

On Object Maintenance in Peer-to-Peer Systems

Kiran Tati and Geoffrey M. Voelker
Department of Computer Science and Engineering
University of California, San Diego

1. INTRODUCTION

Storage is often a fundamental service provided by peer-to-peer systems, where the system stores data objects on behalf of higher-level services, applications, and users. A primary challenge in peer-to-peer storage systems is to efficiently maintain object availability and reliability in the face of node churn. Nodes in peer-to-peer systems exhibit both temporary and permanent failures, requiring the use of redundancy to mask and cope with such failures (e.g., [1, 4, 10, 16, 21]). The cost of redundancy, however, is additional storage and bandwidth for creating and repairing stored data.

Since bandwidth is typically a much more scarce resource than storage in peer-to-peer systems, strategies for efficiently maintaining objects focus on reducing the bandwidth overhead of managing redundancy, trading off storage as a result. Typically, these strategies create redundant versions of object data using either replication or erasure coding as redundancy mechanisms, and either react to node failures immediately or lazily as a repair policy.

In this paper, we revisit object maintenance in peer-to-peer systems, focusing on how temporary and permanent churn impact the overheads associated with object maintenance. We have a number of goals: to highlight how different environments exhibit different degrees of temporary and permanent churn; to provide further insight into how churn in different environments affects the tuning of object maintenance strategies; and to examine how object maintenance and churn interact with other constraints such as storage capacity. When possible, we highlight behavior independent of particular object maintenance strategies. When an issue depends on a particular strategy, though, we explore it in the context of a strategy in essence similar to TotalRecall [4], which uses erasure coding, lazy repair of data blocks, and random indirect placement (we also assume that repairs incorporate remaining blocks rather than regenerating redundancy from scratch).¹

Overall, we emphasize that the degrees of both temporary and permanent churn depend heavily on the environment of the hosts comprising the system. Previous work has highlighted how ranges of churn affect object lookup algorithms [14]; in this paper, we explore how these differences impact the source of overheads for object maintenance strategies. In environments with low permanent churn, object maintenance strategies incur much of their overhead when initially storing object data to account for temporary churn. In environments with high permanent churn, however, object

maintenance strategies incur most of their overhead dealing with repairs — even if the system experiences high temporary churn. Finally, we highlight additional practical issues that object maintenance strategies must face, in particular dealing with storage capacity constraints. Random placement, for example, unbalances storage load in proportion to the distribution of node uptimes, with both positive and negative consequences.

2. CHURN

Peer-to-peer systems experience churn as a result of a combination of temporary and permanent failures. A temporary failure occurs when a node departs the system for a period of time and then comes back. Any data stored on the node becomes unavailable during this period, but is not permanently lost. Examples of temporary failures are when home users login to systems in the evening, or when business users use systems during the day but logoff overnight. A permanent failure corresponds to a loss of data on a node, such as when a disk or machine fails, or when a user leaves a file sharing system permanently. Temporary failures directly impact availability, and permanent failures directly impact reliability.

The degrees of both temporary and permanent churn depend heavily on the environment of the hosts comprising the system. Systems incorporating home and business hosts tend to experience much higher levels of churn than systems incorporating server hosts maintained in machine rooms. For example, Table 1 illustrates the churn characteristics taken from traces of three different host populations, the Overnet file sharing system [3], the PlanetLab testbed [17], and hosts in a large corporation [6].

The observation that different environments experience different degrees of churn is not new, although characterizations of churn tend to focus just on temporary churn (e.g., [14]). Characterizing permanent churn in deployed systems remains an open question, in part because doing so requires long-term measurement as well as assumptions about node behavior; deciding that a node has left permanently within a finite trace essentially requires a threshold for assuming that observing that a node has left the system means that it has left permanently. For the Overnet trace, we consider host departures where the host leaves for more than six days a permanent failure; all other host departures are temporary failures. Given the short period of the trace, using a larger threshold would result in little permanent churn. As a result, we consider this threshold an upper bound on permanent churn for this population. For the PlanetLab trace, we consider host departures where the host leaves for more than thirty days a permanent failure; all other host departures are temporary failures. For the FarSite study, we use numbers

¹We choose one strategy to be illustrative more than to advocate a particular approach, and choose this strategy because of familiarity; the tradeoffs between replication and erasure coding, for example, have been well studied [2, 5, 12, 15, 19], and each has its strengths and weaknesses.

System	Start Date	Duration	Total Nodes	Ave Nodes Per Day	Temporary Failures		Permanent Failures	
					Total	Per Day (Host)	Total	Per Day (Host)
Overnet	Jan 15th, 03	7 days	1469	1028	33084	4736 (4.61)	107	107 (0.104)
PlanetLab	Apr 1st, 04	406 days	655	318	13633	34 (0.11)	593	1.6 (0.005)
FarSite	July 1st, 99	35 days	60000	45000	87500	2500 (0.05)	7000	200 (0.004)

Table 1: Churn in representative systems.

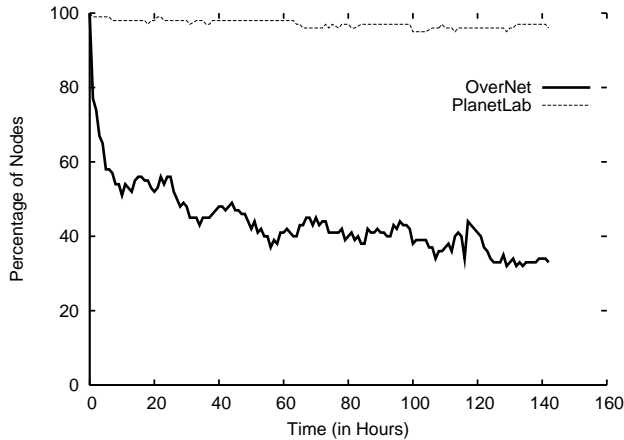


Figure 1: Tracking node availability among a set of nodes over time. The monitored set of nodes are those nodes that were in the system at 24 hours into the trace.

reported in the paper.

Comparing the churn in the different environments, we see that the environments have very different degrees of both temporary and permanent churn. We normalize the metrics per day since user and node behavior tends to be diurnal. The “Ave Nodes Per Day” metric is the number of nodes in a system per day, averaged across all days in the trace. Normalized per day per host in the system, the file sharing trace exhibits an order of magnitude more temporary and permanent churn than the other environments. The wide-area population exhibits twice as much temporary churn as the corporate population, with roughly equivalent permanent churn.

Another way to compare churn in different environments is to consider the impact of churn from the perspective of object maintenance strategies. Systems maintain objects by storing redundant versions of object data among multiple nodes. As a result, the behavior of a maintenance strategy depends on the churn of a set of nodes selected at a particular point in time, such as when the object is initially stored. Figure 1 shows the effects of churn on a fixed set of nodes over time in the Overnet and PlanetLab traces. We examined all nodes in each trace that were available 24 hours into the trace (530 for Overnet, 300 for PlanetLab), and tracked their availability over six days. The graph plots the percentage of nodes in the original set that are available in the system over time.

The Overnet set of nodes exhibits a dramatic drop in availability in the first 24 hours; this drop is due to the high temporary churn in the environment, particularly nodes with short uptimes. Both the Overnet and PlanetLab groups of nodes experience daily variations in availability, also due to temporary churn. Both systems also exhibit a slow decay in node availability over time. This slow decay is due to permanent node failures slowly reducing the original set of

nodes.

Sections 3 and 4 discuss the consequences of temporary and permanent churn in different environments in more detail. By comparing the characteristics of churn in different environments, we want to emphasize that environment matters. As we discuss in more detail later in the paper, differences in environment impact the focus of object maintenance strategies.

3. TEMPORARY CHURN

In this section we focus on the approach of using redundancy to handle temporary churn. Since our goal is to provide insight into the problem, rather than advocate a particular algorithm, we make some simplifying assumptions to highlight temporary churn issues. In particular, in this section we assume that the node population has no permanent churn, only temporary churn, and that a node’s availability characteristics do not significantly vary during its lifetime. Of course, in an actual system these assumptions are not realistic: any population experiences permanent churn, and node availability varies over time. Such shifts in node availability over time change the steady-state dynamics of the group of nodes storing object data. A maintenance strategy can detect and adapt to these changes over longer time scales.

Consider the events that occur immediately after storing object data in a system with only temporary churn. A maintenance strategy selects (typically randomly) a set of nodes on which to store the blocks comprising an object. One characteristic of these nodes is that they are all available at the time of object placement. These nodes, however, vary both in their uptime durations as well as how long they have been active in their current session. As a result, over time a fraction of these nodes will become unavailable due to temporary churn. Eventually, though, the number of simultaneously available nodes will stabilize in a diurnal pattern as nodes depart and arrive on a daily basis.

Given this behavior, a maintenance strategy can create sufficient redundancy to sustain the availability of an object on the minimum set of available nodes during a day. By “minimum”, we mean that a sufficient number of nodes are available at any point in time such that the data they store is available for use; reducing the set by a node implies that at some time the data is not available. For strategies that use replication, unavailability occurs when all replicas are simultaneously unavailable; for those that use erasure coding, it occurs when an insufficient number of nodes are available to reconstruct object data. An object maintenance strategy can proactively estimate this amount of redundancy when initially storing the object (e.g., based on past behavior). Or, it can reactively add redundancy as nodes temporarily depart the system until the amount of redundancy is sufficient to mask temporary failures. Either way, eventually the set of nodes storing object data for a particular file will stabilize into a random process where a number of simultaneously available nodes storing data is sufficient to maintain object

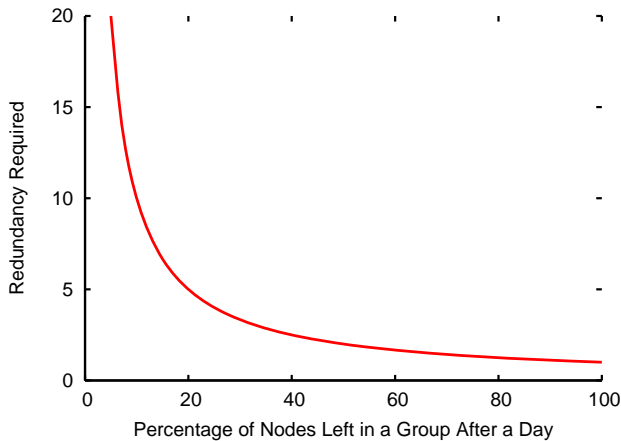


Figure 2: Relationship between temporary churn and redundancy (storage overhead) required to mask it.

availability with high probability (although which nodes are simultaneously available varies over time). We call this state “masking” temporary churn, where an object maintenance strategy is using sufficient redundancy such that temporary churn will not induce repairs (in our idealized model), and result in only infrequent repairs in practice (due to changing node availabilities, etc.).

We also note that the data availability these nodes provide is probabilistic. Object maintenance strategies estimate the amount of redundancy required to provide object availability with a high probability based upon node availability characteristics (e.g., [7], [4]). If nodes in the system experience a sudden shift in availability (e.g., catastrophic simultaneous failures), the probabilistic availability guarantees will not hold. We also note that placement strategies typically assume that failures are not correlated. There are clear examples when failures are correlated (e.g., [9, 20]); however, even with noticeable diurnal patterns, correlation coefficients of node availabilities indicate that correlation of nodes in the system is not strong [3].

Figure 2 illustrates the relationship between temporary churn and the amount of redundancy required to mask it. By amount of redundancy, we mean the storage overhead used by a redundancy technique such as replication or erasure coding; a redundancy of three, for example, means that the storage overhead is three times the file size. For a given group of nodes storing object data, the x-axis varies the percentage of nodes remaining available in the group after one day has passed since storing the object in the system. The y-axis shows the degree of redundancy required to keep the object available. Again, we focus only on temporary churn and assume that no nodes fail permanently. At a high level, it is a straightforward inverse relationship. For example, if any 50% of the original group of nodes are available in steady state and node arrivals are uniformly distributed, then the maintenance strategy will need to store the object with a minimum redundancy factor of two to maintain object availability in the face of just temporary churn.

Once the set of nodes storing object data stabilizes, a system will not need to frequently react to node departures or create further redundancy on additional nodes. As a result, a system incurs primary bandwidth overhead for masking temporary churn when it initially places the object in the system.

With only temporary churn, no repairs are necessary. In environments with little permanent churn, tuning redundancy for masking temporary churn will have the greatest impact on minimizing bandwidth overhead. Of course, an actual system will still occasionally have to repair data redundancy due to temporary churn as, for example, node availabilities vary over time; such repairs will likely be incorporated naturally in the handling of permanent failures, discussed below.

In summary, using redundancy to deal with temporary churn has three implications: (1) an object maintenance strategy can determine a sufficient degree of redundancy to minimize repairs due to temporary churn, or “mask” temporary churn; (2) the amount of redundancy required to mask temporary churn is inversely proportional to the fraction of simultaneously available nodes storing object data; and (3) the bandwidth overhead for coping with temporary churn is dominated by object creation, not by repairs induced by temporary churn.

4. PERMANENT CHURN

Permanent churn drives repairs. When the system permanently loses nodes storing redundant object data, the system must eventually repair the redundancy to ensure data reliability. The frequency with which the system repairs object data depends on the degree of permanent churn and the amount of redundancy restored during repair for long-lived objects. In environments with substantial permanent churn, like those that incorporate business and home nodes, the overhead for repairing long-lived object data dominates the overhead of establishing sufficient redundancy to mask temporary failures. As a result, in these environments tuning repair strategies to deal with permanent churn will have the greatest impact on minimizing bandwidth overhead.

When a system decides to repair object data, it must decide how much redundancy to restore. The more redundancy a system restores during a repair the longer it can delay the next repair, thereby trading off storage to reduce the frequency of repairs. In terms of bandwidth overhead, though, it is not immediately clear what the best choice is. An object maintenance strategy can either make “smaller” repairs more frequently, or “larger” repairs less frequently.

The choice depends upon the distribution of permanent node failures. We show that there exists an optimal balance between the amount of redundancy restored at each repair and the frequency of repair under the following model. Assume that the object maintenance strategy uses erasure coding and lazy repair [4], and that a repair replenishes any remaining data with new redundant data to maintain reliability (as opposed to regenerating it from scratch). Let x be the threshold at which the system triggers repair in terms of the number of nodes storing object data. An object maintenance strategy will restore redundancy by creating new coded blocks of data on N additional nodes (encoding with a large encoding graph enables the creation of incremental encoded blocks over time to supply repairs). Immediately after a repair, an object has blocks stored on $x + N$ nodes. Since the repair threshold is x nodes, from one repair to the next N nodes will fail permanently. This process takes $\frac{2Nd}{N+x}$ time, where d is the average rate of permanent failures measured in terms of half death time (similar to half life time [11]), the amount of time it takes for half of the nodes to fail permanently; if we have N nodes, then it takes d time for $N/2$ nodes to fail permanently.

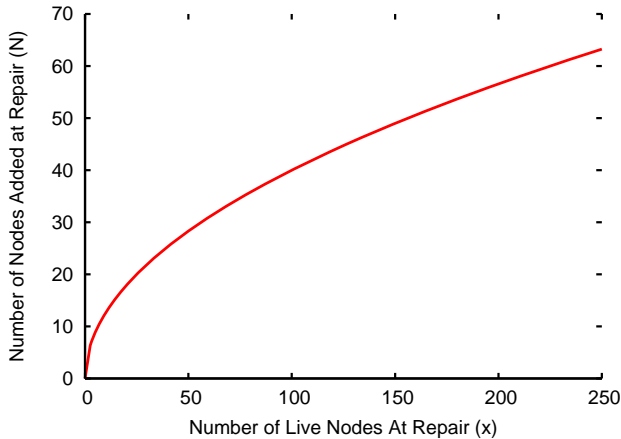


Figure 3: Optimal bandwidth required to mask permanent churn depending on degree of temporary churn (number of nodes required to mask temporary failures).

Our goal is to minimize bandwidth requirements for performing repairs. When using erasure coding, for example, the system must first read the object. Doing so requires f bytes (i.e., f is the object size).² We also store new encoded blocks on N new nodes, requiring another Nf/a bytes, where a is the number of nodes an object gets fragmented onto. Overall each repair consumes $f + Nf/a$ bytes. The total bandwidth needed for a repair is $\frac{f(1+N/a)(x+N)}{2Nd}$ bytes per second, averaged over the interval of time between repairs. The minimum value occurs at \sqrt{ax} and is $\frac{f(1+\sqrt{x/a})^2}{2d}$. A system will typically keep a , object fragmentation, constant. The value of x , the repair threshold, will depend on the amount of redundancy needed to mask temporary churn since the object needs to be immediately available at the time of repair.

Interestingly, the amount of redundancy to restore on a repair that minimizes bandwidth overhead depends upon the degree of temporary churn in the system, but not on the degree of permanent churn; the bandwidth overhead certainly scales with the rate of permanent churn, but the rate does not affect the choice of how much redundancy to repair. Figure 3 illustrates the relationship between the temporary churn experienced by a system and the amount of redundancy to restore on a repair that minimizes repair bandwidth overhead. As an example parameterization, we assume that files have a uniform size (f) of 1 MB, the repair threshold is twice the redundancy required to mask temporary churn, each file is fragmented (a) into 16 blocks, and the nodes in the system permanently fail (d) according to the Overnet trace (which contains both temporary and permanent churn). The x-axis shows the repair threshold (x) in terms of the number of nodes remaining that are storing object data; again, think of x as the number of nodes (amount of redundancy) needed to mask temporary churn. The y-axis shows the amount of redundancy restored on each repair that minimizes repair overhead.

²Optimizations are possible, such as storing a full replica at one node to eliminate the read [15], although we note that such optimizations increase storage cost and may not be practical for very large objects. We could modify our analysis to incorporate them, but our goal is to understand the trends more than absolute overheads.

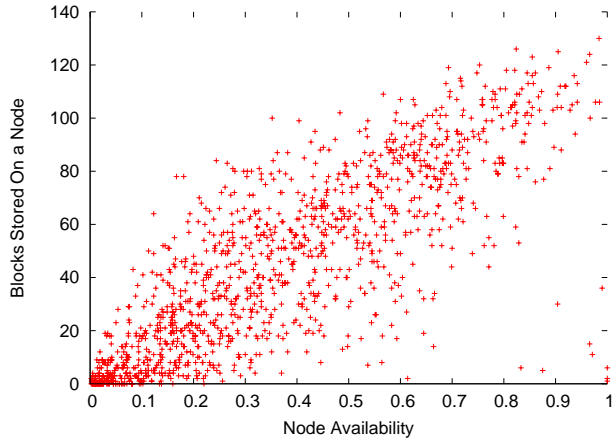


Figure 4: Node's used storage capacity with respect to node's availability.

We also simulated an object maintenance strategy with the characteristics discussed above. We varied the amount of redundancy added after each repair. The results indicate that repairs should only restore a small amount of redundancy as expected from the above analysis (consistent with similar results for a system model focused on permanent churn [13]). With a repair threshold of 32 nodes (redundancy factor of two), a repair should optimally generate 20 new blocks on nodes (restore a redundancy factor of 1.25) in addition to the 32 nodes remaining in the system.

5. CAPACITY CONSTRAINTS

A primary goal of object maintenance strategies is to reduce the bandwidth overhead of making data available and reliable in the face of churn. The strategies tradeoff storage to achieve these goals, but they typically still strive to be storage-efficient. Previous work has evaluated the insertion failures rates of peer-to-peer storage systems as the system reaches capacity [8]. Even so, the constraints of both system and node storage capacity on object maintenance strategies and their overheads have not been given much attention, particularly as node availability and churn varies. In this last section, we motivate the need for maintenance strategies to also consider the constraints of capacity.

Object maintenance strategies that use indirect block placement with lazy repair, as in TotalRecall [4], randomly place object data on nodes in the system. A consequence of random placement is that it unbalances storage load in proportion to the distribution of node uptimes. To illustrate this effect, we simulated placing 1,024 1-MB objects into a system of 2,000 nodes paced evenly throughout a day. We then measured the number blocks each node stores at the end of the day when using a redundancy factor of three to store objects. We used the Overnet trace to simulate node arrivals and departures and determine node uptimes; the effect is similar in other environments, although the distribution of uptimes will change. Figure 4 shows the results of this experiment in a scatter plot. Note that each object is divided into encoded blocks (the parameter a in Section 4). In this experiment we divided object into 32 blocks (if the replication factor is 3, the systems stores 96 encoded blocks for a file). For each node in the system, the graph shows the number of blocks stored on the node according to its uptime. The diagonal cluster

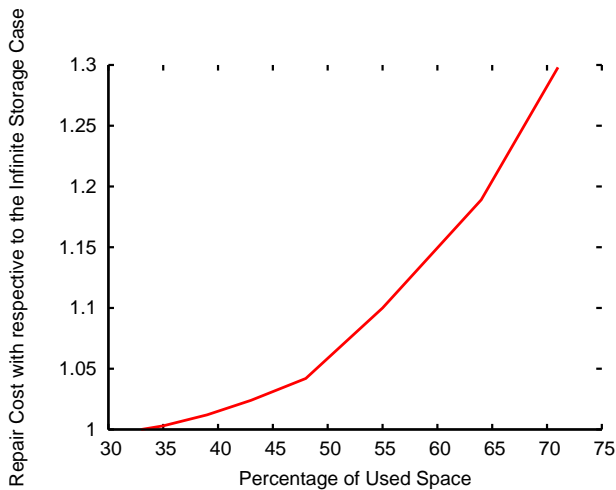


Figure 5: Normalized repair cost as a function of used capacity.

shows the correlation of node uptime and storage load (outliers, such as in the lower right corner, correspond to nodes entering the system as the simulation ends).

This effect is due to the random selection of nodes when placing object data — both when the object is initially created, as well as during repair. Nodes with longer uptimes will be selected more often than nodes with shorter uptimes and, as a result, store more blocks over time. Storage is a comparatively plentiful resource, but nodes still have finite capacities (particularly if only a small fraction of storage on a node is available for use by other nodes). Consequently, over time nodes with longer uptimes will fill to capacity faster than nodes with shorter uptimes.

This effect has both positive and negative consequences. On the positive side, it is a natural mechanism by which the system will favor storing object data on nodes with good availability. Favoring nodes with long uptimes reduces the amount of storage, and hence bandwidth overhead, required to mask temporary churn. (Note, though, that if permanent failures are independent of node uptimes, this effect does not reduce the rate of repairs.) This effect is somewhat similar to the natural formation of stable cores of supernodes in unstructured networks, also due to the bias of node uptimes [18].

On the negative side, object maintenance strategies will need to explicitly respect node storage capacities when making placement decisions. Strategies that use indirect placement can adapt to storage capacity constraints by simply removing nodes at capacity from random selection. Respecting capacities has two consequences: (1) since indirect placement biases towards nodes with higher uptimes, data for newer objects gets placed on nodes with lower uptimes; and as a result, (2) repair overhead increases more than linearly as system storage grows towards capacity.

To illustrate this effect, we repeat the object maintenance simulation and extend it so that all nodes have equivalent capacity constraints. Figure 5 shows the results of capacity constraints on repair overhead. The y-axis shows repair overhead normalized to the situation where nodes have unlimited storage capacity. The x-axis varies capacity as a percentage of used system capacity. Below 32% capacity object maintenance is essentially unconstrained and has equivalent repair overhead as unlimited capacity. Above 32% the system places data on more nodes with lower availability,

increasing the overall repair overhead. At 65% capacity, for example, repair overhead increases by 20%.

Object maintenance strategies that eagerly repair on successors (e.g., CFS) or leaf sets (e.g., PAST) will not exhibit this bias since nodes store data relative to their position in the ID space, and not relative to uptime. Such placement implicitly assumes that nearby nodes in the ID space can always store data given to it, although in practice some nodes in the middle of a successor list, for instance, may reach capacity before other nodes. One approach to this problem is to use replica diversion to introduce a level of indirection, effectively implementing indirect placement [8]. Alternatively, successor placement can skip successors at full capacity when propagating redundant data down the successor list, effectively treating those successors as “failed” nodes with respect to placement. Doing so, however, will likely require either direct or indirect bookkeeping to track which successors store redundant object data, evolving such placement strategies from direct towards indirect placement.

6. CONCLUSION

In this paper, we revisit object maintenance in peer-to-peer systems, focusing on how temporary and permanent churn impact the overheads associated with object maintenance. Overall, we emphasize that the degrees of both temporary and permanent churn depend heavily on the environment of the hosts comprising the system. These differences impact the source of overheads for object maintenance strategies. Finally, we highlight additional practical issues object maintenance strategies must face, in particular dealing with storage capacity constraints. Experience with deployments of peer-to-peer storage systems will undoubtedly raise a number of additional practical constraints that object maintenance strategies will need to address.

REFERENCES

- [1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, and M. Theimer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of 5th OSDI Symposium*, Dec. 2002.
- [2] R. Bhagwan, S. Savage, and G. M. Voelker. Replication strategies for highly available peer-to-peer storage systems. Technical Report CS2002-0726, University of California, San Diego, 2002.
- [3] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.
- [4] R. Bhagwan, K. Tati, Y. C. Cheng, S. Savage, and G. M. Voelker. Totalrecall: System support for automated availability management. In *ACM/USENIX NSDI Symposium*, Mar. 2004.
- [5] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. of the HotOS*, May 2003.
- [6] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. of SIGMETRICS*, June 2000.
- [7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *18th ACM Symposium on Operating Systems Principles*,

Oct. 2001.

- [8] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, May 2001.
- [9] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker. the phoenix recovery system: Rebuilding from the ashes of an internet catastrophe. In *9th Workshop on Hot Topics in Operating Systems*, May 2003.
- [10] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS 2000*, Nov. 2000.
- [11] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *21st ACM Symposium on Principles of Distributed Computing*, July 2002.
- [12] J. McCaleb. <http://www.overnet.com/>.
- [13] S. Ramabhadran and J. Pasquale. Analysis of long-running replicated systems. In *Proc. of the IEEE Infocom Conference*, April 2006.
- [14] S. Rhea, D. Geels, T. Roscoe, and J. Kubiawicz. Handling churn in a dht. In *Proc. of the USENIX Annual Technical Conference*, June 2004.
- [15] R. Rodrigues and B. Liskov. High availability in dhds: Erasure coding vs. replication. In *Proc. of the IPTPS*, February 2005.
- [16] 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 ACM SIGCOMM*, Aug. 2001.
- [17] J. Stribling. http://pdos.csail.mit.edu/strib/pl_app/.
- [18] D. Stutzbach, R. Rejaie, and S. Sen. Characterizing unstructured overlay topologies in modern p2p file-sharing systems. In *Proc. IMC 2005*, June 2005.
- [19] H. Weatherspoon and J. Kubiawicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of the IPTPS*, March 2002.
- [20] H. Weatherspoon, T. Moscovitz, and J. Kubiawicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proc. of the International Workshop on Reliable Peer-to-Peer Distributed Systems*, October 2002.
- [21] J. J. Wylie, M. Bakkaloglu, V. Pandurangan, M. W. Bigrigg, S. Oguz, K. Tew, C. Williams, G. R. Ganger, and P. K. Khosla. Selecting the right data distribution scheme for a survivable storage system. Technical Report CMU-CS-01-120, Carnegie Mellon University, May 2001.