Autonomous Replication for High Availability in Unstructured P2P Systems

Francisco Matias Cuenca-Acuna, Richard P. Martin, Thu D. Nguyen
{*mcuenca, rmartin, tdnguyen*}*@cs.rutgers.edu*

Technical Report DCS-TR-509
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854

April 24, 2003

Abstract

We consider the problem of increasing the availability of shared data in peer-to-peer (P2P) systems so that users can access any content, regardless of the current subset of online peers. In particular, we seek to conservatively estimate the amount of excess storage required to achieve a practical availability of 99.9% by studying a decentralized algorithm that only depends on a modest amount of loosely synchronized global state. Our algorithm uses randomized decisions extensively together with a novel application of the Reed Solomon erasure codes to tolerate autonomous member actions as well as staleness in the loosely synchronized global state. We study the behavior of this algorithm under three distinct environments, modeled using data published from earlier studies of a corporate environment, the Napster and Gnutella file sharing communities, and a file sharing community local to students of our University. We show that while peers act autonomously, the community as a whole will reach a relatively stable configuration. We also show that space is used fairly and efficiently, delivering three nines availability at a cost of six times the storage footprint of the data collection when the average peer availability is only 24%.

1 Introduction

Peer-to-peer (P2P) computing is emerging as a powerful paradigm for sharing information across the Internet. However, we must address the problem of providing high availability for shared data if we are to move P2P computing beyond today's sharing of music and video content. For example, providing high availability is clearly critical for P2P file systems. Even for content sharing similar to today's file sharing, we must address availability for different sharing patterns. For example, when searching for scientific results or legal case studies, today's pattern of a subset of content being very popular and so naturally becoming highly available through individual's hoarding may not hold; in fact, the location of unpopular references may be critical to answering specific queries. Google's cached pages are a practical motivating example addressing this need in the context of the web: it is extremely annoying to find that a highly ranked page is not available because the server is currently down. These cached pages in effect increase the availability of web content.

At the same time, recent measurements suggest that P2P communities are fundamentally different than current servers. These measurements, e.g., [20, 2], show that members of some P2P communities may be offline more than they are online, implying that providing practical availability for shared data, say 99-99.9%, which is comparable

to what end users might expect from today's web services [14], would: (i) be prohibitively expensive storage-wise using traditional replication methods, yet (ii) requiring that data be moved (or re-replicated) as peers leave and rejoin the online community would be prohibitively expensive bandwidth-wise. For example, replicating a file 7 times when the average node availability is only 20% (as reported in [20]) would only raise its availability to 79%. Moreover, if a thousand nodes were to share 100GB (600GB after replication), the bandwidth required to keep all replicas online as members join and leave the online community would be around 3GB per node per day.

We are thus motivated to study how to improve the availability of shared data for P2P communities where individuals may be disconnected as often as, or even more often than, being online. In particular, assuming that peers eventually rejoin the online community when they go offline, we ask the question: is it possible to place replicas of shared files in such a way that, despite constant changes to the online membership, files are highly available without requiring the continual movement of replicas to members currently online? To answer this question, we propose and evaluate a distributed replication algorithm where all replication decisions are made entirely autonomously by individual members using only a small amount of loosely synchronized global state. To maximize the utility of excess storage, we assume that files are replicated in their entirety only when a member hoards that file for disconnected operation. Otherwise, files are replicated using an erasure code [24]. While the use of an erasure code is not in itself novel, we show how to use an erasure code such that the infrastructure does not need to keep track of exactly which fragment is stored where so that lost fragments can be regenerated.

We deliberately study a very weakly structured system because tight coordination is likely difficult and costly in large distributed and dynamic communities [13]. Fundamentally, our approach only depends on peers managing their individual excess storage in a fair manner and having approximate data about replica-to-peer mapping and average peer availability. Without these two types of data, a replication algorithm cannot differentiate between peers with potentially very different availability and thus may need to over-replicate to achieve the desired availability for all files. Beyond this loosely synchronized global state, all decisions are local and autonomous. Of course, one can increase the level of coordination between peers to increase efficiency of the system in utilizing storage and bandwidth. In essence, we seek to conservatively estimate the amount of storage required to provide practical availability of around 99.9%.

Finally, we also assume that disconnected operation is a critical property so that we cannot move files that members have explicitly hoarded on their local storage. These two assumptions—a weakly structured system distributed over the WAN and the criticality of disconnected operation—together with our study of a variety of different P2P environments, are the key difference between our work and that of Bolosky et al. [4], which takes a somewhat similar approach in designing and implementing Farsite [1].

Our contributions include:

- demonstrating that it is possible to increase availability of shared data to practical levels, e.g., 99.9%, using a decentralized algorithm where members operate autonomously with little dependence on the behaviors of their peers.

- showing that the prescience of a small number of highly available members can significantly reduce the replication necessary to achieve practical availability levels. In particular, our simulation results show that a community based on Saroiu et al.'s measurements [20], where the average availability is only 24% but members' availability differ widely, can achieve 99.9% availability for all files if the excess storage is at least 6 times the size of the shared data set (for brevity, we refer to this as 6X excess storage), whereas a community with average availability of 33%, but where all peers look alike, requires 9X or more excess storage to achieve the same availability.

- presenting the detailed design of one particular decentralized and autonomous replication algorithm. In particular, our scheme does not build or maintain any distributed data-structures such as a hash table. Instead, our approach relies only on peers periodically gossiping a very compact content index and availability information. As just mentioned, our simulation results show that in a community where average peer availability is only 24%, we can achieve 99.8% minimum file availability at 6X excess storage. For a community with a higher average peer availability of 81%, we can achieve 99.8% minimum availability with only 2X excess storage capacity.

The remainder of the paper is organized as follows. In Section 2, we first briefly describe PlanetP [6, 5], a gossiping-based publish/subscribe infrastructure, since we assume its use for maintaining the loosely synchronized global data needed for our algorithm. Section 3 describes our replication algorithm and considers the potential impact of faulty peers. While we argue that in most cases, false information provided by a small subset of faulty users should have little impact, it is important to note that we rely on the higher-level application that will use our infrastructure to address security issues. Section 4 presents our simulation results. Section 5 discusses related work. Finally, Section 6 concludes the paper with a discussion of several of our design decisions and a summary of the paper.

2 Background: PlanetP

PlanetP is a publish/subscribe system that supports content-based search, rank, and retrieval across a P2P community [6, 5]. In this section, we briefly discuss relevant features of PlanetP to provide context for the rest of the

paper[1].

Members of a PlanetP community publish documents to PlanetP when they wish to share these documents with the community. Typically, the publication of a document takes the form of giving PlanetP a small XML snippet containing a pointer to a local file, which is left in place by PlanetP. PlanetP indexes the XML snippet and the file, maintaining a detailed inverted index local to each peer to support content search of that peer's published documents. (We call this the *local index*.)

Beyond the local indexing of content, PlanetP implements two major components to enable community-wide sharing: (1) an infrastructural gossiping layer that enables the replication of shared data structures across groups of peers, and (2) a content search, ranking, and retrieval service. The latter requires two data structures to be replicated on every peer: a membership directory and a content index. The directory contains the names and addresses of all current members. The global content index contains term-to-peer mappings, where the mapping $t \rightarrow p$ is in the index if and only if peer p has published one or more documents containing term t. (Note that this global index is *much smaller* than the local indexes maintained only locally to each member [6].) The global index is currently implemented as a set of Bloom filters [3], one per peer that summarizes the set of terms in that peer's local index. All members agree to periodically gossip about changes to keep these shared data structures weakly consistent.

To locate content, users pose queries to PlanetP that can be set persistent in order to function as a publish/subscribe system. To answer a query posed at a specific node, PlanetP uses the local copy of the global content index to identify the subset of target peers that contain terms relevant to the query and passes the query to these peers. The targets evaluate the query against their local indexes and return URLs for relevant documents to the querier. PlanetP can contact all targets in order to retrieve an exhaustive list or a ranked subset to retrieve only the most relevant documents.

Using simulation and measurements obtained from a prototype, we have shown that PlanetP can easily scale to community sizes of several thousands [6]. (We have also outlined how PlanetP might be scaled to much larger sizes than this initial target.) Consistent with findings by Vogels et al. [22], we found gossiping to be highly tolerant of dynamic changes to the communicating group, and so is highly reliable in the context of P2P systems. The time required to propagate updates, which determines the window of inaccuracy in peers' local copies of the global state, is on order of several minutes for communities of several thousand members when the gossiping interval is 30 seconds. Critically, PlanetP dynamically throttles the gossiping rate so that the bandwidth used in the absence of changes quickly becomes negligible.

[1]Although we have assumed PlanetP as the underlying infrastructure for this study, one could also use infrastructures based on distributed hash tables to the same effects [25, 19, 21, 16]. The main advantage of using PlanetP is that updates to the global state are automatically propagated, where groups of updates occurring close together are batched for efficiency. If we instead use a DHT-based system, we would have to poll nodes periodically for updates even if there were no changes.

3 Autonomous Replication

We now proceed to describe our replication approach. We assume that each member of a community *hoards* some subset of the shared files entirely on their local storage; this is called the member's hoard set. Then, in our approach, each member actively pushes replicas of its hoard set to peers with excess storage using an erasure code. The main motivation for this structure is our assumption that members typically want to access a subset of the shared data when disconnected from the community; hence the hoard sets. Note, however, that our approach is not limited by this assumption. In essence, our approach assumes that each member takes responsibility for ensuring the availability of its hoarded data. In loosely organized applications such as current file sharing, such hoarding is uncoordinated and entirely driven by members' need for disconnected access. In more tightly organized applications, such as a file system, the application can implement some coordination for dividing the overall shared data set among individuals' hoard sets.

To simplify our description, we introduce a small amount of terminology. We call a member that is trying to replicate an erasure-coded fragment of a file the *replicator* and the peer that the replicator is asking to store the fragment the *target*. We call the excess storage space contributed by each member for replication its *replication store*. (Note that we do not include replication via hoarding as part of the replication store.) We assume that each file is identified by a unique ID. Finally, when we say "the availability of a fragment," we are referring to the availability of the file that the fragment is a piece of.

Given this terminology, the overall algorithm is as follows:

- Each member advertises the unique IDs of the files in its hoard set and the fragments in its replication store in the global index. Each member also advertises its average availability in the global directory.
- Each member periodically estimates the availability of its hoarded files and the fragments in its replication store.
- Periodically, say every T_r time units, each member randomly selects a file from its hoard set that is not yet at a target availability and attempts to increase its availability; the member does this by generating an additional random erasure coded fragment of the file and pushes it to a randomly chosen target.
- The target accepts and saves the incoming fragment if there is sufficient free space in its replication store. If there is insufficient space, it either rejects the replication request or ejects enough fragments to accept the new fragment. Victims are chosen using a weighted random selection process, where more highly available fragments are more likely to be chosen.

Our goal in designing this algorithm is to increase the availability of all shared files toward a common target availability while allowing peers to act completely autonomously using only a small amount of loosely synchronized global data. Given a replication store that is very large compared to the set of documents being shared, we know that this approach will work [18]. The interesting questions become what is the necessary ratio of the replication store to the size of the document set and what happens when the replication store is not sufficiently large for the community to achieve a target availability, say 99.9%, for all files. We explore these questions in Section 4.

3.1 Randomized Fragmentation and Replication

As already mentioned we use the Reed Solomon (RS) erasure coding in a novel way to support autonomous member actions. In general, erasure codes provide data redundancy at less overhead than normal replication. The idea is to divide a file into m fragments and recode them into n fragments where $m < n$. The key property of erasure codes is that a file can be reassembled from any m fragments where the aggregated size of the fragments is equal to original file size [17].

To date, given (n, m), most uses of erasure codes generate all n fragments and, over time, detect and regenerate specific lost fragments. This approach has three significant disadvantages for highly dynamic environments: (i) as the average per-member availability changes over time, files' availability will change given a fixed n; to maintain a target availability, it may thus be necessary to change n, necessitating the re-fragmenting and replication of some (perhaps all) files; (ii) an accurate accounting of which peer is storing which fragment is necessary to regenerate fragments loss due either to peers leaving the community permanently or ejection from the replication store (replacement); and (iii) it must be possible to differentiate accurately between peers temporarily going offline and leaving the community permanently to avoid introducing duplicate fragments, which reduces the effectiveness of erasure coding.

To overcome these limitations, we choose $n >> m$ but do *not* generate all n fragments. When a member decides to increase the availability of a file, it simply generates an additional *random* fragment from the set of n possible fragments. If n is sufficiently large, the chances of having duplicate fragments should be small, thus maximizing the usefulness of every fragment generated in spite of not having any peer coordination. In this manner, it is easy to dynamically adjust the number of fragments generated for each file to reflect changes in the community.

RS is particularly suited to our proposed manner of use because the cost of generating each fragment is independent of n. The fundamental idea behind RS is that a polynomial of degree $m-1$ in a Galois field $GF(2^w)$ is uniquely defined by any m points in the field. In order to create an erasure code for the blocks $D_1, ..., D_m$ of a file, we need a polynomial p such that $p(t_1) = D_1, ..., p(t_m) = D_m$. Once we have this polynomial, it is easy to create up to $2^w - m$ additional points $p(t_i) = D_i,\ i > m$ such that the polynomial can be reconstructed from any combination of m tuples from the set $\{(t_1, D_1), ..., (t_m, D_m), ..., (t_i, D_i), ...\}$. Observe that, given the polynomial, the generation of

any one fragment is independent of n as desired. According to Rizzo [17], files can be encoded/decoded on a Pentium 133Mhz at 11MB/s (we have seen similar speeds on our implementation). Moreover using w's up to 16 is quite feasible, which translates into a 0.006 probability of having collisions for the environments studied in Section 4.

3.2 Estimating the Availability of Files

Files are replicated in two manners: (i) the hoarding of entire copies, and (ii) the pushing and storing of erasure-coded file fragments. Thus, to estimate the availability of a file, we must identify the set of peers that are hoarding the file and those that are holding a fragment of the file. As already mentioned, we identify these peers using PlanetP's global index, assuming that each peer uses the index to advertise the unique IDs of hoarded files as well as files with fragments in its replication store. We also assume that peers advertise their average online and offline times in the global directory.

Given the set of peers hoarding a file f, $H(f)$, and those that contain a fragment of f, $F(f)$, and assuming that peers behaviors are not correlated (as shown in [2, 4]), the availability of f, $A(f)$, can be estimated as 1 minus the probability that all nodes in $H(f)$ are simultaneously offline *and* at least $n - m + 1$ of the nodes in $F(f)$ are also offline, where m is the number of fragments required to reconstruct f and n is redefined as $n = |F(f)|$. In general, since every peer in the system may have a different probability for being offline, say P_i for peer i, it would be too expensive to compute the exact file availability. Thus, we instead use the following approximation that uses the average probability of being offline (P_{avg}) of peers in $F(f)$:

$$A(f) = 1 - \left(\prod_{i \in H(f)} P_i \right) \times \left(\sum_{j=n-m+1}^{n} \binom{n}{j} P_{avg}^{j} \left(1 - P_{avg}\right)^{n-j} \right), \quad \textit{where} \;\; P_{avg} = \frac{1}{n} \sum_{i \in F(f)} P_i \qquad (1)$$

We use each member's advertised average online and offline times to compute the probability of the member being offline as $P_i = \frac{\textit{avg. offline time}}{(\textit{avg. online time} + \textit{avg. offline time})}$.

Note that the above equation assumes that $H(f)$ and $F(f)$ do not intersect and that all n fragments reside on different peers. We ensure this by allowing only a single fragment of a file to be stored by each peer. Also, when a peer adds a file for which it is storing a fragment to its hoard set, it immediately ejects the fragment.

Also, observe that we are assuming a closed community, where members may go offline but do not permanently leave the community. Currently, we assume that members will be dropped from the directory if they have been offline for some threshold amount of time. Thus, when members permanently leave the community, the predicted availabilities of files may become stale. Since we periodically refresh the predicted availability of files, this should not become a problem.

Finally, note that equation 1 does not account for the possibility of duplicate fragments; as already argued, however, we can make the chance of having duplicate fragments quite small and so the impact should be negligible.

3.3 Replacement

When a target peer receives a replication request, if its replication store is full, it has to decide whether to accept the incoming fragment, and if it does, select other fragments to evict from its replication store. Because we are trying to equalize the availability of all files across the system, we would like to eject fragments with the highest availability. However, if we use a deterministic algorithm, e.g., choose the fragments with the highest availability to make enough space for the incoming fragment, then multiple peers running the same algorithm autonomously may simultaneous victimize fragments of the same file, leading to drastic changes in the file's availability. Thus, we instead use a weighted random selection process, where fragments with high availability have higher chances of being selected for eviction.

Our replacement policy is as follows. We first compute the average number of nines in the availability of all fragments currently stored at the target. Then, if the incoming fragment's number of nines is above 10% of this average, we simply reject it outright. Otherwise, we use lottery scheduling [23] to effect the weighted random selection of victim fragments. In particular, we create a set of tickets and divide them into two subsets with the ratio 80:20. Each fragment is assigned an equal share of the smaller subset. In addition, fragments with availability above 10% of the average are given a portion of the larger subset. The portion given to each fragment is proportional to the ratio between its number of nines and the sum of the number of nines of all such fragments. The notion of different currencies makes this division of tickets straightforward. For example: if a target node has three fragments with availabilities 0.99, 0.9, 0.5 or 2, 1, .3 "nines" respectively then the average availability in nines plus 10% is 0.76. Now if we have 100 lottery tickets the first fragment will get 67+6.6 tickets from the first and second pool respectively, the second fragment will get 13+6.6 tickets and the third fragment will get 0+6.6 tickets. Overall the chances of each fragment to be evicted will be 0.73, 0.19 and 0.06 respectively.

The intuitions behind our replacement policy are as follows. First, we reject the incoming fragment if it will simply become a target for eviction the next time a replication request is received by the target peer. Without this condition, we will simply be shifting fragments around without much effect. Our threshold for this outright rejection may seem rather low; at some cost of bandwidth, if we were less aggressive at rejecting fragments, perhaps over time, the system can reach a better configuration. However, our experimentation shows that this threshold does not have a large effect on the outcome, although when we become significantly less aggressive, bandwidth usage increases significantly[2]. Since we are targeting environments where bandwidth may be a precious commodity (see Section 4), we decided that an aggressive threshold was appropriate.

[2]We experimented with various ratios between the two pools as well as the threshold for rejecting a fragment outright. Space constraints prevent us from showing these results. However, as long as these parameters reflect the two motivating intuitions, changes had very limited impact.

Next, we penalize over-replicated files heavily for the number of nines in their availability, making it highly probable that a fragment of an over-replicated file will be evicted. We use the number of nines rather than the availability itself because it linearizes the differences between values, i.e. the difference between 0.9 and 0.99 is the same as that between 0.99 and 0.999.

3.4 Optimizations at Replicators

While the critical decisions rest at the targets in our algorithm, replicators can implement several optimizations to increase the convergence rate. First, a replicator might favor files that have low estimated availability over files with high estimated availability. Because of hoarding and potentially stale data, it is again necessary to use a weighted random selection process instead of a deterministic process. We use lottery scheduling in a manner similar to replacement, except in the reverse sense, of course, favoring files with low availability.

Second, a replicator can try to find peers with free space in their replication store rather than causing an eviction at a peer whose replication store is full. To implement this optimization, a replicator first selects a target randomly. Then, it inquires whether the target has sufficient free space in its replication store. If not, then the replicator selects another target, again randomly. The replicator repeats this process for five times (it could arbitrarily be any number of times), and then gives up and simply chooses randomly from these five previously chosen targets.

Finally, a member can choose to replicate only a subset of its hoard set to increase the availability of this subset at the cost of reduced availability for the remaining files. The power of our approach is that, if some member is interested in a file being highly available, it can act as a champion for that file by hoarding it. Ultimately, it is up to the application that places the hoarded files to decide when extra copies are needed.

3.5 Resiliency to Misbehaving Peers

Our current scheme, as it stands, is tolerant of a small number of buggy peers or members that are trying to take advantage of peers' replication stores for use as remote storage. First, consider buggy peers that might corrupt fragments in their replication stores. Our use of an erasure code includes embedded error detection so that a corrupted fragment cannot harm the integrity of the file; rather it would only lead to wasted space and lower than expected availability for the corresponding file[3].

Next, consider buggy or greedy members that attempt to push already highly available files or push files at a higher than agreed upon rate. Because replacement decisions are made at the targets, such actions will waste

[3]Cryptographic techniques could easily be employed to further strengthen the protection of data integrity. We view this as an application-level decision rather than one that should be built-in at the infrastructural level that we are exploring.

bandwidth and computing power but otherwise should not affect the efficiency of how the aggregated replication store is used.

Buggy or greedy peers can also provide misleading information along two dimensions: their average availability and whether they have a particular file. The combination that may be most damaging to our algorithm is when a peer advertises very high availability and that it is hoarding a significant portion of the shared data collection. In this case, the community is likely to over-estimate the availability of a significant number of files and so do not replicate them sufficiently. This frees up remote storage for use by the faulty/greedy peer. However, it is actually not in the self-interest of a member to do this because he would get numerous requests for the files he claims to be hoarding; this member is thus giving up bandwidth resources in trying to utilize remote peers' storage. Also, it is relatively easy to detect when a peer is faulty in this manner by verifying claims from peers that seem to have overly large amount of replicated data.

4 Evaluating the Achievable Availability

In this section we present a simulation-based study of our replication scheme's impact on file availability. We begin by describing our simulator and three distinct communities that we will study. Then, we present simulation results and discuss their implications.

4.1 Experimental Environment

We have built an event driven simulator for this study. To achieve reasonable simulation efficiency, we made several simplifying assumptions. First, we assume that all members of a community attempt to replicate files from their hoard sets at synchronous intervals. Second, we do not simulate the detail timing of message transfers. Finally, we do not simulate the full PlanetP gossiping protocol that is used to replicate the directory and the global index. In order to account for potential data staleness due to the latency of gossiping, we reestimate file availability and reassign tickets for the various lotteries only once every 10 minutes[4]. This batching of computations actually serves two purposes. First, it forces peers to work on outdated and inaccurate information (far more inaccurate than the latency provided by PlanetP for the sizes of communities studied here [6]). Second, it simulates a practical real implementation since estimating files' availability and giving tickets to every file and fragment is not a computationally trivial process.

Also for simplicity we will always fragment files using a Reed Solomon code where not only n is fixed (as described earlier), but also m. This means that all fragmented files can be reassembled with 10 fragments, regardless

[4]This simulated time unit is only relevant when compared with the gossiping interval, which is assumed to be 30 seconds. This assumed 30 seconds interval can be thought of as a time unit that is translatable to any real time interval. For example, if we map this unit to 60 seconds, then all time periods reported in our results would simply be multiplied by two.

	File Sharing (FS)	Commercial (CO)	Workgroup (WG)
No. Members	1,000	1,000	100
No. Files	25,000	50,000	10,000
Files per Member Dist.	Weibull(sh:1.93,mean:25)	Weibull(sh:6.33,mean:50)	Weibull(sh:0.69,mean:100) or Uniform
File Size Dist.	Sigmoid(mean:4.3MB)	Lognormal(μ:12.2 σ:3.43)	Constant 290Kb
Hoarded Replica Dist.	mod. Pareto(sh:0.55 sc:3.25)	None	None
Node Availability	24.9% (average)	80.7% (average)	33.3% (fixed)

Table 1: *Parameters used to simulate each environment*.

of its size. Although in practice the algorithm could vary m to achieve the best trade-off between fragment size, download latency, and space utilization, we fixed it to simplify the analysis.

4.2 Simulated Communities

We define three distinct P2P communities to assess the impact of replication under different operating environments which represent different styles of peer-to-peer communities. The first community is representative of a very loosely coupled community sharing mostly multimedia files; the second resembles a corporate department; and the third models a distributed development group. The environments vary in the following ways: (i) The per peer mean uptime and downtime; we model peer arrival to and exit from the online community as exponential arrival processes based on the given means; (ii) the number of files per node; (iii) the number of hoarded replicas per file; (iv) the amount of excess space on each node, and (v) the file sizes. Table 1 summarizes the parameters of each of the communities.

For all three communities, we vary the amount of excess space provided by each member across several different simulations. In all case we refer to excess space as a proportion of the number of bytes in members' hoard sets. In all the scenarios the number of files and nodes has been scaled to allow us to run experiments within reasonable times. As shall be seen, since nodes provide excess space based on what they share, the total number of files and nodes will not affect the algorithm. In all experiments, we assume a target file availability of three nines (99.9%), as provided nowadays by reputable web sites [14].

File-sharing community (FS). The first community that we study is modeled using two sources of data: (i) Saroiu et al.'s reported statistics collected from the Gnutella and Napster communities [20], and (ii) data collected from a local DirectConnect [8] community comprised of University students. In particular, for this community, we simulate 1,000 peers sharing 25,000 files. The number of files per peer is modeled using a Weibull distribution approximating Saroiu et al.'s reported actual distribution. File sizes are modeled using a Weibull distribution approximating our measurements of the local student community, which is shown in Figure 1(a). We cross these data sets because Saroiu et al. do not report the latter. Since the communities serve essentially the same purpose, we believe that data

from one is likely to be representative of the other (we verified this by checking characteristics that were measured in both studies). The number of hoarded replicas per file was also taken from the local community and is shown on Figure 1(b).

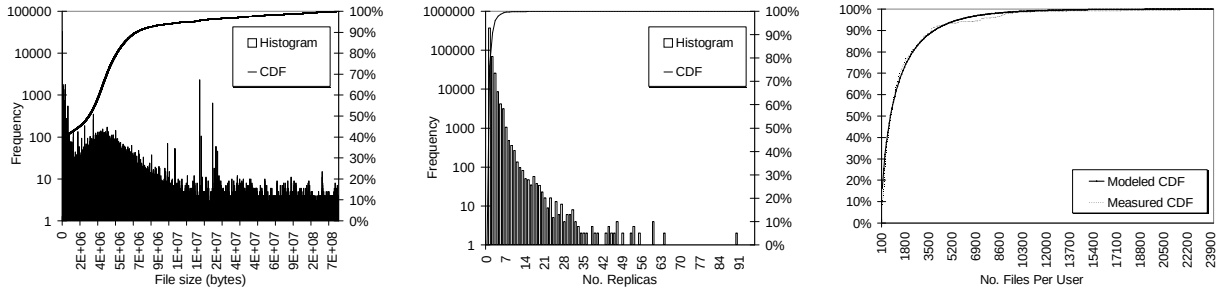
Finally, we use Saroiu et al.'s reported uptime and availability for the Gnutella community to drive members' arrival to and departure from the online community. We made two assumptions because of insufficient data: (i) we correlate availability with average length of online times; that is, the most highly available members also have the longest average online times. This correlation is intuitive because it avoids the pathological case of a highly available member constantly and rapidly flipping between being online and offline. (ii) we correlate the file distribution with availability, where the most highly available members have the most files. This correlation is motivated by measurements on our local student community, and suggests that "server-like" peers typically hold the largest portion of the shared data.

Corporate community (CO). The second community represents what we might expect to see in a corporate or university environment. In addition to being a significantly different environment than FS, studying this environment also allow us to compare our results with Farsite [4]. Farsite has somewhat similar goals to ours but takes a significant, almost inverse approach (see section 5 for more details). Thus, this community is modeled using data provided by Bolosky et al. [4]. All the parameters shown in Table 1 were taken directly from Bolosky et al.'s work with the exception of the absolute number of nodes and total number of files, which were scaled down to limit simulation time.

Workgroup (WG). Finally, the third configuration tries to capture a geographically distributed work group environment, such as a group of developers collaborating on an open-source project. The idea is to evaluate the impact of not having any member with server-like behaviors, i.e., members with large hoard sets that are (relatively) highly available. In particular, we simulate 100 peers sharing 10,000 files of 290KB each—this is the average file size of non-media files found in [10]. The number of files per user is distributed either uniformly or according to a distribution approximating our measurements of the local P2P network, shown in Figure 1(c). Peers will follow a work schedule with a mean of 1 hour online followed by 2 hours offline. Again, arrival is exponentially distributed.

4.3 Other Replication Algorithms

To evaluate the impact of the amount of information assumed to be available to drive the replication algorithm, we will compare our algorithm against two alternatives: the first one, called BASE, simulates nodes pushing and accepting replicas in the complete absence of information on file availability; the second, called OMNI, assumes centralized knowledge, where replication is driven by a central agent that tries to maximize the minimum file availability. Thus,



(a) **(b)** **(c)**

Figure 1: *(a) CDF and histogram of file sizes observed on DC. (b) CDF and histogram of replicas per file observed on DC. (c) CDF of the number of files per user observed on DC and a fitted 2 parameter Weibull distribution (shape:0.69, scale:1167, mean:1598 files per user).*

a comparison with BASE quantifies the effect of having approximate knowledge of current file availability. Comparing against OMNI quantifies the effect of autonomous actions using loosely synchronized data as opposed to having perfect, centralized information.

In BASE, nodes will use FIFO to manage the excess storage and replicators will only select files when its estimated availability is below the target availability. Note that this is really just an optimization to limit simulation time as the results would be essentially the same if BASE didn't implement this optimization.

OMNI is implemented as a hill-climbing algorithm that tries to maximize the availability of the least available file on every step. The idea is that a central agent will compute the number of fragments and their distribution such that it maximizes the minimum file availability. OMNI assumes centralized/perfect knowledge of peer behaviors, file and fragment distributions, and the ability to replicate a fragment on any member at any time. This algorithm was motivated by Bolosky's et.al. work [4], who shows that in an environment like CO it produces allocations close to the optimal.

4.4 Availability Improvements

Availability of Peers vs. Excess Storage Needed. We begin our evaluation considering two factors affecting file availability: the size of the aggregate replication store, and members' average availability. In particular, we use equation 1 as a simple analytical model.

Figure 2(a) shows the file availability plotted as a function of available excess space for mean peer availability equal to that of the three communities. As expected from [4, 9], only a small amount of excess space is required to achieve very high file availability in the CO community: 3 nines availability can be achieved with 1.75X excess capacity while almost 5 nines can be achieved with 2X excess capacity. Achieving high availability is much more difficult for FS-type communities because many peers only go online briefly to locate and download information:

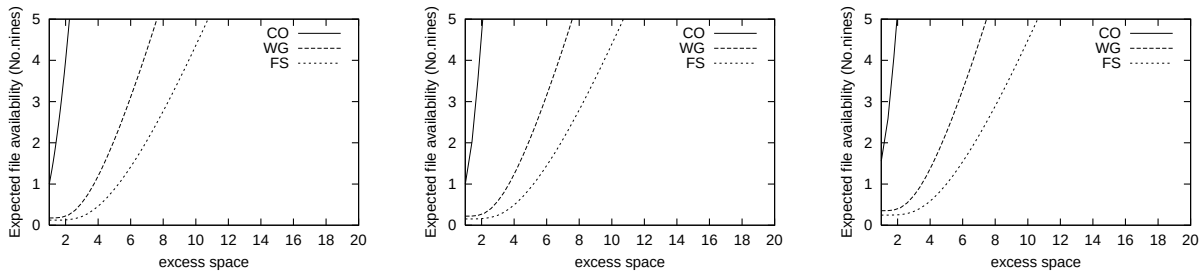


Figure 2: *Achieved file availability in number of nines plotted against excess storage in terms of x times the size of the entire collection of hoarded files for (a) 0 replication from hoarding, (b) 25% of the files have two hoarded replicas, and (c) 25% of the files have five hoarded replicas.*

around 8X of excess capacity is required for 3 nines availability. As shall be seen later, however, this model is pessimistic because using average peer availability looses the importance of server-like peers that are online most of the time. Our simulation results will show that only around 1.5X and 6X of excess capacity is needed to achieve 3 nines availability on CO and FS respectively.

Finally, we observe from figures 2(b,c) that file hoarding has little effect on file availability except in the case where peer availability on average is very high. Although this is again pessimistic for communities with server-like peers, it validates our decision to use erasure coding for replication (and is consistent with previous results obtained by Weatherspoon et.al.[24], showing that erasure coded replication can reduce space usage by 2 orders of magnitude).

Overall Availability. We now consider simulation results for the three communities discussed earlier. Figure 3 shows the CDFs of file availability for all three communities; Table 2 pulls out some of the more interesting points in these CDFs. In this figure, and for the remainder of the paper, all presented file availability are computed using equation 1, using the final placement of files and fragments, with P_{avg} computed using the *expected* availability of each peer. An alternative approach would have been to present the actual measured availability; that is, divide the observed amount of time that a file was accessible by the total simulated time. However, this estimation is often optimistic because it is difficult to simulate a sufficiently long period of time to observe significant downtime for highly available members. We also studied several other estimators which we do not discuss here because of space constraints; we refer the interested reader to our website for a discussion and comparison of these estimators (http://www.panic-lab.rutgers.edu/Research/planetp/).

As expected from our analytical study, the CO community has little trouble in achieving high file availability because the average availability of members is high. At 2X excess storage capacity, over 99% of the files have higher than 3 nines availability, with the minimum file availability of 2.73 nines (or 99.8%).

For the FS community, we see that using around 6X excess storage capacity, we achieve 3 nines availability for

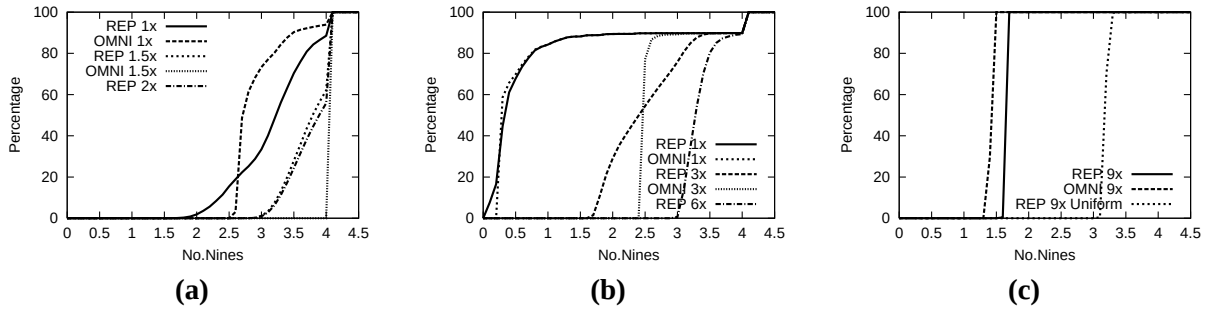


Figure 3: *CDFs for (a) the CO community with 1X, 1.5X and 2X, (b) the FS community with 1X, 3X, and 6X, and (c) the WG community with 9X excess storage capacities. REP refers to our replication algorithm as described in Section 3. OMNI is as described in Section 4.3.*

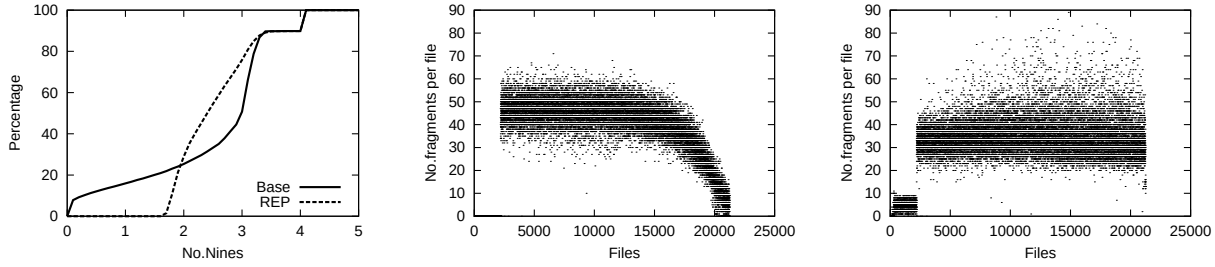
	Min	1%	5%	Avg		Min	1%	5%	Avg
CO 1X	1.2083	1.92	2.17	3.20	FS 3X	1.5944	1.69	1.75	2.55
CO 1X OMNI	2.5976	2.60	2.60	2.90	FS 3X OMNI	2.4337	2.43	2.43	2.63
CO 1.5X	1.5829	2.99	3.14	3.91	FS 6X	2.9199	3.01	3.05	3.36
CO 1.5X OMNI	4.0000	4.00	4.00	4.19	WG 9X	1.5208	1.61	1.61	1.61
CO 2X	2.7357	3.01	3.17	4.02	WG 9X OMNI	1.3518	1.35	1.35	1.41
FS 1X	0.0005	0.01	0.06	0.78	WG 9X Uniform	3.0001	3.11	3.11	3.14
FS 1X OMNI	0.2587	0.26	0.26	0.79					

Table 2: *Descriptive statistics for the experiments in Figure 3 (in nines).*

99% of the files, with the minimum availability being 2.9 nines. This is significantly less than the 8X predicted by our model. Because of the presence of server-like peers that are highly available, files that are hoarded by these peers achieve high availability from the beginning without using extra storage. Further, any fragment stored at these peers increases the availability of the corresponding file significantly and reduces the total number of fragments needed.

Observe that for both CO and FS, there is subset of files that achieve much greater availability than the average. Interestingly, this is not related to the fairness properties of our algorithm. Rather, it is because these files are naturally replicated on very highly available peers through hoarding. If we look at the number of fragments generated per file, we would see that most of these files were never fragmented at all, or had only a few fragments, which are useless since files need at least 10 fragments to increase availability. This constitutes a key difference between our assumed environment, and thus the approach to replication, and a file system such as Farsite. In the later environment, the "excess" hoarded copies would be collapsed into a single copy and thus increasing the fragment space; however, our view of hoarded copies as user-controlled copies prevents us from taking that approach.

For the WG environment, we observe an interesting effect. When files and excess storage are uniformly distributed among peers 9X excess capacity is enough to achieve 3 nines availability, as predicted. Nevertheless, if files and excess capacity are non-uniformly distributed, then average availability drops sharply. This degradation, also



(a) **(b)** **(c)**

Figure 4: *(a) CDFs of file availability for the FS community 3X. The number of fragments per file with the files ordered according to the average availability of the nodes that are hoarding them for (b) BASE, and (c) REP.*

observable on OMNI, arises from the small number of peers in the community and our assumed per-peer coupling between the size of the hoard-set and the replication store. In this case, peers with the most files to replicate also have the most excess storage, which are not useful to them. Further, since there are no high-availability peers that are up most of the time, it is not easy for replicators to find free space on the subset of peers with large replication stores. This points to the importance of even a few server-like peers that provide both excess capacity as well as high availability in communities similar to FS. Note that because OMNI tries to improve minimum availability, in this case while the minimum availability is higher than REP, its average availability is lower.

Comparing Against Having Centralized Knowledge. Having centralized information so that a static replication layout can be precomputed has two main benefits: (i) requires excess less space to achieve target availability, and (ii) increases the minimum availability. While the former is not insignificant—for example, for CO, OMNI was able to achieve over 4 nines availability with only 1.5X excess storage and close to three nines availability with only 3X excess storage for FS—we believe the latter is a more important difference. (Note that a comparison of files achieving more than 3 nines availability between REP and OMNI is also not fair since REP stops attempting to increase a file's availability once it reaches 3 nines.) With the continual rapid growth in capacity and drop in price of storage, having sufficient excess space on order of 2-9X does not seem outrageous. However, increasing the minimum file availability may be critical for applications like file systems.

One main reason why we cannot approach OMNI more closely for minimum availability, even with the above replicator-side optimization, is that it is difficult for members with low availability to maintain high availability for files in their hoard sets where there is insufficient excess storage. Over time, more available peers win in the competition for excess storage because they have the advantage of being able to push their hoarded files more often.

Effects of Availability-Based Replacement. To assess the importance of being able to estimate file availability during replacement, Figure 4(a) shows the CDFs of the expected file availability for BASE and REP on the FS community with 3X excess storage. Observe that the main effect of the receiver implementing a smart replacement

algorithm is to increase fairness. Under BASE, nearly 16% of the files have availability less than a single nine, compared to only 1% under REP. REP also achieves a better average availability of 1.7 nines than BASE's 0.9 nines.

Figure 4(b) and (c) show the reason behind these differences. BASE's FIFO replacement strategy favors peers who are frequently online. These peers achieve faster push rates and so have an unfair fraction of the replication store. Their files end up having many more fragments than those on less available peers, even though the community needs the latter's content to be more highly replicated. This effect is magnified when a file is hoarded by several peers with high availability, because it gains an even greater aggregated push rate. Under REP's replacement policy, the replication store is divided more evenly, with some favoring of files from less available peers.

In brief the effect of BASE's uncontrolled replacement policy leaves the control of the excess space to the most available nodes, which are the ones that need it the least.

Bandwidth Usage. Finally, we study the amount of bandwidth used after a community has reached stability. To put bandwidth in perspective, we will use the average number of bytes transferred per hour per node over the average file size. In essence, this metric captures the rate of replication that continues because, lacking centralized knowledge, the system is not able to discern that it should cease trying to increase availability—there is just not sufficient excess storage.

For CO, when the excess space is only 1X, REP continues to replicate on average 10 files per hour. If we increase the amount of excess storage to 2X the average amount of files replicated drops to zero, recall from figure 3(a) that 2X is enough to reach the target availability of 3 nines. Similarly, in FS the number of files replicated per hour goes from 241 to 0 as we increase the excess space from 1X to 3X (figure 3(b)). When comparing these results to the ones obtained by the BASE algorithm we find that for FS with 3X excess storage (figure 4), BASE replicates 816 files per hour against zero in REP.

This points out an important advantage of being able to dynamically estimate file availability: when the available excess storage is not too much less than what is required to achieve the target availability, our approach is relatively stable. This implies that while it is important to set the target availability to what the community can support—for example, it is not reasonable to target three nines availability when there is only 1X excess storage in FS as shown by the large number of files being pushed—it doesn't have to be highly accurate.

5 Related Work

While many recent efforts in peer-to-peer research has focused on how to build extremely scalable systems, relatively little attention has been given to the issue of availability in these systems. A primary distinction of our work is that it considers availability for a wide range of communities that are distinct from those assumed by previous P2P systems

like CFS [7], Ivy [15], Farsite [1], and OceanStore [12]. In particular, we consider communities where the level of online dynamism is high and there may be considerable offline times for members operating in disconnected mode.

In contrast to our approach of randomly generating erasure coded fragments from a large space of such fragments, OceanStore [12] proposes to replicate files using a fixed number of erasure coded fragments, which are repaired periodically but at a very low rate (on the order of once every few months) [12]). In their scenario, nodes have relatively high availability and so replication, and repair of fragments, are motivated mostly because of disk failures. Conversely, in the environments we study, the wide variance of node availability makes it difficult to adopt a similar approach and also requires a more pro-active replication algorithm.

Similarly, Ivy [15] uses a distributed hash table called DHash [7] to store file blocks as well as a fixed number of replicas across a P2P community. To ensure availability, they assume that replicas are refreshed over time, either through a peer voluntarily migrating its content before going offline or through some agent responsible for periodically detecting and regenerating lost replicas. Further, because the location of data in this system depends on the current set of online members, it is difficult for them to adopt a less dynamic placement strategy such as ours. As already argued in Section 1, we believe that the constant refresh of data would be too expensive in highly dynamic communities.

We have found that the use of adaptive erasure codes and estimated node availability plays a key role in equalizing the overall file availability. Along these lines, FarSite [4] uses information about node availability to increase minimum file availability in a corporate LAN environment. While the placement approach taken in FarSite is somewhat similar to ours, the details are quite different because of the different underlying assumptions. Since FarSite is aimed at corporate LAN environments, where average node availability can be quite high, they have chosen not to use erasure coding in their replication. Rather, each file has a fixed number of copies, generally 3, which are carefully placed on nodes to maximize the minimum file availability. Further, FarSite takes a more aggressive stance toward using network bandwidth to adjust to changing node availability while better utilizing all the available disk resources. While bandwidth is plentiful in a corporate LAN environment, it may be much more constrained in other P2P environments. Finally, our replication scheme is appropriate for disconnected operation, where members may wish to access part of the shared data while offline. FarSite does not consider such communities.

6 Conclusions and Future Work

In this work, we have addressed the question of increasing the availability of shared files for a range of P2P communities. We have attempted to quantify a conservative estimate of the amount of excess storage required to achieve practical availability of 99.9% by studying a decentralized algorithm that only depends on a modest amount of

loosely synchronized global state. Indeed, our results show that it is exceedingly difficult to achieve this level of availability if individuals do not at least have approximate knowledge of peers' availability and files' current availability, which exactly comprise the global state that we assume are maintained by our infrastructure.

Our results are extremely encouraging because they demonstrate that practical availability levels are achievable in a completely decentralized P2P system with low individual availability. For example, even in a community where the average availability is only 24%, we can achieve a minimum and average availability of 99.8% and 99.95%, respectively, at only 6X excess storage. With today's low cost storage, this degree of replication seems quite feasible. This result demonstrates that such availability levels do not require sophisticated data structures, complex replication schemes, or excessive bandwidth. Indeed, our achieved availability levels are on par with today's managed services [11, 14].

On the other hand, our results demonstrate that the presence of a small fraction of server-like members which are online most of the time is critical. A community where each member is available 33% of the time, giving an average availability of 33% (which is significantly more than 24%), requires 9X excess storage to achieve 99.9% availability even when the content and excess storage is uniformly distributed.

Finally, note that while we have focused on studying how to increase the availability of static content to limit the scope of this work, our approach is not limited to the sharing of read-only data. At worst, updating a file is the same as injecting a new version and letting the replacement mechanism evict the old copies. Applications can greatly optimize this process, for example by propagating updates as diffs. Therefore, as update propagation and replication is highly dependent on the application, we leave this as an issue for future work.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[2] R. Bhagwan, S. Savage, and G. Voelker. Understanding Availability. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2003.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] 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 *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2000.

[5] F. M. Cuenca-Acuna and T. D. Nguyen. Text-Based Content Search and Retrieval in ad hoc P2P Communities. In *Proceedings of the International Workshop on Peer-to-Peer Computing (co-located with Networking 2002)*, May 2002.

[6] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, June 2003.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[8] Direct Connect. http://www.neo-modus.com.

[9] J. R. Douceur and R. P. Wattenhofer. Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, Oct. 2001.

[10] K. M. Evans and G. H. Kuenning. Irregularities in file-size distributions. In *Proceedings of the 2nd International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, July 2002.

[11] J. Gray. Dependability in the Internet Era. Keynote presentation at the Second HDCC Workshop, May 2001.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, 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 *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov. 2000.

[13] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, July 2002.

[14] M. Merzbacher and D. Patterson. Measuring end-user availability on the web: Practical experience. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, June 2002.

[15] A. Muthitacharoen, R. Morris, T. Gil, and I. B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Aug. 2001.

[17] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2):24–36, Apr. 1997.

[18] T. Roscoe and S. Hand. Transaction-based charging in mnemosyne: a peer-to-peer steganographic storage system. In *Proceedings of the International Workshop on Peer-to-Peer Computing (co-located with Networking 2002)*, May 2002.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001.

[20] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, Jan. 2002.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Aug. 2001.

[22] W. Vogels, R. van Renesse, and K. Birman. The Power of Epidemics: Robust Communication for Large-Scale Distributed Systems. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, Oct. 2002.

[23] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 1994.

[24] H. Weatherspoon and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002.

[25] Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, April 2000.