Separating Durability and Availability in Self-Managed Storage

Geoffrey Lefebvre and Michael J. Feeley

Department of Computer Science
University of British Columbia

{geoffrey,feeley,}@cs.ubc.ca

ABSTRACT

Building reliable data storage from unreliable components presents many challenges and is of particular interest for peer-to-peer storage systems. Recent work has examined the trade-offs associated with ensuring data availability in such systems. Reliability, however, is more than just availability. In fact, the durability of data is typically of more paramount concern. While users are likely to tolerate occasional disconnection from their data (they will likely have no choice in the matter), they demand a much stronger guarantee that their data is never permanently lost due to failure. To deliver strong durability guarantees efficiently, however, requires decoupling durability from availability. This paper describes the design of a data redundancy scheme that guarantees durability independently from availability. We provide a formula for determining the rate of redundancy repair when durability is the only concern and show that availability requires much more frequent repair. We simulate modified versions of the Total Recall block store that incorporate our design. Our results show that we can deliver durability more cheaply than availability, reducing network overhead by between 50% and 97%.

1. INTRODUCTION

In recent years, many projects have aimed at building reliable storage from unreliable components. Of particular interest have been peer-to-peer systems, which build reliable storage in a completely decentralized manner. For these systems, reliability is typically achieved by focusing on data availability. Providing availability in such an environment, for example, has been addressed explicitly in systems such as Total Recall [3].

This work has shown that strong availability guarantees can be expensive to achieve with unreliable storage components such as those found in peer-to-peer systems. The problem is that individual nodes tend to leave and rejoin these systems frequently and to have low average availability (i.e., $< 50\%$). In addition, overall node populations exhibit a high rate of churn with frequent nascent additions and permanent losses.

Availability is expensive to maintain due to the combined costs of storing redundant data and maintaining this redundancy as nodes fail. Redundancy allows the system to handle transient departures but inevitably, some departures are permanent. Over time, redundancy decays, and unless it is restored via a repair process, the system eventually loses data. To prevent this, the repair process monitors the state of the system and stores new copies of the data on other nodes when needed. The network bandwidth needed to transfer these copies is principally what makes maintaining availability an expensive proposition in peer-to-peer environments [4].

The key idea of this paper is to observe that data reliability really has two components — availability and durability — and that users have different expectations of each. The typical user is willing to accept availability of say two 9's of probability (i.e., 1%) as long as data unavailability is transient. Users expect a much stronger guarantee that failures do not cause permanent data loss, perhaps as strong as ten 9's (i.e., 1 in 10^{10}). This is precisely the principle behind offline backups.

By decoupling durability from availability, it is possible to provide ten 9's of durability guarantee without needing a similarly strong availability guarantee. This decoupling provides substantial benefits, because durability can be guaranteed much more cheaply than availability in terms of network bandwidth. Our results show that in the worst case, our durability-based approach consumes less then half of the bandwidth consumed by the availability-based mechanism of Total Recall. In the best case, our approach is over thirty times more efficient.

The main challenge is that while measuring availability is a simple task, evaluating durability is much harder. Availability can be directly monitored, for example, using heartbeats to determine which nodes are online. If the minimal number of nodes necessary to retrieve the data are online, the data is available. On the other hand, durability can not be measured directly, but only estimated. A node that fails to respond to heartbeats is unavailable but has not necessarily failed permanently.

The focus of this paper is to address this concern. We develop a heuristic that allows us to calculate durability as a probabilistic guarantee. Since durability cannot be measured directly, our system tolerates changes in redundancy without immediate knowledge of these changes. We determine an upper bound on changes (i.e. failures) that the system can tolerate without risking data loss, as defined by our durability guarantee. We also describe the design of an automated redundancy mechanism based on this formulation. Finally, we evaluate this mechanism by simulating a hypothetical system that incorporates our durability mechanism into the Total Recall block store.

The rest of this paper is organized as follows. Section 2 describes our durability-based approach for the most common redundancy techniques: replication and erasure encoding. Section 3 briefly describes the design of our redundancy mechanism based on the Total Recall [3] block storage architecture. Section 4 presents simulation results that demonstrate the benefit of our approach and Section 5 concludes.

2. REDUNDANCY MAINTENANCE

Any system that protects data from failure does three things: it stores data redundantly, monitors its redundancy and responds to failures by restoring the redundancy when needed. These mechanisms are guided by two key parameters: the amount of redundancy created and the time required to detect and repair failures. Reliability is improved either by increasing the amount of redundancy or by shortening the repair interval, or both. But, doing so increases system overhead, particularly the network bandwidth required to copy redundant data between nodes.

There are two main techniques for storing data redundantly: replication and erasure encoding. Using replication, a system stores multiple exact copies of the data on distinct nodes. With *erasure encoding*, the system subdivides and encodes data to generate a set of fragments with overlapping redundancy, which are then distributed to distinct nodes. This encoding produces n fragments from $m < n$ data subdivisions such that any subset of m fragments is sufficient to reconstruct the original data. Erasure encoding is more space efficient than replication; for the same storage overhead and expected node availability, erasure encoding provides a significant increase in availability over replication [2, 9]. On the other hand, erasure codes have a higher processing cost and metadata overhead, which can be significant for small files [3]. Since both techniques have their respective advantages, our approach is designed to work with either one.

While research on reliability in peer-to-peer systems has focused on ensuring data *availability*, we instead focus on ensuring *durability*. Durability is a weaker property of a system that ensures data can be eventually retrieved, but with some delay if it is currently unavailable. It is, of course, typically the most important property as well. A system that guarantees durability need only be concerned with permanent node failures and can thus ignore transient ones, as long as it can tell the difference between them. If it can, and most failures are transient, then the system is able to guarantee durability at a lower cost than availability. Or, put another way, the same redundancy degree and repair frequency yields a stronger guarantee of durability than of availability. Numerous studies have in fact shown that in peer-to-peer systems, the ratio of transient over permanent failures is high. A study of the Overnet peer-to-peer file sharing system, for example found that approximately 20% of the population permanently leaves the system every day, but the average number of joins and leaves is 6.4 per node per day [1]. In this scenario, transient departures outnumbered permanent departures by a ratio of more than 15 to one.

The remainder of this section describes our approach for monitoring and repairing redundancy when durability is the only concern. We expect, however, that our technique would actually be used in conjunction with standard high-availability approaches so that durability and availability are both maintained, but with different probabilities. We begin by discussing how to approximately differentiate permanent and transient failures. Key to our approach is to establish an upper bound on the number of failures a system can continuously sustain, and repair, without loosing data; we compute different bounds for replication and for erasure encoding. We then use these bounds to derive a probabilistic model that determines how long nodes can remain disconnected from the system before repair is required. This model is parameterized by a desired durability guarantee, expressed as a probability.

2.1 Detecting Permanent Failures

Differentiating permanent and transient failures is necessarily an estimation. While a node is disconnected, the system is of course unable to determine whether the node's departure is permanent. Instead, the system must wait and declare a departed node as *failed* (i.e., permanently failed) only after some period of communication silence from the node. We define τ as the period of time the system waits before declaring a silent node as *failed*. Increasing τ decreases the chance that a transient departure is mislabelled as permanent and can reduce the frequency of repairs; but too large a value can jeopardize durability.

As we will see later, τ depends on node lifetime. In a system were typical lifetimes and transient-departure duration are on the same time scale, it is difficult to offer a strong durability guarantee and a large value for τ. In this situation, we can improve failure detection accuracy without decreasing τ by requiring that all current and recently departed nodes send heartbeats with period τ. For nodes that have departed because they are idle and thus powered off, the power-management system of modern PCs support preprogrammed timed wakeup suited to this purpose. On the other hand, if the system can be configured with a large τ this heartbeat approach is unnecessary, because most nodes will reconnect within τ on their own. While the range of τ is workload dependent, our evaluation in Section 4.1 indicates that sufficiently large values can be achieved in practice.

2.2 Maintaining Redundancy

Since increasing τ increases the efficiency of the system, our goal is to determine τ_{max}, the maximal value of τ that does not jeopardize durability. To achieve this goal we must first determine the largest number of failures the system can survive, even continuously, in a period of length τ. In doing so, we must account for the delay incurred to detect failure and for the duration of the repair process. We must also account for the fact that the data needed to make the repair may not be available at the time the repair is triggered; a problem not faced by approaches that ensure availability. Our goal is thus to compute the largest τ such that we avoid scenarios where all of the nodes needed to restore a data item fail within any period of length τ.

The first step is to determine the maximum number of failures the system can sustain in any period of length τ. To do this, we divide time into epochs of length τ such that all non-failed nodes will be available at least once in every epoch. As a result, a node that fails during epoch ε_i can be declared as *failed* in epoch ε_{i+1}. Similarly, a repair triggered in epoch ε_{i+1} can finish by epoch ε_{i+2}, because the repair process need not wait more than τ for all required nodes to become available. The remainder of the calculation differs for replication and for erasure encoding.

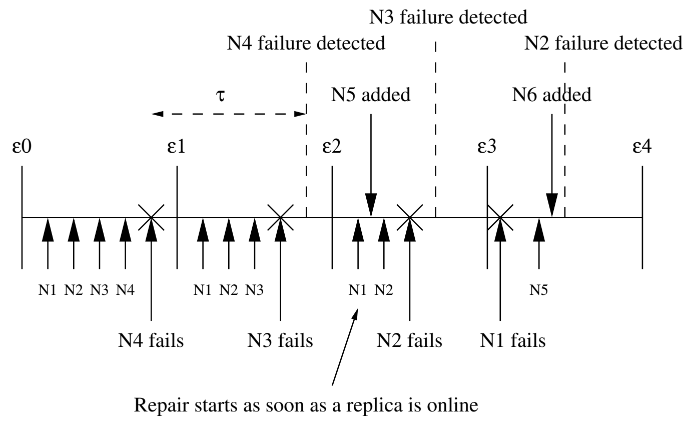

Figure 1: **Failure scenario for replication with $k = 4$**

2.2.1 *Replication*

For *replication*, a repair is triggered as soon as a node is labelled as *failed* by the system. This approach makes sense for replication, because, unlike erasure encoding, the repair cost is strictly proportional to the number of replicas to be restored.

Thus, if k replicas exist at the beginning of an epoch, there will always remain at least one replica at all times, as long as at most $\lceil \frac{k}{3} \rceil - 1$ nodes fail in any epoch, even when these failures are continuous. This property holds because replicas that fail during epoch ε_i are replaced during epoch ε_{i+2} at the latest. This fact ensures they are at least $\lceil \frac{k}{3} \rceil$ replicas at the beginning of any epoch, making it possible to sustain this failure rate indefinitely. Figure 1 shows a failure scenario for $k = 4$. Arrows represent instants where nodes are connected to the system. Repairs are triggered as soon as failures are detected but can only start when one of the replica connects to the system. For $k = 4$, as long as no more than one replica fails in an epoch, there always remain at least one replica.

2.2.2 *Erasure Encoding*

For *erasure encoding*, we follow a lazy repair approach similar to that found in Total Recall and Oceanstore [6]. We delay repair until redundancy falls below a *threshold* in order to realize the cost-amortizing benefits inherent in this approach. To determine τ_{max} for erasure codes, we have to determine two values: (1) the repair threshold and (2) the maximum number of nodes that can fail in an epoch.

Using f_{max} as the maximum number of failures in an epoch, we first determine the repair threshold. We start with n fragments stored on n nodes at the beginning of epoch ε_i and with m representing the minimal number of fragments necessary to retrieve a file. Since a repair can take an epoch to complete, the threshold is at least $m + f_{max}$. Because failures are only detected an epoch later, there can be as much as a f_{max} discrepancy between the number of nodes assumed to have failed and the actual number of failed nodes. This fact raises the repair threshold to $m + 2f_{max}$.

The value of f_{max} is a design parameter and its range is $[1, \frac{n-m}{2}]$ since $m + 2f_{max} \leq n$. A large value allows nodes to be disconnected for a longer time but raises the value of the repair threshold, making repairs more frequent and less efficient. A smaller value of f_{max} lowers the threshold but also lowers the bound on how long nodes can remain disconnected which can also increase the frequency of repairs.

2.2.3 *Durability Guarantee*

We can now define the probability distribution for the number of nodes that can fail in a period of length τ. We use this distribution to evaluate τ_{max} for both redundancy mechanisms. We make the assumptions that nodes are independent and that their lifetime is exponentially distributed. The assumption of independence has been shown to be valid in certain peer-to-peer systems [1]. In the case where this assumption is invalid, techniques exists to select nodes with a low level of correlation [10].

Although not perfect, the assumption of exponentially distributed lifetimes is a useful approximation. The memoryless property of the exponential distribution allows us to formulate our equation for durability in a simple manner. Ideally we would like to infer a probability distribution based on past failures; this is left for future work.

The probability of a node failing during a period of length τ is $1 - e^{-\mu\tau}$ where $1/\mu$ is the node's expected lifetime. The probability of more than j out of l nodes failing during an epoch is given by the following equation:

$$P = \sum_{i=j}^{l} \binom{l}{i} (1 - e^{-\mu\tau})^i e^{-(l-i)\mu\tau} \qquad (1)$$

We know from the previous two subsections that $j = \lceil \frac{k}{3} \rceil - 1$ for replication and $j = f_{max}$ for erasure encoding. Since the failure of more than j nodes during an epoch could potentially endanger the durability of the data, we have to ensure that it occurs with very low probability. By making $P < N^{-c}$

where $N = 10$ and c represents the number of 9's and by choosing the value of l that maximizes P, we are able to offer guarantees in terms of durability.

For replication, P is maximal when $l = k$. Given k, μ and the desired value for c, we can numerically solve for τ. For erasure encoding, P is largest when $l = n$. We can relax this constraint by observing that the number of nodes failing in an epoch only matters when $m + 2f_{max}$ or fewer nodes remain. We can set the beginning of epoch ε_0 at the $n - (m + 2f_{max})^{th}$ failure and use $l = m + 2f_{max}$.

Figure 2 and 3 provide values for τ given a replication factor k for replication and given f_{max} and m for erasure encoding (Figure 3 is for $m = 32$).

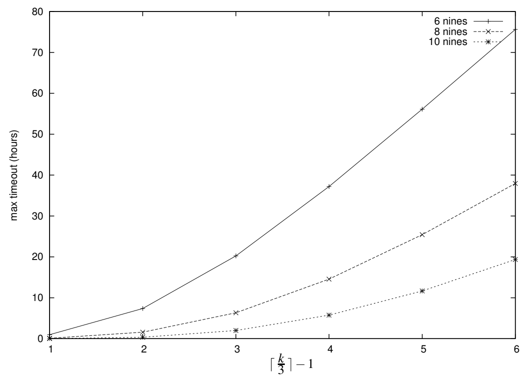

Figure 2: **τ as a function of k for #9's ($1/\mu$ = 100 days)**

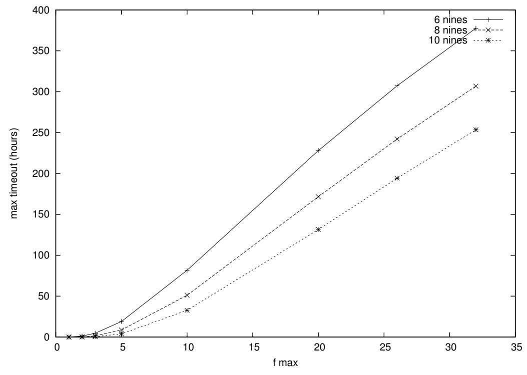

Figure 3: **τ as a function of f_{max} for #9's ($1/\mu$ = 100 days)**

3. SYSTEM ARCHITECTURE

The design of our redundancy maintenance mechanism is based on the Total Recall block store [3]. We give here a brief overview of the Total Recall block storage layer and describe the key modifications needed for our redundancy maintenance mechanism.

Figure 4 illustrates the architecture of our modified Total Recall block storage layer. Each file stored in the system is associated with a unique ID. A file's ID is used to map a file to its *master* node. The master node ensures the consistency

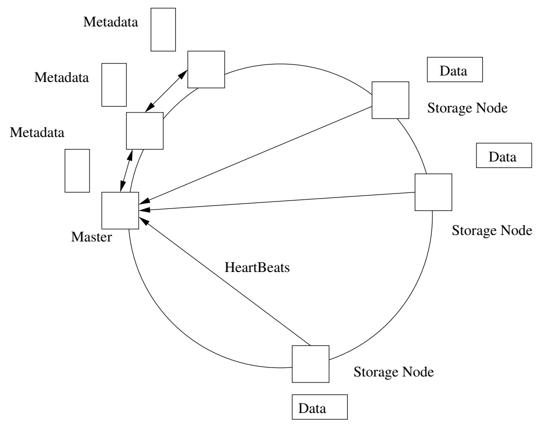

Figure 4: **Total Recall Architecture**

and monitors the redundancy of files under its responsibility. A file's data is not stored on its master node, but is stored instead on *storage* nodes. In Total Recall, every node participating in the system is both a master node for some files and a storage node for others. The master node only maintains the file's metadata. A file's metadata contains the list of storage nodes that store the file's data.

The metadata is stored on the master and replicated on the master's *successors*[8]. The metadata redundancy is maintained similarly to the way that DHash[5] maintains block's replicas. As soon as a departure is detected, the metadata is replicated on another successor node. This step ensures that the metadata is always available. When a master fails, another node can immediately take over. Storage nodes can find their current master at any time by using the routing mechanism provided by the underlying distributed hash table. Although, a priori it seems expensive to constantly maintain the replication of the metadata as nodes join and leave the system, results seems to show otherwise. Less than 1% of the maintenance bandwidth in Total Recall is used to maintain the metadata.

To keep track of when storage nodes were last heard of, we extend the metadata with a timestamp for each node present in the metadata. These timestamps are used to determine which storage nodes have failed. Unlike Total Recall, masters passively monitor their storage nodes. By passively we mean that a master does not probe each of its storage nodes but instead storage nodes periodically send heartbeat messages to their master when connected to the system. The metadata is updated every time a heartbeat is received. Since τ is usually in the order of hours, heartbeats are not required to be as frequent as in Total Recall. We use a period of 30 minutes; by comparison, Total Recall storage nodes are "pinged" every 30 seconds. In our case, what is really important is to capture the presence of every node, regardless of how long that presence exists, as any presence represents a proof that a node hasn't failed. It makes sense in this case for storage nodes to send heartbeats to their master instead of the other way around. Storage nodes notify their master as soon as they connect and then do so periodically.

When the timestamp for a storage node is older than τ, the master declares the nodes as *failed*. Depending on the redundancy mechanism and on the number of failed nodes, the master may trigger a repair if necessary. Our current design is meant to operate in a non-malicious environment since we assume that failures are fail-stop.

4. RESULTS

This section presents preliminary results obtained from simulation. The simulator models a simplified version of the data storage architecture presented in Section 3. The redundancy maintenance mechanisms for both replication and erasure codes are implemented. The goal of the simulation is to estimate the network bandwidth used by our durability-based approach and to compare it to an approach based on availability. We also evaluate the impact of running the system with the timer mechanism described in section 2.1.

To drive the simulator we use a synthetic trace based on an Overnet host availability trace [1]. We used synthetic data instead of the real trace because of the need to run experiments for long periods of simulated time since our approach is probabilistic. Simulations to measure bandwidth were run for a period of 1 year of simulated time and we also ran experiments for extended period of time (20 years) without ever losing data. The lifetime of nodes in the system is modelled as a Poisson process similar to the one described in [7]. The mean lifetime is 5 days since 20% of the nodes permanently leave the system every day. Individual node's period of connectivity and disconnectivity are exponentially distributed with means obtained from the trace. Nodes are online on average for 2.5 hours before being disconnected on average for 3 hours. When simulating the system using the timeout mechanism, we simply bound the intervals nodes remain disconnected to τ. This ensure that nodes that haven't failed always reconnect in time.

The results presented here do not take into account hosts' bandwidth limitation. We assume that as soon as a block is available it can be retrieved. The reason behind this assumption is that availability not bandwidth is the main factor influencing the duration of repairs. We currently use values for τ that are around 8 hours. Assuming broadband hosts, this leaves enough time to retrieve and repair large files (100's of MB).

4.1 Durability vs Availability

We first compare our approach to the availability-based approach used in Total Recall. Total Recall repairs a file as soon as its available redundancy factor falls below 2. We also look at the optimal case for availability, where a file is repaired when its available redundancy factor reaches 1, the minimum redundancy factor needed to be able to reconstruct the file. For our durability-based approach, the values in Table 1 were used. These values were obtain by numerically solving equation (1) for τ. We choose values around 8 hours for τ and set f_{max} to the lowest value that meets the target durability. Since the mean disconnected period is just over 3 hours, the *wakeup timer* described in Section 2.1 is seldom used.

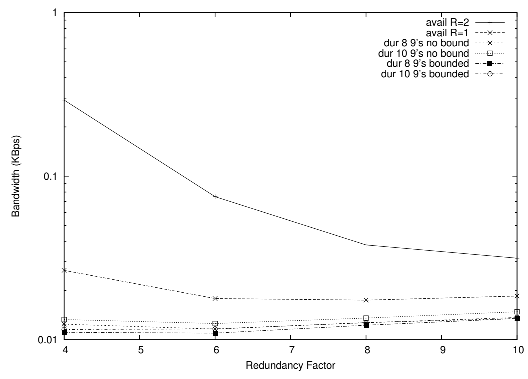

Figure 5: Bandwidth to maintain a 1 MB file split in 32 fragments

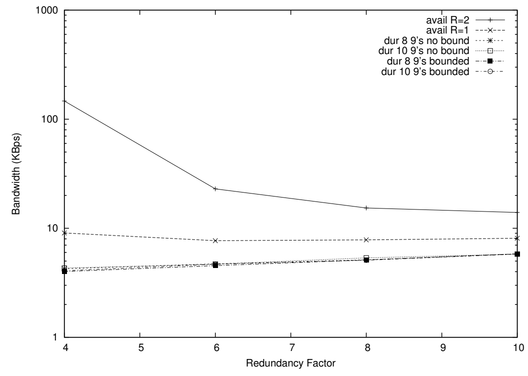

Figure 6: Bandwidth to maintain a 512 MB file split in 256 fragments

Our evaluation is based on erasure encoding and measures the cost of maintaining a 1 MB file split into 32 fragments and a 512 MB file split into 256 fragments. The fragments are encoded to generate $m * R$ fragments where R is the redundancy factor. Figure 5 and 6 compare both approaches for different redundancy factors.

The results show that our durability-based approach is significantly cheaper than the availability based approaches. While availability performance improves as the redundancy factor increases, it never reaches the level of our durability-based approach. More importantly, the performance of the durability-based approach remains constant, meaning that the durability-based approach doesn't need a large redundancy factor to work well.

Finally, observe that there is minimal differences between the *bounded* (using the wakeup timer) and the *unbounded*

	1 MB		512 MB	
# 9s	τ(hours)	f_{max}	τ(hours)	f_{max}
6	8.5	16	8.25	48
8	7.4	18	8.2	54
10	7.6	22	8.2	60

Table 1: Values for τ and f_{max}

approach. This confirms our intuition that choosing a large enough value for τ can minimize the utility of the wakeup timer.

4.2 Effect of the Wakeup Timer

To measure the effect of the wakeup timer, we reran the first benchmark using a value of three hours for τ, instead of seven. Figure 7 shows the results. The smaller value of τ forces the *unbounded* system to declare many transient departures as permanent ones, triggering repair more frequently. The *bounded* system is not affected since its performance is approximately the same as in the first set of experiments. This shows that the wakeup timer is effective in helping the system differentiating between transient and permanent failures.

Unfortunately due to space considerations, we are unable to show our results for replication. These results would show that our durability-based approach is much cheaper than current approaches based on availability.

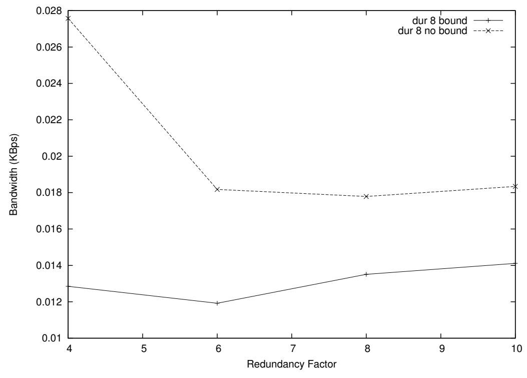

Figure 7: **Effect of using the wakeup timer**

5. CONCLUSION

Traditionally, reliability has typically been equated with availability. We believe, however, that reliability's two key components — *availability* and *durability* — are best considered separately. By separating these complimentary goals, we are able to deliver durability at a substantially lower cost than when coupled with availability. As a result, a system can provide very strong durability guarantees without incurring the substantial overhead of providing an equally strong guarantee of availability. As users are typically less demanding of availability than durability, this approach provides a useful optimization.

Along with the current parameters used to characterize availability such as the redundancy factor, we present new parameters that characterize the durability of redundant data. Our results are currently based on simulation but we are working on an implementation that will enable us to perform experiments on facilities such as Planet Lab or in an emulated network environment. An interesting problem that remains is how to build reliable failure prediction based on past history instead of solely relying on a probability distribution.

6. REFERENCES

[1] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proc. of IPTPS '03*, 2003.

[2] R. Bhagwan, S. Savage, and G. M. Voelker. Replication strategies for highly available peer-to-peer storage system. technical report cs2002-0726, ucsd, 2002.

[3] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G.M. Voelker. Total recall: System support for automated availability management. In *First Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[4] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *HotOS IX*, 2003.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *In Proc. of SOSP*, 2001.

[6] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[7] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proc. of PODC*, 2002.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001.

[9] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS '02*, 2002.

[10] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proceedings of International Workshop on Reliable Peer-to-Peer Distributed Systems*, pages 362–367, October 2002.