulation as well, though in these simulations there are more gradual file availability losses at points.

We have not developed a satisfactory explanation for the dramatic slope of the servers of happiness and shares of happiness graphs. We suspect that these events correspond to a timestep in which a very large number of storage servers are offline at once. Both algorithms will react to these situations by placing all of the shares on the remaining storage servers. A weakness of shares of happiness is its ignorance of the topology of the storage servers as it relates to the need for shares to be distributed across storage servers. In this situation, this problem presents itself when storage servers that had previously failed rejoin the network. Intuitively, we would like for the shares that had been previously allocated to the small number of storage servers remaining after a failure to be spread out across the new storage servers. However, as long as the storage servers holding large numbers of shares remain operational, shares of happiness will not attempt to spread the shares out. Then, later, if all or most of those storage servers fail, shares of happiness can be left with a large number of files that are unrecoverable. In summary, a steeply-sloped line is not surprising for shares of happiness. The steeply sloped line for servers of happiness is somewhat surprising, since it takes advantage of newly-created storage servers to spread shares around. It is possible that the failures of servers of happiness correspond to situations in which more than $n-k+1$ storage servers fail at once, in which case a well-distributed file would be unrecoverable. More study and instrumentation of the simulation is necessary to understand this behavior.

In general, servers of happiness yields better availability outcomes than either the random algorithm or the shares of happiness algorithm. It retains files for longer than either of the other two algorithms, and retains more of them than either of the other two algorithms. Availability outcomes improve across the board as the number of servers increases, and are generally better in tests with unbounded storage server capacity than in tests with an upper bound on storage server capacity.

Efficiency

As mentioned, we evaluate the efficiency of share placement algorithms by keeping track of how many shares each algorithm places over the course of the simulation. We wish to minimize this quantity; in other words, we want a share placement algorithm that places as few shares as possible, subject to the constraint that the algorithm also has acceptable availability characteristics. Since the number of shares placed correlates to storage space utilization, network bandwidth, and CPU time, minimizing the number of shares placed has desirable implications for an actual distributed filesystem.

To evaluate algorithms, we will view graphs of the number of shares placed over time. On each graph, we will plot a line for each of the random, shares of happiness, and servers of happiness share placement algorithms. There are two attributes of each line that we care about. The maximum y-axis value of a line is the number of shares placed over the course of the simulation by the algorithm corresponding to that line. The slope of the line is a way to understand how many shares the algorithm tends to place per timestep. A steeply-sloped line corresponds to an algorithm that places a large number of shares per timestep; a more gradual slope corresponds to an algorithm that places a smaller number of shares per timestep. To effectively meet our requirements above, then, we should favor an algorithm with a relatively gradual slope and a small number of shares placed over the course of the simulation. These requirements give preference to algorithms that have lost all of their files, however. If an algorithm has no recoverable files, then it cannot perform any repair operations, will not place any shares, and will have a slope of zero and will not place any more shares over the course of the simulation. If such an algorithm loses all of its files early in the simulation, it can look very appealing, since it will place far fewer total shares than algorithms that preserve file availability until the end of the simulation. To avoid this, we must consider whether and where the slope of the line associated with an algorithm goes to zero over the course of the simulation.

We first consider figure 5.7, a graph showing the share placement rates of the three algorithms during the 6 server simulation.

Recall from figure 5.1 that no algorithm retained files for more than 500 timesteps in the

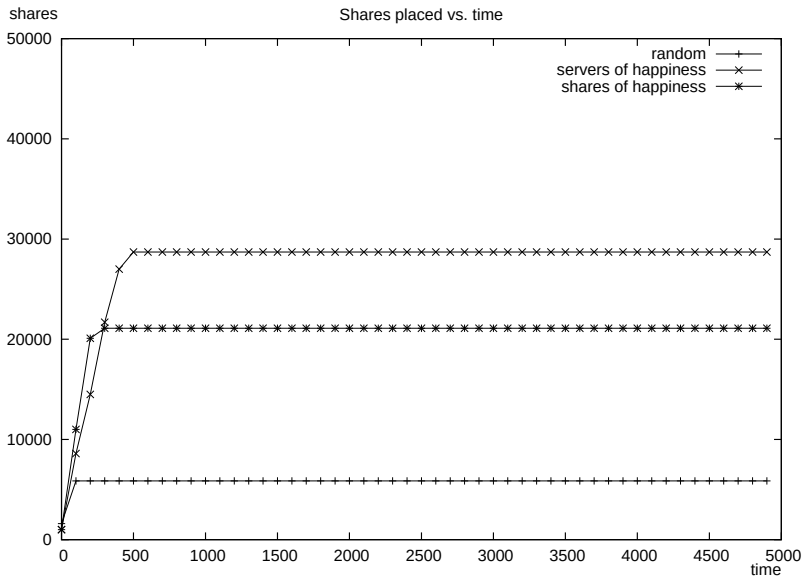

Figure 5.7: Shares placed versus time for a 6 server simulation

6 server simulation. As expected, then, all of the lines in figure 5.7 level off quickly, by the end of the 500th timestep. The random algorithm stops placing new shares by the 100th timestep, but has the most gradual slope of the three algorithms, suggesting either that it lost a large number of files before the other algorithms or that it does not place as many shares as the other two algorithms when functioning normally. The servers of happiness algorithm places 50% more shares than the shares of happiness algorithm over the course of the simulation, but also retains its files for longer than the shares of happiness algorithm. The servers of happiness algorithm places shares at a slower rate than the shares of happiness algorithm. Overall, considering our desire to minimize resource utilization subject to the requirement that we retain files for as long as we can, and considering the points at which the algorithms lose all of their files and stop placing new shares, the servers of happiness algorithm produces a better availability outcome than the others in the case of 6 servers, since it retains its files for longer than the other algorithms and needs to place fewer shares per timestep to do that.

The results in figure 5.8 are similar to the results in figure 5.7.

As before, the random algorithm loses all of its files very early in the simulation. The shares of happiness algorithm follows suit at around 500 timesteps. The servers of happiness

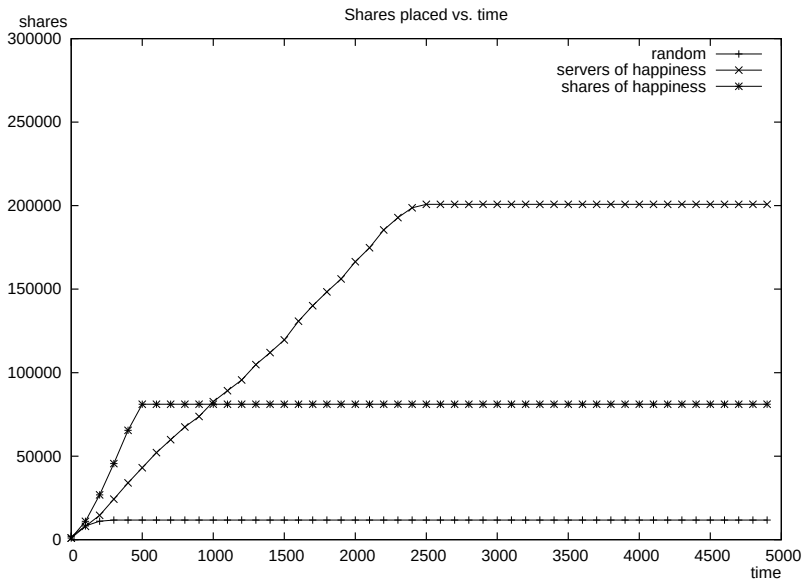

Figure 5.8: Shares placed versus time for an 8 server simulation

algorithm loses all of its shares by 2500 timesteps. The servers of happiness algorithm places more shares than either of the other two algorithms, which is to be expected considering the fact that it preserves its files for 1500 more timesteps than either of the other algorithms. In terms of share placement rate, servers of happiness clearly outperforms shares of happiness. The servers of happiness algorithm takes 1000 timesteps to place as many shares as the shares of happiness algorithm placed by 500 timesteps, when it lost all of its files. If both algorithms had retained all of their files until the end of the simulation, and if both algorithms had maintained the same share placement rate throughout the simulation, then the shares of happiness algorithm would have placed far more shares by the end of the simulation than the servers of happiness algorithm to achieve equivalent availability characteristics. Figure 5.9 is consistent with these results.

Once again, the random algorithm loses all of its files very early in the simulation. The shares of happiness algorithm loses all of its files at around 2000 timesteps, and the servers of happiness algorithm follows suit at around 4000 timesteps. As before, the servers of happiness places shares at a much more gradual pace than shares of happiness. Indeed, despite retaining its files for almost twice as long as the shares of happiness algorithm, the servers of happiness places around half as many shares in total over the course of the

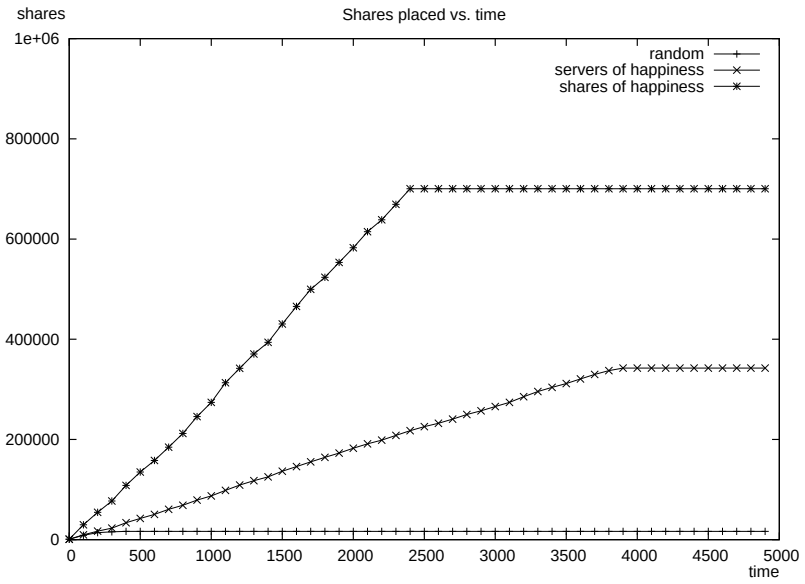

Figure 5.9: Shares placed versus time for a 10 server simulation

simulation as the shares of happiness algorithm, which is a very significant improvement. The line associated with the servers of happiness algorithm has a smaller slope in figure 5.9 than in figures 5.7 and 5.8. This is a result of the number of servers in these simulations. When there are more servers, we expect the servers of happiness algorithm to do fewer repairs per timestep, since it's less likely that enough of the servers will be broken at a particular timestep to require a repair of a file. In other words, servers of happiness has to perform fewer repairs per timestep when there are more servers than when there are fewer servers. These trends continue into the 12 server simulation, which is graphed in figure 5.10.

Figure 5.10 corresponds to a simulation in which both servers of happiness and shares of happiness retain all of their files until the end of the simulation. Servers of happiness has a clear efficiency advantage in this case. The shares of happiness algorithm places more than 3 times as many shares as the servers of happiness algorithm to achieve the same availability outcome.

Figures 5.11 and 5.12 graph the placement rates of the three algorithms in simulations in which the storage servers have an upper limit on the number of shares that they are willing to place.

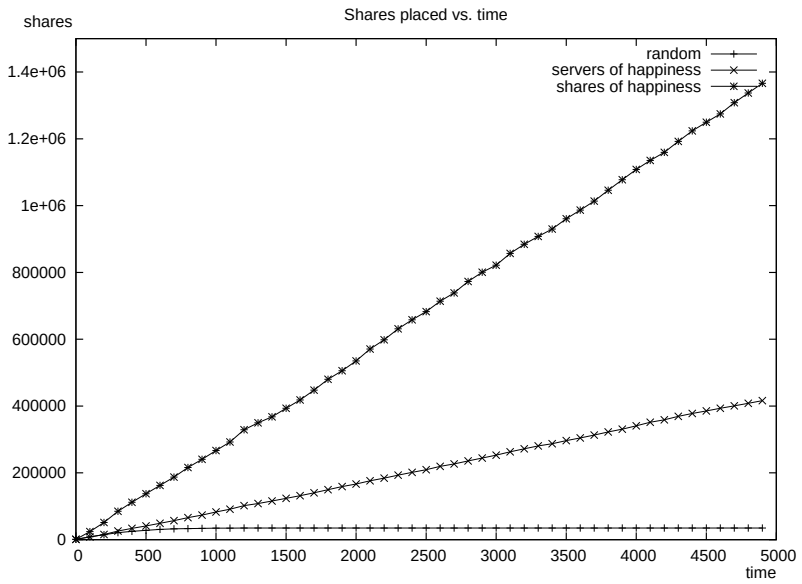

Figure 5.10: Shares placed versus time for a 12 server simulation

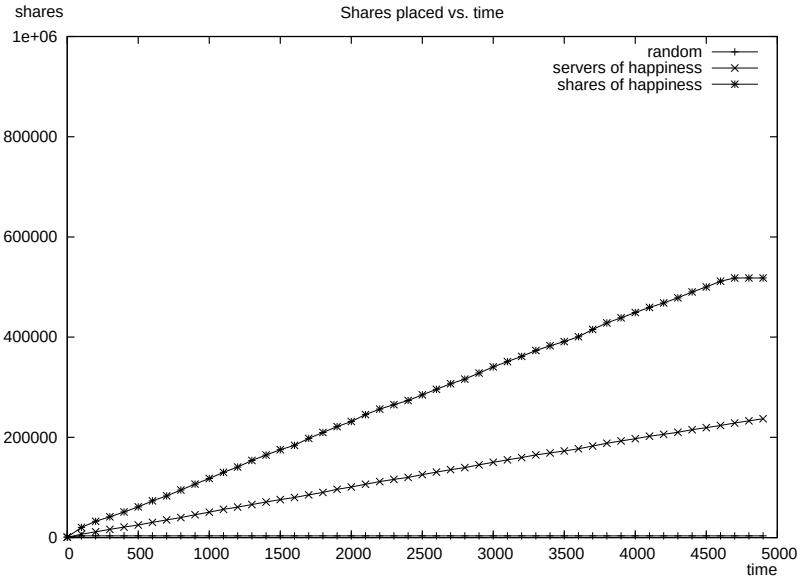

Figure 5.11: Shares placed versus time for a 10 server simulation with 280 share capacity

These are roughly consistent with figures 5.9 and 5.10. Specifically, the random algorithm places the fewest total shares, but at the cost of unacceptable file loss. The shares of happiness algorithm places more shares in total than either of the other two algorithms, and places shares much more quickly than the other two algorithms.

In general, servers of happiness has better efficiency characteristics than the other two

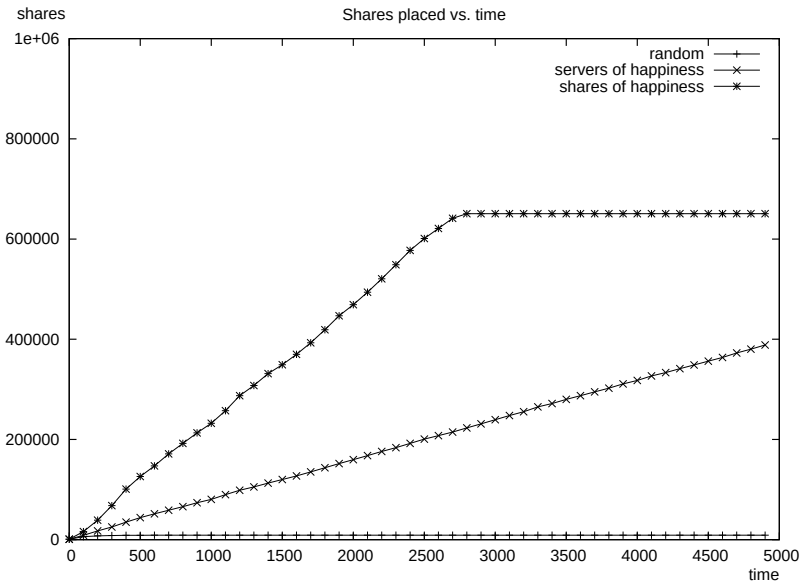

Figure 5.12: Shares placed versus time for a 12 server simulation with 280 share capacity

algorithms when file availability requirements are taken into account. Given equivalent availability outcomes, it tends to place fewer total shares than shares of happiness, and places fewer shares per timestep than shares of happiness. This suggests that the servers of happiness algorithm needs to do fewer repairs over the course of the simulation (suggesting that it is better able to deal with server failures than shares of happiness), and that it is better able to detect and use existing shares than shares of happiness.

5.7 Conclusions

We have discussed the results of a simulation that tested the performance of the servers of happiness, shares of happiness and random share placement algorithms on a storage grid with unpredictable server failures. Our results show that the servers of happiness algorithm is a significant improvement over the shares of happiness algorithm both in terms of file availability and efficiency. It retains more files for longer than the shares of happiness algorithm, and manages to do so with a far smaller share placement rate than the shares of happiness algorithm. In an actual distributed filesystem, these results suggest that the servers of happiness algorithm would do a better job of conserving resources than the shares

of happiness algorithm, and would do a better job of retaining and repairing files than the shares of happiness algorithm. We have not yet tested these conclusions, but intend to modify our work so it is suitable for inclusion in Tahoe-LAFS.

Chapter 6

Summary, Future Work, and Conclusions

We have presented the servers of happiness share placement algorithm. Specifically, we've introduced David-Sarah Hopwood's idea to use a bipartite graph to test for availability properties in Tahoe-LAFS, and used this idea to develop a share placement algorithm designed to produce share allocations that pass the shares of happiness test. We have discussed a simulation framework that allows us to test the performance of share placement algorithms on a distributed filesystem with configurable failure and recovery rates. We have shown that, in this simulation, our servers of happiness algorithm improves both in terms of availability and in terms of resource utilization on its predecessor, the shares of happiness algorithm. We have also left some unanswered questions. In particular, we would like to more thoroughly investigate the cause of the sudden and total file loss exhibited by both the shares of happiness and servers of happiness algorithms, and understand why these algorithms do not fail more gradually, like the random algorithm. This would likely involve improvements to the simulation framework; in particular, to its ability to output and effectively visualize data. In general, we feel that enhancing the simulation framework would allow us to better understand both the servers of happiness and shares of happiness algorithms, and possibly to discover ways that they could be improved. We would also like to better understand the relation of the number of servers and the encoding parameters

for files. Clearly, availability outcomes in our experiments improved as the number of servers increased, but this isn't a particularly novel conclusion. Ideally, we would like a guide for setting $\mathtt{n}$, $\mathtt{k}$, and any necessary health criterion based on the number of storage servers expected to be functioning at any given time. Finally, we would like to examine the assumption in our simulation that all storage servers have infinite storage capacity. Clearly, this isn't the case in the real world. Intuitively, these limitations would favor servers of happiness over shares of happiness, since servers of happiness has a generally smaller share placement rate than shares of happiness, but experimental validation of this intuition would be useful.

Works Cited

[1] Amazon.com. Amazon s3 pricing. http://aws.amazon.com/s3/pricing/, 2011.

[2] Rmy Card, Theodore Tso, and Stephen Tweedie. *Design and Implementation of the Second Extended Filesystem*, pages 1–6. 1994.

[3] Kevan Carstensen and Contributors. Servers of happiness. http://tahoe-lafs.org/trac/tahoe-lafs/browser/trunk/docs/specifications/servers-of-happiness.txt?rev=4497, 2010.

[4] Dropbox Company. Dropbox - simplify your life. https://www.dropbox.com/pricing, 2011.

[5] Linux Kernel Contributors. Ext3 filesystem. Documentation/filesystems/ext3.txt in the Linux kernel source tree, 2009.

[6] Linux Kernel Contributors. The second extended filesystem. Documentation/filesystems/ext2.txt in the Linux kernel source tree, 2009.

[7] The Tahoe-LAFS developers. "shares of happiness" is the wrong measure; "servers of happiness" is better. https://tahoe-lafs.org/trac/tahoe-lafs/ticket/778, 2009.

[8] Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazons highly available key-value store. In *In Proc. SOSP*, pages 205–220, 2007.

[9] David-Sarah Hopwood, Zooko Wilcox-O'Hearn, and Tahoe-LAFS developers. Welcome to tahoe-lafs. http://tahoe-lafs.org/source/tahoe-lafs/trunk/docs/about.html.

[10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[11] Andrew Loewenstern. Dht protocol. BitTorrent Enhancement Proposal 0005, 2008.

[12] Petar Maymounkov and David Mazires. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[13] Zooko O'Whielacronx and Contributors. Configuring a tahoe-lafs node. http://tahoe-lafs.org/source/tahoe-lafs/trunk/docs/configuration.rst, 2010.

[14] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD Conference*, pages 109–116, 1988.

[15] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[16] Cindy Swearingen. Zfs. http://hub.opensolaris.org/bin/view/Community-+Group+zfs/WebHome, 2010.

[17] David Teigland and Heinz Mauelshagen. Volume managers in linux. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 185–197, Berkeley, CA, USA, 2001. USENIX Association.

[18] Brian Warner and Contributors. File encoding. http://tahoe-lafs.org/trac/tahoe-lafs/browser/trunk/docs/specifications/file-encoding.txt?rev=3696.

[19] Zooko Wilcox-O'Hearn and contributors. Mnet: Faq. http://mnetproject.org/faq.

[20] Zooko Wilcox-O'Hearn and contributors. Mnet: related projects. http://mnet.sourceforge.net/relatedprojects.php.

[21] Zooko Wilcox-O'Hearn and contributors. tahoe-lafs.org "zfec". http://tahoe-lafs.org/trac/zfec.

[22] Zooko Wilcox-O'Hearn, David-Sarah Hopwood, and contributors. Tahoe-lafs architecture. http://tahoe-lafs.org/trac/tahoe-lafs/browser/trunk/docs/architecture.rst.

Appendix: Proofs about Maximum Matchings

Let $G = (V1 \cup V2, E)$ be a complete bipartite graph. Consider S, a subgraph of G. Let M be a maximum matching in S. Let G' be the subgraph of G induced by removing all vertices in M and all edges whose endpoints include vertices in M. Let M' be a maximum matching formed in G'.

Lemma 1: (*disjointness of* M *and* M')
M *and* M' *are vertexwise-disjoint*

Proof. By construction of G', no vertex that appears in an edge in M can appear in M'. Therefore, M and M' are vertexwise-disjoint. □

Lemma 2: (M *and* M' *are a matching*)
$M \cup M'$ *is a matching*

Proof. By hypothesis, M and M' are each a matching. By lemma 1, M and M' are vertexwise-disjoint, and, therefore, disjoint. Then $M \cup M'$ is a matching, as required. □

Lemma 3: (M *and* M' *are a maximum matching*)
$M \cup M'$ *is the same cardinality as a maximum matching* G.

Proof. Note that that the size of a maximum matching in G is the size of the smaller of $V1$ and $V2$. Suppose that this size is n. Suppose that $|M'| = m$. Note that m vertices are

removed from each vertex set to construct G' from G. Since G' is still complete, the size of a maximum matching in G' is $n-m$. Since M and M' are disjoint, $|M \cup M'| = m+n-m = n$, as required. □

Now let G be a complete bipartite graph, and let $R = (V1 \cup V2, E)$ be a graph such that one of $V1$ and $V2$ is a subset of one of the vertex sets in G. Let M be a maximum matching formed in R. Let G' be the subgraph of G induced by removing all of the vertices in M. Let M' be a maximum matching formed in G'. Let H be the graph formed by combining G and R.

Lemma 4: (*disjointness of* M *and* M')
M *and* M' *are vertexwise-disjoint*

Proof. Similar to lemma 1. □

Lemma 5: (M *and* M' *are a matching*)
$M \cup M'$ *is a matching.*

Proof. Similar to lemma 2. □

Lemma 6: (M *and* M' *are maximum*)
$|M \cup M'|$ *is at least as large as the size of a maximum matching in* G.

Proof. Let n be the size of a maximum matching in G. Since G is complete, we know that n is the size of the smaller of the two vertex sets composing G. Let $m = M$. Note that the smallest vertex set in G' is at most m vertices smaller than the smallest vertex set in G. Since G' is still complete, the size of a maximum matching in G' is at least $n - m$. Then $|M \cup M'|$ is at least $m + n - m = n$ units in size, as required. □

Lemma 7: (M *and* M' *are a maximum matching in a new graph*)
Let H *be the combination of* G *and* R. *Then* $M \cup M'$ *is a maximum matching in* H.

Proof. Assume, for contradiction, that there is a matching Z in H such that $|Z| > |M \cup M'|$.

We first note that Z can be expressed as the disjoint union of a set of edges Q from R and another set of edges F from a subgraph G'_Q of G induced by Q, as described above. If Z contains edges from $G - G'_Q$, then Z is not a matching, a contradiction; thus, edges in F must come from the subgraph of G induced by Q. Clearly, Q and F must themselves be matchings if Z is to be a matching. Note that $|Q| \leq |M|$, since M is a maximum matching in R, and since Q is a matching in R. Since G'_Q is a complete graph, the size of F is bounded from above by the size of the smaller of the two vertex sets in G'_Q.

Formally, we say that for some complete graph G, and some edge set Q from R, as described above, the contribution bound of the complete graph G'_Q, as described above, is the size of the smallest vertex set in G'_Q; we denote this as:

$$\mathit{Smallest}(G'_Q)$$

Using this terminology, we can express the cardinality of Z as:

$$|Z| \leq |Q| + \mathit{Smallest}(G'_Q)$$

noting that Z need not use a maximum matching from $\mathit{Smallest}(G'_Q)$, and that Q and G'_Q have disjoint edge sets by construction, thus allowing us to add $|Q|$ to the bound.

We can express the cardinality of $M \cup M'$ as

$$|M \cup M'| = |M| + \mathit{Smallest}(G'_M)$$

We now observe that $|M| > |Q|$. If $|M| = |Q|$, $\mathit{Smallest}(G'_Q) = \mathit{Smallest}(G'_M)$, since G is a complete graph, and we have that

$$\begin{aligned} |Z| &\leq |M| + \mathit{Smallest}(G'_M) \\ &\leq |M \cup M'| \end{aligned}$$

which is a contradiction. Since $|M| > |Q|$, we have that

$$\mathtt{Smallest}(G'_Q) \leq \mathtt{Smallest}(G'_M) + (|M| - |Q|)$$

since $\mathtt{Smallest}(G'_Q)$ is at most $|M|-|Q|$ larger than $\mathtt{Smallest}(G'_M)$, as G'_Q has $|M|-|Q|$ more vertices in the vertex set it shares with R as G'_M. Then

$$\begin{aligned}
|Z| &\leq |Q| + \mathtt{Smallest}(G'_Q) \\
&\leq |Q| + \mathtt{Smallest}(G'_M) + (|M| - |Q|) \\
&\leq |M| + \mathtt{Smallest}(G'_M) \\
&\leq |M \cup M'|
\end{aligned}$$

which is a contradiction. Then $|M \cup M'|$ is a maximum matching in H.

□

Theorem: 8. *The share placement that results from the three-phase matching construction used in servers of happiness is a maximum matching in the bipartite graph induced by the state of the filesystem.*

Proof. Note that the three phase share placement protocol is a composition of lemma 7 and lemma 3. Specifically, it is composition of a matching in a partially distinct graph of readonly share placements combined with a matching in an induced complete subgraph from which a maximum matching is extracted by combining a maximum matching in a subgraph of the complete graph and a maximum matching in an induced complete subgraph. Then, by lemmas 3 and 7, the end result is a maximum matching. □