Robust Distributed Storage Using Erasure Codes

James W Anderson Kelly Koskelin Cliff Frey

December 5, 2002

Abstract

This paper describes a Robust Distributed Storage system (RDS) that can tolerate multiple simultaneous server failures. RDS achieves high reliability while maintaining efficient use of storage by encoding data using erasure coding techniques and distributing the encoded data fragments across multiple servers. Erasure coding allows data to be reconstructed from a subset of the stored fragments, providing availability even when multiple servers fail.

1 Introduction

Hard disks, being mechanical components, are inherently unreliable. There are many techniques for increasing the reliability of storage systems by redundantly storing data across several disks, possibly distributed across different machines. Most current distributed schemes use replication to copy the data to multiple nodes. While replication can provide reliable storage, in general, $n+1$ replicas are needed to survive n failures, which means that $n+1$ times the amount data to be stored is needed as the total storage in the system. For a system that stores a large amount of data or must survive the loss of many nodes, the storage overhead for replicated data can quickly become untenable.

This paper describes the Robust Distributed Storage (RDS) system, a new distributed storage system that uses erasure coding algorithms to tolerate multiple node failures while using significantly less storage overhead than replication. Distributing erasure coded data allows RDS to provide higher availability than replication schemes with far less storage space and network bandwidth.

The system has been designed and implemented for two purposes. The first is to provide a testing ground for erasure coding algorithms. Many algorithms are known, and their asymptotic performances are known, yet few of these algorithms have been implemented in software. RDS is a platform that can be used to analyze how these algorithms perform on real data and hardware and to gauge their usefulness and practicality in actual systems.

RDS is also designed to be a useful storage system. It provides a block device interface that can be used with a cluster of storage nodes (disks or servers). Although each individual node may not be reliable, the system as a whole provides reliable storage. Currently, RDS can be used by one client with read-write access, multiple clients with read-only access, or multiple clients with read-write access to separate partitions. Also, because the system provides a block device interface, one client can mount RDS as a file system and then export it using a network file system protocol, such as NFS. Used in this way RDS provides a highly reliable back-end.

Section 2 presents an overview of the system, and section 3 describes some of the major design decisions involved. Section 4 discusses common client uses of the system, followed by performance analyses in section 5. Finally, we discuss related work and conclude.

2 System Overview

RDS is structured in three layers: the client interface, the controller, and the storage layer, consisting of fragment servers. Each layer communicates with the one below it via RPCs. The client interface layer provides a block device interface to the end client host and is the only part of the system that needs to be run in the kernel. The client sends read and write requests for blocks to the controller. The controller performs the necessary encoding and decoding and communicates with the fragment servers to service these read and write block requests. The fragment servers store the encoded data fragments.

Layer	**Responsibilities**	**Machine**
Client Interface	block device interface; send block requests to Controller	Client
Controller	service block requests; code data; access storage layer; ensure integrity of blocks	Client
Storage	Store fragments	Distributed servers

Table 1: RDS Software Layers

2.1 Client Interface

An existing implementation of the Network Block Device (NBD) was modified to provide a useful client interface for the system. NBD provides a block device interface to remote servers. The client host uses the `nbd` kernel module—the only part of the system that is not run in user space—to create a new NBD device. This device can be treated like any other block device and thus a user may easily create a file system on top of RDS. The kernel passes all read and write requests for the NBD device to the NBD client. The NBD client sends these requests through a UNIX socket to an NBD server. Under normal usage, these requests would be sent over the network using a TCP connection and the server would normally service the requests from a random-access file. For the RDS system, however, we modified the NBD server to send RPCs containing the request to the controller and to then return the response from the controller, rather than serving the request directly from a file. In the RDS system, the NBD client and server are typically run on the same machine (although there is no requirement to do so) and are collectively known as the NBD client interface. Additionally, even though the NBD server uses RPCs to communicate with the controller, the controller is also typically run on the same machine as the the NBD client interface (although, once again, there is no requirement to do so). The use of RPCs between these layers provide a well-defined interface and hard modularity between components. The machine running the NBD client interface plus the controller will be referred to as the client within the RDS system.

2.2 Controller

The controller is the central layer and core component of the RDS system. The controller coordinates all communication from the client layer to the storage layer. The controller performs the erasure encoding and decoding and issues RPCs to the fragment servers to store and retrieve the encoded fragments. If some fragment servers fail to respond, the controller will continue to serve read and write requests as long as enough fragments can be retrieved to rebuild the data. This point of failure is determined by details of the erasure coding algorithm (section 3.1). The controller ensures the integrity of the data through the use of an atomic write protocol (section 3.2 and by correcting invalid data fragments (section 3.3).

The controller receives read and write requests for blocks from the client and turns them into read and write requests for fragments. To perform this translation the controller encodes and decodes blocks according to the chosen coding scheme. We implemented a Reed-Solomon [3] scheme, as well as a few other simple coding schemes. The modular design of the controller allows the coding algorithms to be easily changed.

The controller decides which fragment server is responsible for a given fragment by taking the fragment's unique identifier (section 2.3) modulo the number of servers. This fragment distribution method has the limitation that the controller communicates with a fixed, unchanging set of fragment servers. Although fragment servers will inevitably become unavailable, our implementation assumes that they will eventually be restored. Similarly, in our implementation, new servers cannot be added. The permanent loss or addition of a server would require redistributing the fragments across the new set of servers. It would not be difficult to modify our implementation to handle both of these scenarios.

2.3 Fragment Servers

Fragment servers occupy the lowest layer in RDS and communicate with the storage medium. Fragment servers receive read and write RPC requests from the controller and retrieve or store fragments accordingly. Fragments are identified by a timestamp and a globally unique 64 bit fragment identifier. The timestamp is used to ensure that when a block is reconstructed all fragments used are from the same version of the block (section 3.3). The fragment identifier is obtained by concatenating the globally unique block identifier with the block-specific fragment number, thus ensuring a unique identifier for the fragment. Fragments are stored in a tree-like directory structure indexed by the fragment identifier. This avoids the slow linear searches that would result if all of the fragments were stored in a single directory.

3 Design and Implementation

Designing RDS involved a number of major considerations. We needed to choose and implement an erasure coding algorithm. To maintain the integrity of blocks, it is necessary to ensure atomic writes. Additionally, because writes occur even when some servers are down or unreachable, it is necessary to restore fragments that are lost due to transient network failures or node failures.

3.1 Erasure Coding Algorithms

Erasure codes provide robustness by adding parity data to their input. Erasure codes treat data as a sequence of letters. Consider an erasure code that takes an input of n letters, and adds m letters of parity data. This code can recover the input data even if m letters are lost. Often codes are discussed in terms of the number of letters of input and the total number of letters of output (n,$n+m$). For instance, a (4,5) erasure code takes four letters of input data ($n = 4$), adds one letter of parity data ($m = 1$), and then can tolerate the loss of any one letter. The space overhead of a code is m/n, and the number of failures tolerated by a code is m.

3.1.1 Reed-Solomon

An existing library implementing a Reed-Solomon erasure code is used in RDS. Reed-Solomon codes are a class of erasure codes. The performance of Reed-Solomon codes is quadratic on the total number of letters and tolerated erasures. For a given number of tolerated errors, Reed-Solomon codes can only accept inputs of a certain length; for RDS, this length is 4 bits. RDS uses an (11,15) code that can tolerate the loss of any 4 fragments and has a space overhead of 36%. To achieve the same degree of robustness with replication, there would need to be 5 copies of all of the data, requiring a space overhead of 400%.

Reed-Solomon codes have the added benefit of being error-correcting. If more than the minimum number of fragments are available, this code will use the parity data to check for errors in the input data. For every two letters of parity data, one letter can be corrected.

3.1.2 Other Algorithms

In addition to using the Reed-Solomon library, we implemented several other simpler algorithms, or schemes, to use for testing and benchmarking. The simplest is the Null scheme, which uses the entire block as the fragment. The Divide scheme simply divides a block into equal sized fragments with no redundancy. The final, more interesting, scheme is the XOR scheme. This works like Divide scheme except it additionally computes a parity bit fragment; that is, each bit of the parity fragment is the XOR of the corresponding bits in all the other fragments. This gives the XOR scheme a single fragment of redundancy (it is an (n,$n+1$) erasure code). If one fragment is lost, it can be recomputed by calculating the XOR of the remaining fragments (including the parity fragment). As a point of comparison, RAID5 [4] uses an equivalent XOR algorithm to achieve redundancy.

3.2 Atomic Writes

RDS ensures that all writes are atomic using a two-stage quorum-based commit protocol. When the controller receives a block write request it sends prepare messages to the fragment servers. Upon receiving a prepare message, a fragment server synchronously writes the new fragment to disk, but marks the data as a "new" copy, keeping the old data for that fragment as the "current" copy. Once the fragment server has ensured that the new data has been written to the disk, it responds with a success message to the controller. The controller will wait for enough responses to know that the block can be recovered, even if the entire system were to crash, before it sends commit messages. When a fragment server receives a commit message it updates its "current" copy of the fragment to be the "new" data that it had previously prepared. The fragment server performs this update atomically.

Without the atomic commit protocol, there could be failures that would result in a block containing invalid data, which would violate the semantics of a block device. For instance, if the controller used merely a single-stage write and crashed after sending half of the fragments, then the block would have become corrupted: too many of the old fragments would have been overwritten to reconstruct the old data, but not enough of the new fragments would have been written to reconstruct the new data. The two stage protocol prevents this type of block corruption by ensuring the atomicity of writes in all failure cases. In other words, if some or all nodes fail during a write, either the write will complete successfully or have no effect.

In one failure case, more fragment servers become unreachable than the coding algorithm can tolerate, but the controller does not fail. For this situation, the controller, having not received the requisite number of responses, knows that the write has failed. It sends abort messages to the remaining fragment servers, instructing them to delete their "new" copies, and then it responds with a failure message to the client.

The more complicated case involves the failure of the controller and possibly some number of fragment servers, or perhaps even the catastrophic failure of the entire system crashing. If the controller crashes in the middle of a write, some of the fragment servers may have both the new and old data on disk. On restart, the controller runs a recovery process that fixes this situation. First, the controller asks all of the fragment servers if they have any uncommitted data. If they do, the controller tries to read the block(s) corresponding to all uncommitted fragments. Using the result (success or failure) of this read, the timestamp for the read (if successful), and the timestamps of each of the fragments, the controller decides how to correct one of four possible outcomes for an interrupted write, which are illustrated in Figure 1. For this figure and the following explanations, assume the use of a (3,4) erasure code. In (A), the commit stage has not been reached and the old data is successfully read. The controller recovers by sending abort messages to the fragment servers that have successfully prepared. In (B), the commit has begun, but the old data can still be successfully read because less than m servers have completed the commit. However, since the write had begun (the controller can determine this by checking the timestamps of the new fragments verses the block), the controller recovers by sending commit messages to the servers that have prepared but not committed. In (C), neither the old data nor the new data can be read—too many servers committed to rebuild old data but too few have committed to rebuild the new data. As in (B), the controller recovers by sending commit messages to the servers that have prepared but not committed, thereby making the new data readable. In the final case (D), the new data can be successfully read since enough servers committed. The controller recovers by sending commit messages to any servers that have prepared but not committed.

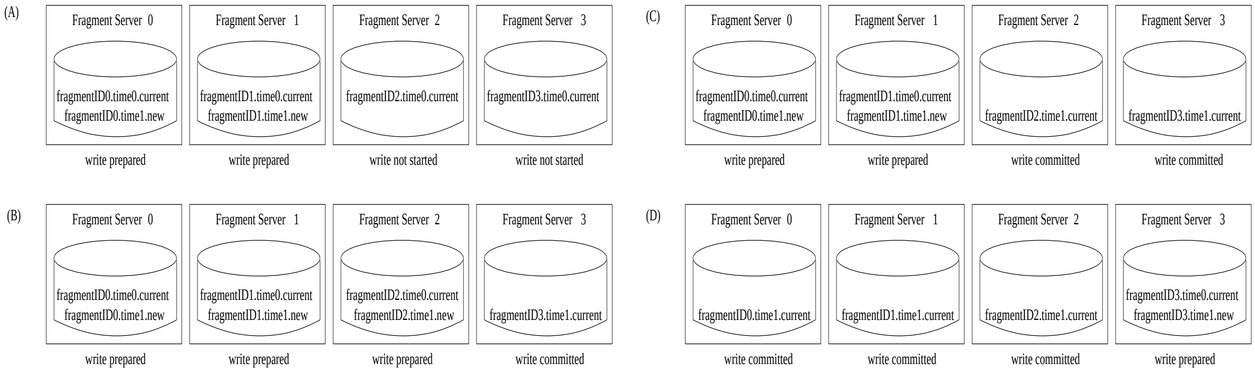

Figure 1: Atomic Write Recovery Cases

3.3 Block Restoration

There are two situations in which a fragment server may have missing or out of date fragments. A server may have missed some updates due to transient network failures or because it had crashed or was offline. Additionally, a server may need to be completely reconstructed if its disk fails and becomes unreadable.

On reads, the controller can detect if a fragment server has missed updates because the timestamp(s) on the out of date fragment(s) will not match those of the rest of the fragments. For every read, the controller keeps track of which fragment servers respond with a "missing fragment" status or have out of date fragments and updates these servers with the most recent data. This is a somewhat expensive operation because it involves re-encoding the block that was just read to reconstruct the missing data. Restoration on reads is important for ensuring the integrity of commonly read blocks, but does not help for blocks that have not been read recently.

If a fragment server has crashed or has been offline, it may have missed a number of writes or possibly lost all of its previous data (if its disk failed). To restore missing fragments in this situation, the controller periodically queries the fragment servers to see if they have been down since the last query. Each time a fragment server boots it sets an "I just rebooted" flag. When the controller detects a rebooted fragment server, it reconstructs the this server's data by reading every block in the system and issuing writes just for the fragments that belong on that server. After beginning the reconstruction process, the controller tells the fragment server to clear its "rebooted" flag.

Reconstruction is expensive, but the frequency of checking the fragment servers for reboots can be varied to meet the usage patterns of the system. For commonly read blocks reconstruction will happen on each read; less frequently accessed blocks can be reconstructed by running the mechanism used to restore a server that has rebooted. Since this is an intensive operation, it can be run when system usage is low.

4 Client Usage

4.1 One Client, Read-Write

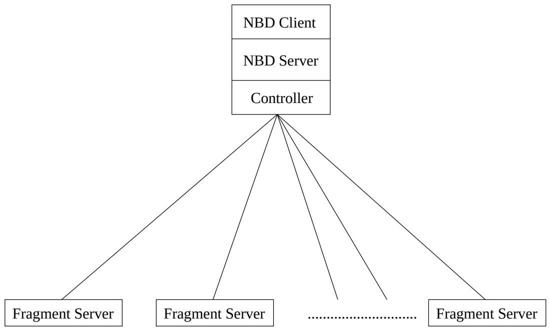

Figure 2: One Client, Read-Write

Because RDS presents a block device interface to the client, in normal read-write operation only a single client may access the system. RDS can serve as back-end storage for a server that needs high capacity reliable storage, such as a file or database server. In the case of a file-server, it is possible for the RDS client to export the RDS system over some other network file system protocol, such as NFS, to multiple clients with read-write access. We expect that RDS may be an attractive option for systems with very large storage requirements, where using a system that achieves robustness through replication becomes prohibitively expensive. While this configuration will provide robust storage, it does not provide high availability because the RDS client is a single point-of-failure. However, a failure of the RDS client will not result in the loss of any data, but only the loss of service until the client reboots.

4.2 Multiple Clients, Read-Only

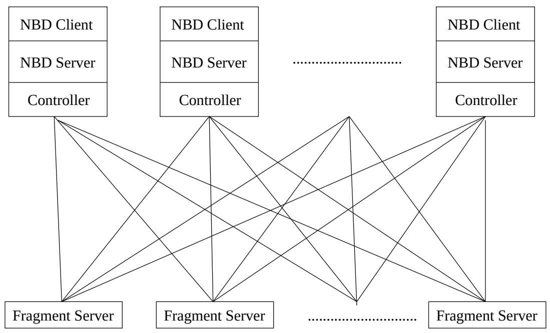

Figure 3: Multiple Clients, Read-Only

RDS can provide both high availability and reliability when configured to be used read-only. First, a single client (in read-write mode) writes the data to the RDS. The fragment servers are then rebooted in read-only mode. After that step, any number of clients may simultaneously read from the RDS. All of these clients run their own RDS client software and communicate with the fragment servers directly. The failure of one client will not affect the any other client. As long as enough fragment servers are reachable by any given client, the system will remain available to that client. This scenario is ideal for any sort of file distribution system.

4.3 Multiple Partitions, Read-Write

In the previous scenarios, any RDS client had access (at the block level) to the entire contents of the RDS. There is a third mode of operation which allows read-write access by multiple clients, but no the client may access the data of any other client. All of the clients share the same set of fragment servers, but each client is configured with its own partition, or set of blocks in the RDS, that it is allowed to access. To ensure correctness, none of the partitions may overlap. This usage mode has the advantages of the read-only mode, in that it also provides high availability by having each RDS client run its own controller. Typically, RDS might be used in this mode to provide some amount of highly available and reliable storage for each individual in a collaborative group. Every member of the group runs a fragment server and contributes a portion of their disk to the RDS system, and in turn configures their controller to access their partition (through the fragment servers of everyone else in the group). This type of configuration might also be used in any networked workstation environment, in which each workstation participates in the system.

5 Performance

Our analysis of RDS's performance indicates that RDS suffers from substantial overhead unrelated to the coding algorithm. Modern disks can provide throughputs of around 30 megabytes per second (MB/s) and modern NFS implementations can achieve similar throughputs. Our system, on completely cached loads, can only read a few hundred kilobytes per second (KB/s). Some factors for this reduced performance are unavoidable, and others are specific to our implementation.

5.1 Network Overhead

Table 2 shows the effect of network overhead on RDS. In the local case, the fragment servers are run on the same machine as the controller. This allows the kernel to avoid much of the network

	Read (KB/s)	Write (KB/s)
Local	599	162
Remote	171	61.7

Table 2: **Network overhead comparison.** These benchmarks reflect the Reed-Solomon scheme.

protocol processing and removes any network latency. In the remote case the controller is run on the same machine as in the local case but the fragment servers are run on a different machine on a 10Mbit network. The observed slowdown is caused by the CPU overhead of packet processing and the latency of the network. The machine used for these benchmarks had two 933MHz Pentium III processors and 1GB of memory. The benchmark averaged repeated reads and writes of 1MB file.

The overhead observed for remote reads and writes are due to the fact that for each block request multiple fragments must be transmitted over the network, but fragments are only a few hundreds bytes long. With the Reed-Solomon coding scheme every 8KB block read generates 15 RPCs. Writes are twice as expensive because of the two-stage commit process.

5.2 Encoding Overhead

There are two parameters of a coding algorithm that affect performance; the expense of the encoding and decoding and the number of fragments generated. More fragments cause more network overhead. Table 3 shows throughput results under three different encoding schemes. The first two entries compare Reed-Solomon to XOR with a similar number of fragments. This shows the overhead of the Reed-Solomon coding slows reads down by at least 25 percent and does not slow down writes. The writes are probably faster for Reed-Solomon because there are slightly fewer fragments. It makes sense that writes are less sensitive to CPU overhead from decoding because the extra network overhead from the two stage commit makes the number of fragments much more important. The bottom two entries in the table show the overhead of having more total fragments. We can see that having approximately three times as many fragments slows reads down by 37 percent. This indicates that decreasing the number of RPC calls in our code could help performance significantly.

The client machine used for these benchmarks had two 933MHz Pentium III processors and 1GB of memory. The server had a 200MHz Pentium Pro processor and 128MB of memory. The client and server were connected through a 10Mb Ethernet switch. The benchmark averaged repeated reads and writes of 1MB file.

Erasure coding algorithms that are both efficient and provide a high degree of data protection do not exist in practice. Simple algorithms such as XOR exist and can be made quite efficient, but all known algorithms for higher degrees of protection have significant overhead. Used in isolation, our Reed-Solomon encoder can only encode and decode at a rate of a few megabytes per second. In order to get performance comparable to current disks and networks, dedicated hardware would be required or the work of encoding and decoding would need to be distributed to the fragment servers themselves.

5.3 Number of Server Machines

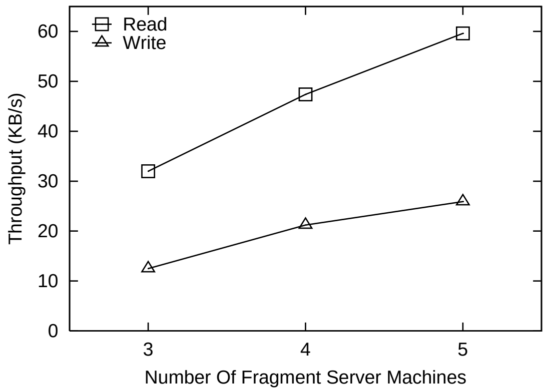

Figure 4: Throughput vs Number of Fragment Server Machines

RDS provides the most robustness when each of $n+m$ (15 for our Reed-Solomon scheme)

	Servers	Tolerated Failures	Read (KB/s)	Write (KB/s)
Reed-Solomon (11,15)	15	4	171	61.7
XOR (16,17)	17	1	215	44.1
XOR (4,5)	5	1	346	251.0

Table 3: **Encoding Scheme Throughputs.** The "Servers" column denotes the number of fragment server processes. All fragment server processes ran on the same physical machine.

fragment servers runs on its own physical machine. Unfortunately, we were unable to fully test our system with that many machines. The best that we were able to achieve was to run a total of 15 fragment server processes distributed among five machines. Figure 4 shows performance of our system versus the number of machines running fragment servers. Performance improves significantly as more fragment server machines are used because they all can perform fragment read and write operations in parallel.

The client machine used for these benchmarks had two 933MHz Pentium III processors and 1GB of memory. The servers varied but were all comparable to or faster than a machine with a Pentium Pro processor. Some of the servers were connected to the client by a 10Mb Ethernet. The benchmark averaged repeated reads and writes of 1MB file.

5.4 Other Performance Factors

Even in a ideal implementation, each request to the controller would still need to be sent to the minimum number of fragment servers required to service that request. This adds at least one network-round-trip delay to any response. The other unavoidable factor is any overhead generated by the encoding and decoding algorithms. With proper read-ahead and caching, the network-round-trip delay will only effect latency and not throughput. Ultimately, sustained throughput should be as high as the encoding rate of the erasure code.

In order to improve the performance of our system, a few changes need to be made. First, the controller needs to be able to read many fragments at a time to handle large requests efficiently. This would dramatically decrease the number of RPC calls made between the controller and the fragment servers. Second, the fragment servers should store fragments in one large file instead of storing each fragment in its own file. This would improve the performance of large reads and writes because there would no longer be the overhead of having many files and directories. If these improvements were implemented, then it is quite possible that the CPU usage of the controller would become the bottleneck instead of network latency.

A significant trade-off in our system is our client interface. We chose to export a block device as the storage layer that would be backed by erasure codes. This simplified implementing a file interface because a file system can be easily mounted on the block device. It has the drawback, however, of forcing the granularity of access to be relatively small, generally a few kilobytes. Many erasure codes could be much more efficiently implemented with larger chunks of data.

6 Related Work

RDS protects data by using some space to store redundant data similar to the way in which RAID [4] protects data with mirroring or a checksum. Unlike RAID, which is limited to either replication or a single checksum disk, erasure codes have a more flexible redundancy model. Our system is also meant to be used over a network whereas most RAID implementations require all of the disks to be connected to just one computer.

Existing systems such as Freenet [1] and CFS [2] use replication to store data across many untrusted hosts. Both of these systems can lose data when untrusted hosts on the network either hide their data, become disconnected, or crash. Freenet uses a probabilistic model that favors files that are accessed frequently. CFS uses a fixed replication factor. The replication factor has the disadvantages of both requiring more bandwidth to create the replicated data copies when new data is stored and using significantly more disk space to store the replicated copies.

Typhoon [6] uses erasure codes and is a system meant for transmitting video data over unreliable broadcast or multicast networks. Because packets that are sent are erasure encoded, the clients still receive all of the data even with a considerable amount of packet loss. They use a probabilistic encoding algorithm that has lower CPU overhead than Reed-Solomon codes.

Frangipani [5] is a file system built on top of the Petal storage layer. Our system is similar to the Petal storage layer except that it provides some guarantees on the ordering of writes and uses replication. The file system itself is specially designed to allow concurrent write access to Petal. A similar design of adding some guarantees to our system and using a specialized file system could allow many controllers to share fragment servers with read-write access, improving availability.

7 Conclusion

RDS was designed to test erasure coding algorithms in practice and to be a useful storage system. In its current implementation, the performance of RDS is dominated by network overhead, regardless of the coding algorithm used. Future implementations should concentrate on reducing network overhead to make clear the differences between the algorithms.

One of the major advantages of our system is its efficient use of storage. Because we use erasure codes, our system provides high reliability with relatively low storage overhead for redundant data. To be more precise, the amount of redundant data required by RDS is 4/11, or 36% overhead (that is, the total storage needed for the system is 1.36 times the amount of data that it will store). A replicated storage system able to survive four node failures would require at least 400% redundant storage overhead. This is quite a significant difference, especially for very large amounts of data.

Finally, we believe that we actually have built a useful storage system. We ran a number of stress tests, and were very pleased with the results. We set up what we thought would be a typical usage: a single client with read-write access to many fragment servers. We created an ext2 file system on top of RDS, and then exported this file system over NFS. With several NFS clients performing continuous operations on the mounted file systems (playing MP3s, running compilations, and copying data), we then simulated a number of failures in the system by killing various fragment servers. As expected, we were able to kill four of the fragment servers and still have the clients continue their use uninterrupted. We then deleted the data from some of these fragment servers to simulate disk failures, while leaving old data in others, and rebooted them. Finally, after the reconstruction process rebuilt the servers, we killed another four servers. As before, the system survived these losses and the NFS clients, unaffected, continued their operations. Given the resilience of our system in these rigorous tests, we believe that RDS succeeds in providing a usable robust storage system.

References

[1] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[2] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[3] Phil Karn. Forward reed-solomon codes, 2002. URL: `http://www.ka9q.net/code/fec/`.

[4] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks RAID. In *Proceedings of the ACM SIGMOD '88*, June 1998.

[5] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[6] Hakim Weatherspoon, Matthew Delco, and Shelley Zhuang. Typhoon: An archival system for tolerating high degrees of file server failure. 2000. URL: `http://www.cs.berkeley.edu/~hweather/Typhoon/TyphoonReport.html`.

[7] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. *Proceedings of the First International Workshop on Peer-to-Peer Systems*, 2002.