Efficient Cooperative Backup with Decentralized Trust Management

NGUYEN TRAN, FRANK CHIANG, and JINYANG LI, New York University

Existing backup systems are unsatisfactory: commercial backup services are reliable but expensive while peer-to-peer systems are cheap but offer limited assurance of data reliability. This article introduces Friendstore, a system that provides inexpensive and reliable backup by giving users the choice to store backup data only on nodes they trust (typically those owned by friends and colleagues). Because it is built on trusted nodes, Friendstore is not burdened by the complexity required to cope with potentially malicious participants. Friendstore only needs to detect and repair accidental data loss and to ensure balanced storage exchange. The disadvantage of using only trusted nodes is that Friendstore cannot achieve perfect storage utilization.

Friendstore is designed for a heterogeneous environment where nodes have very different access link speeds and available disk spaces. To ensure long-term data reliability, a node with limited upload bandwidth refrains from storing more data than its calculated maintainable capacity. A high bandwidth node might be limited by its available disk space. We introduce a simple coding scheme, called XOR(1,2), which doubles a node's ability to store backup information in the same amount of disk space at the cost of doubling the amount of data transferred during restore. Analysis and simulations using long-term node activity traces show that a node can reliably back up tens of gigabytes of data even with low upload bandwidth.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Design, Reliability

Additional Key Words and Phrases: Cooperative backup, social network, erasure code

ACM Reference Format:
Tran, N., Chiang, F., and Li, J. 2012. Efficient cooperative backup with decentralized trust management. ACM Trans. Storage 8, 3, Article 8 (September 2012), 25 pages.
DOI = 10.1145/2339118.2339119 http://doi.acm.org/10.1145/2339118.2339119

1. INTRODUCTION

Users are keeping an increasing amount of valuable data in digital format. Today, it is common for users to own more than tens of gigabytes of digital pictures, videos, experimental traces, and so on. Backing up this data properly is a burden for many users. Even expert users often fail to back up data due to the inconvenience and expense of current solutions. In our research group, for example, many members keep important data on their desktops and not all of them currently back up such data.

An ideal backup solution would ensure that valuable data survive threats such as disk failures, operator mistakes, theft, and natural disaster. The backup data should be available online so that users can gain access from anywhere there is Internet. It should also be easy to operate; systems that present a continuing operational headache are likely to be neglected and fail. Finally, the system should be cheap (ideally, it would

This project was partially supported by the NSF CAREER award CNS-0747052. N. Tran is also supported by a Google Ph.D. fellowship in distributed systems.
Author's address: N. Tran, email: trandinh@cs.nyu.edu.

DOI 10.1145/2339118.2339119 http://doi.acm.org/10.1145/2339118.2339119

be free). Convincing users to pay for an eventuality (disk failure) that they view as distant and unlikely is a significant nontechnical challenge that can be avoided by providing backup services at little or no cost.

Network backup systems have the potential to meet these requirements. By storing data off-site, they provide a strong defense against a wide variety of threats. They can also be made to operate fully automatically (no need to rotate backup media, for instance). However, not all users find paid network backup services like dotMac or Amazon S3 affordable. For example, a graduate student is unlikely to pay Amazon $180 a year to back up her 100GB experimental trace files even though they are important. Peer-to-peer storage systems, in which users make use of each other's idle disk space to store backed up data, are a promising alternative to inexpensive online backups. However, existing peer-to-peer systems offer limited assurance for data reliability as a user's data is stored on potentially malicious or unavailable peer nodes [Cox and Noble 2003; Aiyer et al. 2005; Lillibridge et al. 2003; Rowstron and Druschel 2001b; Ngan et al. 2003].

This article presents Friendstore, a cooperative back up system that allows users to back up valuable data on other peer nodes that they trust. In Friendstore, each node acts as its own trust authority to admit a subset of nodes for storing its backup data. In practice, a user admits nodes belonging to her friends or colleagues and trusts them to be available and provide timely restore service when needed.

Friendstore's decentralized trust management allows it to solve two long-standing problems that have plagued other peer-to-peer backup systems [Cox et al. 2002; Cox and Noble 2003; Lillibridge et al. 2003; Batten et al. 2002]. First, peer nodes have little incentive to remain available. Maintaining data durability despite the resulting high membership churn requires more data to be transferred than low bandwidth links can support [Blake and Rodrigues 2003]. Given the large amounts of data to be stored by a backup system, this problem is particularly severe. Second, without trust, there is no assurance that nodes storing others' backup data will provide restore service in times of need. A malicious node can deny service, but even a selfish node has no real incentive to assist others to restore as the node requesting restore has nothing of immediate value to offer the node providing the restore service. Since Friendstore stores data on trusted nodes only, it offers a nontechnical solution to both the availability and denial-of-service problems; users enter storage contracts with their friends via real world negotiations. Such contracts are reliable because social relationships are at stake. These benefits come at a cost; by storing backup data only on a subset of nodes, Friendstore cannot utilize all storage resource as efficiently as a homogeneous peer-to-peer system [Dabek et al. 2004; Rowstron and Druschel 2001a; Rhea et al. 2005].

Although Friendstore's architecture is conceptually simple, a number of technical challenges remain in order to provide reliable storage with the highest possible capacity. The capacity of Friendstore is limited by two types of resources: wide area bandwidth and the available disk space contributed by participating nodes. Bandwidth is a limiting resource because nodes must recopy backup data lost due to failed disks. To prevent a node from storing more data than it can reliably maintain, we propose letting each node calculate its maintainable capacity based on its upload bandwidth and limit the amount of backup data it stores in the system accordingly. The system's capacity may also be limited by the available disk space. We propose trading off bandwidth for disk space by storing coded data in situations when disk space, instead of bandwidth, is the more limiting resource. Our scheme, XOR(1,2), doubles the amount of backup information stored at a node, at the cost of transferring twice the amount of data during restore in order to decode the original data.

The technical challenges addressed in this article, namely calculating maintainable capacity and trading off bandwidth for storage, are not unique to Friendstore but are

present in all replicated storage systems. However, the targeted deployment environment of Friendstore makes addressing these challenges a pressing need. Friendstore runs on nodes with a wide range of bandwidth and disk space. Some nodes are limited by the upload bandwidth, hence they must refrain from storing more data than the maintainable capacity. Other nodes are limited by the available disk space, so it is attractive to store more information in the limited disk space using coding.

We evaluate the long term behavior of Friendstore in simulations with a real world machine availability trace. Our results show that each Friendstore node can back up 48GB of data reliably even with a small upload bandwidth of 150kbps. Our preliminary deployment of Friendstore shows that nodes often have much higher upload bandwidth, suggesting the backup capacity of Friendstore will be high in practice.

The article is organized as follows. Section 2 discusses the underlying trust model that has inspired Friendstore's architecture. We next present Friendstore's overall design (Section 3), how a node calculates maintainable capacity (Section 4) and how it trades off bandwidth for storage (Section 5). In Section 7, we evaluate the long-term reliability of Friendstore using trace-driven simulations and share lessons from our early software deployment. Section 8 discusses related work and Section 10 concludes.

2. TRUST MODEL

The viability of all cooperative backup systems depends on the majority of participants cooperating, hence the name. Unfortunately, no technical solution can ensure that nodes always cooperate. For example, a node storing others' data can faithfully adhere to a system's protocol for a long time but decide to maliciously deny service when it is asked to help others restore. Therefore, the best a system can do is to ensure that our assumptions about how well behaved nodes act are highly likely to hold in practice. Systems do so by pruning the set of trustworthy nodes to eliminate misfits and creating disincentives to violating assumptions in the first place. For example, a node can frequently check that others are faithfully storing its data and remove any node that fails periodic checks [Cox and Noble 2003; Lillibridge et al. 2003; Aiyer et al. 2005] from the system. A disincentive to misbehavior could be punishments in the form of deletion of the offending node's backup data [Cox and Noble 2003] or expulsion from the system by a central authority [Aiyer et al. 2005]. Both of these approaches have drawbacks pruning mechanisms based on system-level health probes can be imprecise (e.g. it is difficult to distinguish a node who has just suffered from a hard disk crash from one that purposefully deleted others' data). Inflexible disincentives (e.g. deletion of an expelled node's data) could cause the system to be unnecessarily fragile. It is easy to imagine an incident such as unexpected software crashes or temporary network congestion leading to an escalating spiral of punishments. Some peer-to-peer file sharing applications rely on a reputation system to help a node choose cooperative peers [Kamvar et al. 2003]. Calculating a node's reputation requires observations of its past misbehavior. In a cooperative backup system, a node misbehaves by denying others' restore requests in times of need. Therefore, such misbehavior not only takes a long time to observe but also is likely to involve data loss for users. As a result, reputation systems are not applicable for cooperative backup.

Friendstore leverages information in the social relationships among users to select trustworthy nodes and provide strong disincentives against noncooperative behavior. Each user chooses a set of trusted nodes that she believes will participate in the system over the long term and have some minimal availability according to her social relationships. A Friendstore node only stores backup data on those trusted nodes. By exploiting real-world decentralized trust relationships in this way, Friendstore is able to use simple and lightweight technical mechanisms to provide a reasonable expectation that nodes will behave as expected. Friendstore checks for the presence of remote

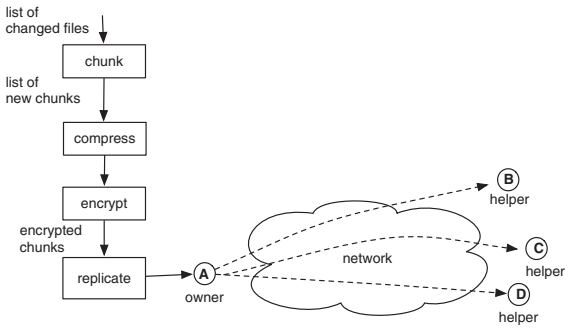

Fig. 1. The sequence of steps an owner takes for preparing a collection of files for backup on remote helpers.

backup data infrequently and uses a long timeout to mask transient node failures knowing that the unresponsiveness of a trusted neighbor is more likely due to its user's vacation than an act of malice. The use of social relationships may also help ensure some minimal level of node availability in times of need because a user can always contact her friends to turn on their computers to help with the restore. Also, disincentives in this system carry more weight since they stem from possible disruption of the social relationships; violation of trust results in the resentments of one's friend which we believe most users want to avoid. Friendstore defers punishments for a misbehaving node such as deleting of its backup data, to individual users who are free to use their own retribution policies based on more complete and accurate information. Although a Friendstore user trusts her friends for being available and not performing denial-of-service when she needs the data, she does not trust the friends on data's privacy and integrity. Her friends may want to look at her data. Their storage may be faulty.

3. BASIC DESIGN

3.1. Overview

Friendstore consists of a collection of nodes administered by different users. Each node runs an identical copy of the software and communicates with a subset of other nodes over the wide area network. The software running on a node has two roles: backing up a node's local data and helping others store their backups. We refer to a node as an *owner* when it is performing activities involving its own data and a *helper* when it is acting to help others. Each node is named and authenticated by its public key, and a user chooses a subset of helpers for storing data by configuring her node with the public keys of her friends' nodes.

An online backup system undertakes a number of activities: to store local data on remote helpers (backup), to periodically check that remote copies of its backup data are still intact and create new ones if not (verify and repair), and to retrieve remote backups following a disk crash (restore). We describe how Friendstore performs each task in turn.

3.2. Backup

An owner prepares a collection of files for backup in a sequence of steps as shown in Figure 1. The owner processes the files by splitting large files into smaller pieces, compressing, and encrypting individual pieces using symmetric encryption. Finally, it uploads r copies of its encrypted chunks on distinct helpers. We use the term *replica* to refer to these encrypted chunks stored at helpers. In our prototype implementation, r is set to two. We prefer replication over erasure coding for two reasons. First, a

space-efficient erasure code needs to store fragments on many distinct helpers, hindering incremental deployment. For example, a node needs at least five helpers in order to back up its data using Reed-Solomon(4,5). In contrast, a node can start backup with only one helper using replication. Second, nodes tend to create redundant replicas when they mistake offline neighbors for crashed nodes. It is easier to incorporate a redundant copy to achieve a bigger effective replication factor using replication, than a fixed rate erasure code [Chun et al. 2006].

Owners do not modify replicas at helpers, once created, but can explicitly delete them. To garbage-collect data, helpers expire replicas after a default period of three months. As a heuristic, an owner chooses a helper with the most available donated space for storing its backup data. Intuitively, this heuristic helps balance the storage load of the overall system by preferentially putting data on less loaded helpers.

3.3. Verify and Repair

Although an owner trusts its helpers to preserve its replicas with best effort, it must still be diligent in detecting accidental replica losses in order to repair damage over time.

A simple verification strategy is for an owner to periodically ask the helper to compute and return the hash of a randomly chosen replica. By comparing a helper's hash value with that computed from its local file system data, an owner can detect replica corruption and repair quickly. To prevent a helper from reusing precomputed hash values, which defeats the purpose of verification, an owner requests the hash of a replica starting from a random offset with wraparound so it is equally costly for a helper to precompute all hash values for a replica than to just compute the requested hash upon request.

An owner sends a batch of verification requests to a helper infrequently, for example, every 200 hours. Each verification request has a lenient deadline of 200 hours, so helpers can also batch requests from different owners and delay computation until a nonbusy time. A verification request also serves as a renewal notice to extend the expiration time of the verified replica at the helper. Both verification requests and replies are piggybacked on the ping messages exchanged between a pair of neighbors every hour.

Whenever an owner receives incorrect hash values from a helper, it immediately resends lost replicas and initiates an additional synchronous round of verification in the hope of quickly detecting other losses on that helper. When an owner repeatedly fails to contact a helper for verification, it must decide how long to keep retrying before assuming the helper has permanently failed and regenerating the replicas at some other helper. We use a large timeout threshold of 200 hours to mask most transient failures without significantly compromising long-term data durability. Although we expect very few offline instances to exceed this threshold, a node will inevitably create unnecessary replicas due to long offline periods during the course of its operation over many years. An owner always reintegrates a helper's replicas when it comes back online so it is less likely to create additional ones again in the future [Chun et al. 2006].

3.4. Restore

Restoring data after an owner's disk crash is quite straightforward. However, since the owner might lose all its persistent data after a disk crash, it must store its private key used to encrypt the replicas, offline. The owner also needs to remember the identities of its helpers. For convenience, a node can keep its helpers' identities on a well-known centralized server. During the restore, the owner first downloads metadata for the backup, which was replicated at all helpers. The owner chooses the latest version among all the metadata sent back by the helpers, and starts to request the helpers for

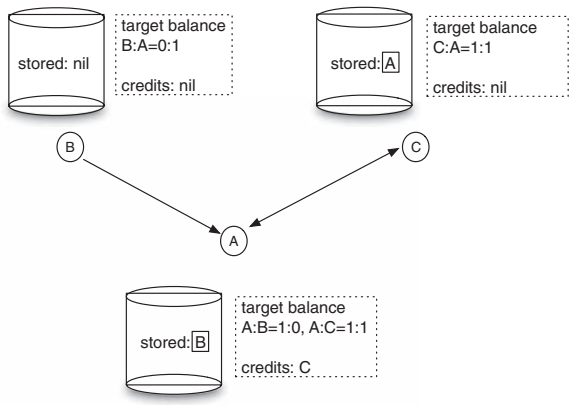

Fig. 2. An example illustrating the bilateral credit system used in Friendstore. Node A credits C for storing one unit of its data. This credit represents A's promise to store one unit of C's data upon request in the future. As node A has a 1:0 target balance with B, node B does not need to credit A for storage.

its lost replicas. Data is restored in the order of the priorities specified by users. As we discussed earlier, a helper has no real incentives to help an owner restore. The reason we believe it is likely to do so in practice comes from the real world social relationship between the users.

3.5. Fairness

Friendstore discourages one particular form of free-riding, namely, a neighboring node does not contribute enough disk space. Friendstore allows a user to specify a target balance with each neighbor. We support two types of target balances: equal exchange (1:1) and altruistic storage (1:0). In an equal exchange, a node agrees to store 1 unit of data in return for putting 1 unit of data on its neighbor. While equal exchange is a good rule when dealing with strangers [Cox and Noble 2003; Ngan et al. 2003], a user may have many reasons to behave altruistically towards neighbors belonging to herself or friends with strong social ties. For example, a user can specify 1:0 target balance between her desktop and laptop machines so Friendstore would back up the laptop's data on the desktop node but not the other way around.

Each pair of neighbors in an equal exchange relationship keeps a bilateral credit summary to ensure each keeps its promise to reciprocate the other's storage contribution. Figure 2 shows an example. Node A is in an equal exchange relationship with C. After A stores a unit of data on C, it remembers to credit C, that is, it will store at least one unit of data should C ask it to do so in the future. On the other hand, because A behaves altruistically towards B, node B does not need to give any credits to A after storing data on A. An owner prefers storing data at a helper with whom it has the most credits when deciding among multiple eligible choices. When a node is denied storage by a neighbor with whom it has positive credits, it reports such incidence to its user for punishment or further investigation.

Friendstore's bilateral credits are strictly between a pair of neighbors and not verifiable by others. Credits are also not transferable. More sophisticated schemes that use cryptographically verifiable, transferable claims [Cox and Noble 2003; Aiyer et al. 2005; Fu et al. 2003] result in more efficient global storage utilization. Unlike Friendstore, previous claims-based systems have homogeneous setups, where all nodes can use the donated storage on all other nodes. We choose to use the simpler bilateral credit scheme because Friendstore's global storage utilization is more limited by the underlying trust-graph structure than the inefficiencies of bilateral credits.

4. CALCULATING THE MAINTAINABLE CAPACITY

In this section, we analyze the maintainable storage capacity as limited by the network bandwidth. Each owner (or helper) uses the calculated maintainable capacity to restrict its storage consumption (or contribution).

If backup data is to be stored reliably, it must be recopied by owners as disks fail. The rate at which an owner can upload data determines the amount of data it can reliably store on remote helpers. This amount could be much less than the disk space available at helpers. To ensure the reliability of backup, we do not not want to store more data than can be maintained over the long term. Therefore, we propose letting each owner calculate its maintainable storage capacity (s_{max}) based on its upload bandwidth, and use this estimate to limit how much data it attempts to store on helpers. Similarly, we calculate the maintainable capacity for each helper (d_{max}) and use the estimate to limit the amount of data it contributes to other owners. For simplicity, we assume that a node's download bandwidth is larger than its upload bandwidth. Similar arguments apply when a node's download bandwidth is the more limiting resource.

Intuitively, the reliability of replicated data is affected by the amount of bandwidth required to recover from permanent disk failures relative to the amount of available bandwidth at each node [Chun et al. 2006]. When an owner stores s units of backup data with replication level r on remote helpers, it must consume $\lambda_f \cdot r \cdot s$ units of bandwidth to recopy r replicas per unit of data when disks fail at rate λ_f. Likewise, when a helper stores d units of replicas for others, it needs to upload one out of every r copies to help owners restore, consuming $\lambda_f \cdot \frac{d}{r}$ units of bandwidth. Since a node acts both as an owner and a helper, the total bandwidth required to recover from permanent failures is: $\lambda_f \cdot r \cdot s + \lambda_f \cdot \frac{d}{r}$. The required recovery bandwidth cannot exceed a node's available upload bandwidth (B), that is, $\theta = \frac{B}{\lambda_f \cdot (r \cdot s + \frac{d}{r})} \geq 1$. Intuitively, θ represents the ratio of the data creation rate over the permanent failure rate. θ must be greater than one just for the system to remain feasible.

In practice, permanent failures do not come at deterministic intervals. Therefore, in order to ensure low data loss, a node must limit the amount of data it stores remotely (s_{max}) and the amount of data it keeps for others (d_{max}) so that $\theta \gg 1$. Our simulation results show that when the required recovery bandwidth is less than one tenth of the actual available bandwidth, there is little data loss ($< 0.15\%$) over a five-year period (Section 7). Therefore, we use $\theta_{thres} = 10$. Additionally, we need to account for nodes with less than perfect availability (A), which lowers their effective upload bandwidth. Putting everything together, s_{max} and d_{max} must satisfy the following equation.

$$\lambda_f \cdot \left(r \cdot s_{max} + \frac{d_{max}}{r} \right) = \frac{1}{\theta_{thres}} \cdot B \cdot A. \tag{1}$$

In order to calculate s_{max} and d_{max} from Equation (1), we approximate the average disk lifetime as 3 years, that is, $\lambda_f = \frac{1}{9.5 * 10^7}$ failures/sec. Empirical studies suggest that hard disks have mean time to replacement of at least 5 years [Schroeder and Gibson 2007; Pinheiro et al. 2007]. We use three years as an approximation to take into account other factors contributing to permanent data loss such as operator error, accidental data loss during node reinstalls, and so on. More pessimistic users can use a lower estimate for λ_f. To simplify our analysis, we assume a node is in equal exchange relationships with all its neighbors so that a helper stores twice the amount of backup data ($d_{max} = 2s_{max}$) with a default replication level of 2. Substituting $\lambda_f, r = 2, \theta_{thres} = 10, d_{max} = 2s_{max}$ into Equation (1), we obtain $d_{max} = 2s_{max}$ and $s_{max} = 3.1 \times 10^6 \cdot B \cdot A$.

To calculate s_{max}, each Friendstore node estimates its availability and upload capacity via either measurements or explicit user inputs. For example, if a node is configured

with a capped 150kbps upload bandwidth for Friendstore and has 50% availability, it should limit the amount of backup to $s_{max} = 3.1 * 10^6 \cdot \frac{150*10^3}{8} \cdot 0.5 = 29$GB. Additionally, this node should contribute no more than $d_{max} = 2s_{max} = 58$GB disk space to others. On the other hand, if a node is on a high-speed campus network with 1Mbps upload bandwidth and 99% availability, it can back up and contribute 13 times more data and disk space ($s_{max} = 377$GB, $d_{max} = 754$GB).

Once a node has calculated its s_{max} and d_{max}, its owner refrains from storing more than s_{max} units of backup data on remote helpers and its helper does not store more than d_{max} units of data for other owners. Such restriction is important: if a node stores more backup data than s_{max}, it gives its user a false sense of security that her data are reliably backed up even though the node does not have enough capacity to ensure the longevity of those data. Similarly, if a node contributes more than d_{max}, it only hurts its neighbors' data reliability with no benefits to itself.

The calculation of s_{max} and d_{max} ensures low data loss rate in Friendstore if users operate in this safety zone. In addition to that, some users may have an extra requirement on service level agreement, for example, short restore time. In this case, each user can further constrain s_{max} and d_{max} based on her download capacity, number of friends, and their upload capacity. We have not implemented this feature in Friendstore, but it is not difficult to compute s_{max} and d_{max} given the owner's designed restore time, download capacity, and helpers' upload capacity. In practice, because neighboring users have real-world relationships, some form of manual intervention could dramatically increase the upload capacity of the neighbors. For example, following a disk crash, a user could request her friends to operate their nodes with a higher than usual availability and capacity to speed up the restore process. Her friends can also send data by mailing her portable storage devices such as DVD discs. Shipping portable storage using First-Class Mail service takes 2-3 days within the US. Therefore, using postal service is a more desirable option for users who replicate large amount of data. For example, it takes more than 3 days for a user with a typical DSL download capacity of 1MBps to restore 260GB of data. This calculation assumes the user has enough friends to utilize its download capacity fully. In this case, shipping portable storage is faster.

In Friendstore, each user explicitly configures an upload limit (B) for her node. We expect the configured upload limit to be much smaller than the actual uplink capacity. The software will monitor the current upload capacity to ensure that it is larger than B. If not, Friendstore will readjust its calculation of maintainable backup volume and delete some data stored at the machine as a helper. The verify and repair process at its neighbors will detect the data being deleted and replicate it at other nodes. The user who reconfigures B can also notify her friends to trigger the replication early, to avoid moving data around unnecessarily due to temporary network degradation, Friendstore triggers the change of its maintainable backup volume only if its measured upload capacity is less than B for five consecutive days.

5. STORE MORE INFORMATION WITH CODING

Disk space, rather than upload bandwidth, can also be the limiting resource in many circumstances. Since each Friendstore owner only has a few trusted helpers to store data, it is important to utilize helpers' available disk space efficiently. In this section, we explore coding schemes to back up more information in the same amount of disk space. We start by understanding the fundamental tradeoffs of various schemes in terms of their space saving, bandwidth overhead, and failure tolerance. Next, we show how to incorporate coding schemes in the basic design of backup, restore and verification procedures.

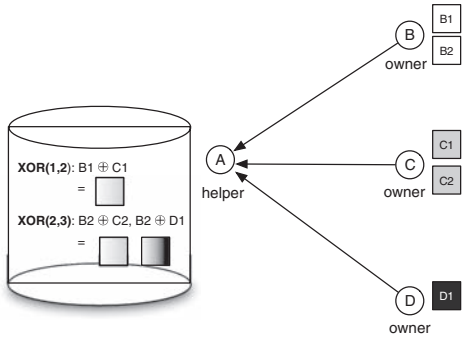

Fig. 3. An example of how a helper uses XOR(1,2) and XOR(2,3) to store check blocks from multiple owners.

5.1. Coding Tradeoffs

When upload bandwidth is plentiful, disk space could be the limiting resource. For example, when operating on a college campus network with 1Mbps links, a helper can reliably store up to $d_{max} = 754$GB data for other owners. But in reality, its idle disk space can be far less than d_{max}. In such scenarios, coding is attractive, as it trades off bandwidth for the ability to store more information in the same amount of disk space.

The intuition behind our coding schemes is to make use of the original data stored at an owner to help others recover their replicas. In particular, instead of storing multiple replicas belonging to different owners, a helper can code them into fewer blocks. We use the term *check block* to refer to coded replicas. For example, in Figure 3, instead of storing $B1$ and $C1$, helper A can store $B1 \oplus C1$, consuming one unit space as opposed to 2. To restore $B1$, helper A needs to fetch the original replica ($C1$) from owner C before combining it with its local check block to decode $B1$, that is, $B1 = (B1 \oplus C1) \oplus C1$. As the example illustrates, coding comes at the cost of both increased restore bandwidth and possible restore failures; replica $B1$ could be lost if node C fails during restore. What are the fundamental tradeoffs between space-saving, restore bandwidth overhead, and failure tolerance, and what are the corresponding coding schemes to achieve them?

We have found two general classes of coding schemes, XOR(1,n) and XOR(n-1,n). Both schemes achieve certain optimal tradeoff points with regard to space saving, restore bandwidth overhead and failure tolerance (see Appendix 10). In XOR(1,n), a helper codes n distinct owners' replicas together into a single check block, resulting in a space saving factor of $\frac{n-1}{n}$ and $n-1$ extra amounts of data transfer during restore. XOR(n-1,n) works as follows. Let $r_1, r_2, \ldots, r_n$ be n replicas from distinct owners. A helper stores $n-1$ check blocks in the form of $r_1 \oplus r_2, r_1 \oplus r_3, \ldots, r_1 \oplus r_n$. Figure 3 gives an example of XOR(2,3). XOR(n-1,n) can tolerate the failure of any $n-1$ of the n original data blocks. Suppose the only survived owner is i, with an original block r_i, the helper can retrieve r_i to XOR with its local check block $r_1 \oplus r_i$ to obtain r_1. After restoring r_1, the helper can proceed to restore the rest of the original blocks. Compared with XOR(1,n), XOR(n-1,n) achieves a smaller space saving of $\frac{1}{n}$ at the cost of only one extra unit of data transfer.

We note that XOR(n-1,n) differs from traditional coding schemes such as Reed-Solomon in that it is not designed to tolerate the failure of any combination of $n-1$ original and check blocks. This is because in Friendstore, all the check blocks on a set of original blocks reside on the same machine (see Figure 3). Therefore, all check blocks fail together and it is not meaningful to tolerate a subset of check block failures.

Coding increases the risk that a crashed owner will fail to restore its data because some other owner whose original data block is needed to decode the check block also

Table I. Tradeoffs in Terms of Increased Data Transfers During Restore, Space-Savings, and Probability of Restore Failure for XOR(1,n) and XOR(n-1,n)
The term p denotes the probability of a node failure during MTTR.

Coding scheme	Extra data transfer during restore	Space saving	Failure during restore
XOR(1,2)	$1\times$	$\frac{1}{2}$	$2 \cdot p$
XOR(1,n)	$(n-1)\times$	$\frac{n-1}{n}$	$n \cdot p$
XOR(2,3)	$1\times$	$\frac{1}{3}$	p
XOR(n-1,n)	$1\times$	$\frac{1}{n}$	p

fails. Let p be the probability that a node suffers a disk crash during the MTTR period. Without coding, the probability of restore failure is p (the helper fails). Suppose optimistically, that coding does not significantly increase MTTR, then the probability of restore failure using XOR(n-1,n) is $p + (1-p) \cdot p^{n-1}$. The first term represents the probability that the helper fails and loses all its stored checkpoints. The second term calculates the probability of decoding failure when the helper does not fail but the other owners involved in the check blocks fail. Since XOR(n-1,n) can tolerate the failure of $n-1$ original data blocks if all check blocks are safe, decoding failure occurs only when all n original blocks fail, that is, p^{n-1}. Ignoring second order terms when $n > 2$, we can approximate the overall restore failure probability as p. Since all XOR(n-1,n) schemes for $n > 2$ have the same failure probability, we will only consider XOR(2,3), which has the best space-saving. Similarly, using XOR(1,n), the failure of at least one of the $n-1$ other owners will cause decoding failure. Therefore, the overall probability of restore failure is $p + (1-p) \cdot (1-(1-p)^{n-1}) \approx n \cdot p$.

Table I summarizes the tradeoffs of various coding schemes. We point out that the probability of restore failure does not directly translate into permanent data loss rates because each Friendstore owner maintains $r = 2$ replicas. Therefore, we conjecture that a small increase in restore failure probability, for example, $2p$, is reasonable. We leave a more thorough evaluation of XOR(1,2) and XOR(2,3) to Section 7, using simulations. For the rest of the Section, we restrict our discussion to XOR(1,2), the default coding scheme used in Friendstore.

Since coding trades off bandwidth for storage, we must be careful not to apply XOR(1,2) in situations when bandwidth is the limiting resource. As Figure 3 shows, with coding, owner C must upload its replica again in response to owner B's failure as well as helper A's failure. Therefore, we update Equation (1) to reflect the additional data transfer by an owner when coding is used: $\lambda_f \cdot ((r+1) \cdot s'_{max} + \frac{d'_{max}}{r}) = \frac{1}{\theta_{thres}} \cdot B \cdot A$. Each owner uses the new estimate (s'_{max}) to constrain the amount of data it attempts to store on remote helpers while allowing XOR(1,2). Similarly, a helper uses d'_{max} to limit the amount of information it stores as coded replicas for other owners. For a node with 1Mbps upload bandwidth, $d'_{max} = 282$GB with coding and $d_{max} = 377$GB without coding. If the actual spare disk space is much smaller than 282GB, coding is attractive, as the node can store more information with the same amount of limited disk space.

One might wonder if the spacing-saving of XOR(1,2) can also be achieved by letting owners use erasure codes instead of replication ($r = 2$) in the first place. Unfortunately, efficient erasure codes require many fragments to be stored on distinct neighbors, a tough requirement for Friendstore, where many nodes have a small number of helpers. Even for a node with 5 helpers, using replication ($r = 2$) with XOR(1,2) results in less storage overhead than the most efficient erasure code (Reed-Solomon(4,5)) and achieves better reliability as well.

5.2. Backup with Coding

Since coding makes a node's restore procedure depend on another node that is not its immediate trusted neighbor, Friendstore allows each owner to explicitly specify whether or not it would allow a particular replica to be coded at a helper. The owner's incentive for coding is to back up more data in the limited spare disk space of its helpers. The helper's incentive for coding is to obtain more storage credits by storing more information for its neighbors.

The owners' decision to allow coding comes with additional responsibilities. In particular, it needs to upload its encrypted replicas to the helper again to help others restore. Therefore, an owner should allow coding only for replicas that correspond to immutable files that will not be deleted or modified, such as photos, mp3s, and videos. We rely on the normal verification process to detect replica loss due to unexpected changes in original files. It is in an owner's interest to help other owners restore because doing so also repairs its own replica that is lost due to others' failures.

When a helper is running short of donated space, it starts to compute check blocks using replicas for which coding is permitted by their owners. It uses a greedy algorithm that recursively chooses replicas from two owners with the most replicas left for coding. This way, we reduce the chances of not having space when the remaining replicas to be coded belong to fewer than two owners.

5.3. Verifying Check Blocks

Storing check blocks complicates the normal verification process. Since a helper does not have the original replica in its local storage, it cannot directly calculate the requested hash value of the replica. Nevertheless, we would like the helper to be able to compute the requested hash value with the help of other neighbors that store the original data for the check blocks. To achieve this, we make use of a homomorphic collision-resistant hash function with the property that $h_G(x + y) = h_G(x)h_G(y)$, where $G(p, q, g)$ specifies the hash function parameters [Krohn et al. 2004]. To apply a homomorphic hash function on check blocks, we change the XOR operator in XOR(1,2) to be the addition operation over Z_q, where q is a large prime. We illustrate the new verification protocol with the example in Figure 3. Suppose owner B asks helper A to compute the hash for replica $B1$. Helper A first requests the hash $h_G(\bar{C}1)$ from owner C, where $\bar{C}1$ is the complement of $C1$ in Z_q. Helper A then computes the requested hash as $h_G(B1) = h_G(B1 + C1)h_G(\bar{C}1)$.

To ensure that helpers only code data blocks when allowed, an owner requests nonhomomorphic hashes for data blocks that are not supposed to be coded and homomorphic hashes for others. Unfortunately, an owner cannot tell if a helper codes using XOR(1,2) as opposed to the less reliable XOR(1,n) with a large n. It is possible to detect such misbehavior, for example, the owner could monitor the rate of verification traffic from a helper asking for hashes of its original data blocks. If the rate far exceeds what is needed to verify check blocks for XOR(1,2), the owner has grounds for suspicion. We did not implement this check, as we believe such misbehavior is unlikely to arise in practice with trusted neighbors.

6. IMPLEMENTATION

This section describes the implementation of Friendstore in terms of its basic configuration, backup, and maintenance procedures. Friendstore is written in Java with 9664 lines of code and runs on Linux, Windows, and Mac OS X.

filename	<CID, chunk size (bytes) >	file attr
/d/foo.mp3	<E45F, 135434> <9ADF, 235>	...
/d/bar.mp3	<4D6A, 255136>	...

Metadata table: file-to-chunk mapping

remote id

CID	seqno	XOR ok?	expiration group id	replica set
E45F	1	yes	1	B, C
9ADF	2	yes	1	D, E
4D6A	3	yes	2	C, E

Metadata table: chunk-to-replica set mapping

Fig. 4. The metadata tables maintained at each owner for its collection of backed up files. Chunks, instead of files, are the units for backup operations.

6.1. Software Setup

The Friendstore software bootstraps a user's trusted neighborhood using existing social networks on Google and Facebook. A user configures Friendstore with a subset of Google or Facebook contacts with whom to perform cooperative backup. Other important configuration information includes a randomly generated DES key used to encrypt data, a list of directories to be backed up (the default is a user's home directory), the amount of donated disk space, and a desired throttle rate for upload bandwidth. Friendstore calculates s_{max} and d_{max} according to a user's configuration of the throttled upload bandwidth and a default availability of 0.5. The user can specify priorities for the directories. Directories with higher priorities will be replicated and restored earlier than others.

6.2. Backup

Friendstore runs as a background user-level process. It scans the backup directory daily to obtain a list of changed files according to the file modification time. The software splits large files into smaller chunks according to the chunking mechanism developed in LBFS [Muthitacharoen et al. 2001]. This chunking mechanism avoids transferring all chunks again if the user modifies, inserts, or deletes a few bytes of a file because only a few chunks get affected. Chunk identifier (CID) is the SHA-1 hash of its content. Therefore, we avoid redundant data transfers for files that have the same content but are stored with different names. Friendstore keeps a metadata table that maps each file to the sequence of chunks it contains (shown as the top table in Figure 4) so it can reconstruct files from chunks during restore. The maximum chunk size is chosen to be 10MB on average. Bigger chunk sizes incur smaller overhead for the metadata table but might cause more data transfer when big files change. The software follows the pipeline of steps to prepare changed files for transfer to remote neighbors as shown in Figure 1. Our implementation does not yet include compression.

To ensure maximal data privacy, an owner does not reveal SHA-1 hashes of the original data to remote helpers. Instead, it generates a remote identifier for each CID. The remote identifier is a concatenation of a monotonically increasing sequence number, an expiration group ID and an extra bit to indicate whether coding is desired for this chunk. The software groups smaller files into one expiration group to save the renewal overhead of small chunks; renewing the expiration time for any chunk during verification will automatically renew all chunks within the same group. In a second metadata table, Friendstore keeps the CID to remote identifier mapping as well as the

set of helpers that currently store a replica for the chunk (Figure 4). Because a helper only knows chunks' remote identifiers and not their CIDs, it cannot coalesce duplicate chunks from different owners as is done in Pastiche [Cox et al. 2002]. As studies have shown that duplicate data among a small set of users is small [Bolosky et al. 2000], we think the potential benefit of coalescing duplicate chunks is outweighed by its privacy implication. Both metadata tables in Figure 4 are encrypted and replicated on at least three helpers if possible.

The helper stores replicas belonging to different owners as ordinary files rooted at an owner-specific directory. For each owner, the helper keeps track of how much data that owner has stored, in order to calculate how many credits it has on that neighbor. The helper periodically computes check blocks only when its donated space is almost full.

6.3. Verify and Repair

The implementation of verification and repair follows in a straightforward manner from the design. The software includes additional features to reduce the need to regenerate replicas due to offline helpers. Users can gracefully shut down the Friendstore process which leaves a special IM status message to indicate that the node is temporarily unavailable as opposed to failed. Users can also tell Friendstore not to timeout certain neighbors if they are confident that those nodes' owners will inform any permanent failures promptly. For example, the no-timeout option is suitable for an intermittently connected laptop machine belonging to the same user.

7. EVALUATION

This section examines how well Friendstore performs as a backup service. The evaluations based on trace-driven simulations support the following conclusions.

—Although Friendstore sacrifices perfect global storage utilization to incorporate decentralized trust relationships, the actual utilization of donated disk space is high (>75%). Disk utilization further improves when nodes have more trusted neighbors.
—When bandwidth is plentiful, coding significantly reduces the amount of disk space each node must contribute in order to back up large amounts of data with replication.
—Friendstore can back up a significant amount of data reliably over a long period of time. The maximum backup capacity is limited by a node's upload bandwidth, but even with 150 kbps bandwidth and 81% availability, a node can back up 48GB with less than 0.15% loss rate over a five-year period. To ensure low loss rates, it is important for a node to limit the amount of data stored in the system according to s_{max}.
—Friendstore can use a lenient threshold of 200 hours to delay regenerating replicas on an unresponsive neighbor. This threshold effectively masks most transient node offline events while maintaining a low data loss rate. The number of replicas generated due to false timeouts is low.

We also evaluate the prototype implementation using microbenchmarks to show our software has good backup and restore throughput. Last, we present statistics from our small-scale deployment of 21 nodes over a period of two months and share our lessons learned from running Friendstore in practice.

7.1. Experimental Setup

Our trace-driven simulations use the following datasets.

Trust graph. We construct a trust graph by sanitizing a crawled Orkut social network [Li and Dabek 2006; Ford et al. 2006]. Social networks usually contain nodes with very high degree, and the degree follows the power-law distribution [Mislove

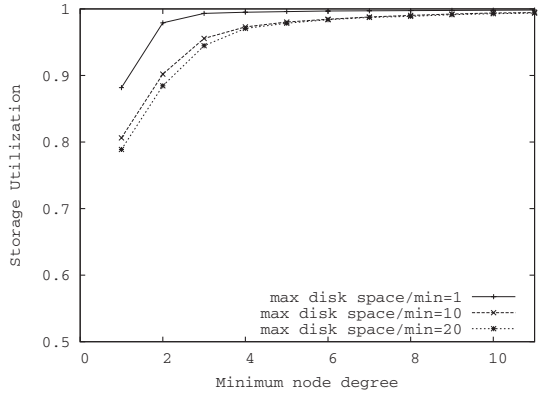

Fig. 5. Storage utilization as a function of minimum node degree in the Orkut graph. The various lines correspond to experiments where different nodes donate varying amounts of disk space. The donated space for each node is sampled from a uniform distribution with different maximum and minimum donation ratios.

et al. 2007]. Due to the open nature of social networks, not all friends in such networks are trusted friends. Friendstore only requires users to have a small number of trusted friends to participate in the system. Therefore, we sanitize the crawled Orkut graph by pruning edges so that the maximum node degree is 35. The resulting graph consists of 2363 nodes with median node degree 5 and average 4.7. The trust graph of a real large-scale Friendstore deployment could be different from the Orkut graph. Nevertheless, the Orkut graph has many interesting characteristics typical of all social graphs, such as the clustering effects. We also use a synthetic graph to check that our conclusions are not very sensitive to the specific social graph in use.

Node availability. We use the FARSITE trace [Bolosky et al. 2000] of the availability of corporate desktop machines to simulate transient node failures. This trace monitors 51, 663 desktops for a period of 840 hours. Our experiments on data durability span a five-year period, much longer than 840 hours. In the experiments, we randomly sample one complete node up-down event sequence from the FARSITE trace every 840 hours. The median simulated node availability based on the trace is 81%. This median availability is similar to that observed in our two-month deployment of Friendstore (75.3%).

Permanent failures. We generate a synthetic trace for permanent node failures. Using the model of disk failure rates in Schroeder and Gibson [2007], we use a Weibull distribution for interfailure arrival times with shape parameter 0.71. The average failure interarrival time is 11.1 hours for the population of 2363 nodes, with an average per-node lifetime of three years.

7.2. Storage Utilization

One concern with Friendstore is that its storage utilization might be low: a node might find out that all of its helpers are full even though available disk space exists elsewhere in the system. We show that Friendstore achieves good utilization when operating under typical social relationships.

Figure 5 shows the space utilization of Friendstore for the 2363-node Orkut graph. Since each helper contributes the same amount of disk space as its corresponding owner tries to consume, a homogeneous storage system (e.g. DHash [Dabek et al. 2004], Pastry [Rowstron and Druschel 2001b], OpenDHT [Rhea et al. 2005]) would be able to achieve perfect utilization. In comparison, Friendstore's utilization is less (87%). Most wasted storage space resides on nodes with very low degrees. As our crawled topology is only a subgraph of the Orkut network, nodes in this subgraph have lower degrees

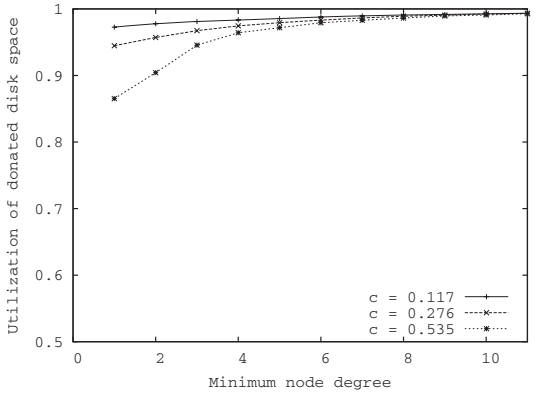

Fig. 6. Storage utilization as a function of minimum node degree for three synthetic graphs with different clustering coefficients (c). The ratio of maximum to minimum donation is 10.

than they do in the full graph. In particular, more than 23% of nodes have less than three neighbors in the subgraph while more than 95% of those nodes have degrees ≥ 5 in the full Orkut graph. We vary the minimum node degree by adding new links to the subgraph according to Toivonen et al. [2006], while preserving the original dataset's clustering coefficient of 0.23. Figure 5 shows that the space utilization increases quickly to reach more than 95% with ≥ 5 minimum node degree. Figure 5 also shows that the space utilization remains high even when there is a large variance in the amount of donated space by different nodes.

Figure 6 shows Friendstore's storage utilization using a synthetic trust network generated by the social network model in Toivonen et al. [2006]. We vary the clustering coefficient of the generated graph, c, from 0.117 to 0.535 (the Orkut graph has $c = 0.23$) and do not cap the maximum node degree. Bigger clustering coefficients correspond to graphs where a node's neighbors are more likely to have links among themselves. Graphs with smaller clustering coefficients tend to have higher space utilization. More importantly, Figure 6 shows that the overall space utilization is not very sensitive to the specific social graph in use.

7.3. Coding Benefits

Coding could significantly reduce the amount of donated space required to back up a certain amount of data. Figure 7 shows the amount of donated space required at each node in order for it to store a certain amount of backup data on others. All numbers on the x and y-axes are normalized by a node's backup demand which varies from 36 to 360 units. Without coding, each node must contribute more than twice the amount of backup space it consumes, in order to store 99% of data with a replication factor of two. The reason that a node does not back up 100% of its data when contributing twice as much space, is due to Friendstore's imperfect storage utilization. Using XOR(1,2), the amount of required contribution is reduced to 1.1 times the back up demand to backup the same amount of data with two replicas.

7.4. Long-Term Data Durability

We use the availability trace and the permanent failure model described in Section 7.1 in an event-driven simulator to examine the reliability of Friendstore over a five-year period. We ignore the limitation of donated disk space by letting each node contribute more space than needed and focus on how a node's limited upload bandwidth restricts its maintainable capacity. Unless otherwise mentioned, we simulate an asymmetric

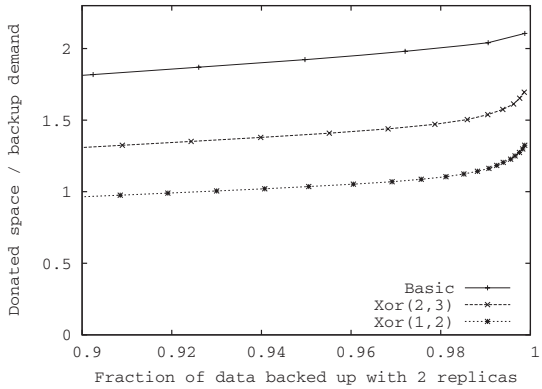

Fig. 7. The amount of donated space required in order to back up one unit of data in Friendstore with two replicas. Coding reduces the required space contribution.

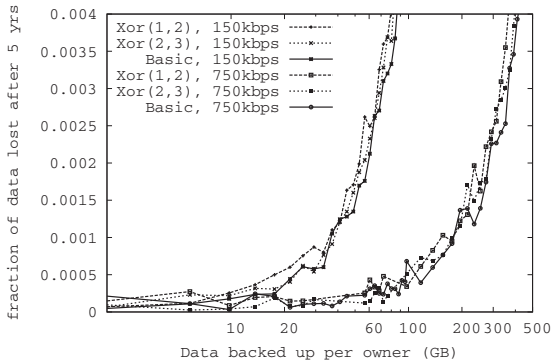

Fig. 8. The fraction of data lost after they were first backed up in the system five years ago as a function of the amount of backup data stored in the system by each owner. Experiments are done with different upload bandwidths, with and without coding.

DSL access link of 150 kbps upload bandwidth and 750 kbps download bandwidth. Whenever a node suffers a permanent failure according to the model, we delete all its data. The failed node rejoins the system five to six days later and becomes available according to a new availability sequence sampled from the availability trace. The newly joined owner will attempt to restore from its helpers immediately. All experiments use the default 200-hour timeout threshold unless otherwise mentioned. When measuring loss events, we only consider data stored in the system in the beginning of the experiments; newly inserted data have better reliability because they have been exposed to fewer permanent failures in their lifetimes.

Figure 8 shows the fraction of data lost at the end of five-years as a function of the amount of data each owner stores in Friendstore in the beginning of the experiments. If nodes do not backup at all, 81.2% of the data will be lost after five-years. We begin all experiments with one existing remote replica per data item so a perfect backup service would have zero data loss. In Friendstore, the loss rate increases as each node stores more backup data because it cannot promptly recopy replicas lost due to disk failures with limited upload bandwidth. According to Equation (1), $s_{max} = 3.1{*}10^6 \cdot B \cdot A = 48$GB for owners with 150kbps upload bandwidth and 81% availability, and $s'_{max} = 36$GB if coding is used. As we can see from Figure 8, when each owner stores no more than s_{max} GB of data, the probability of data loss after five years is very low ($< 0.15\%$). When an

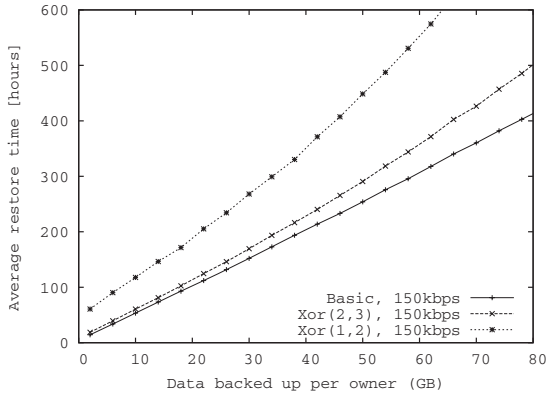

Fig. 9. Average restore time for a crashed node as a function of the amount of backup data for each owner, with and without coding. Note that restore time is limited by a node's download bandwidth, which is 750kbps in these experiments.

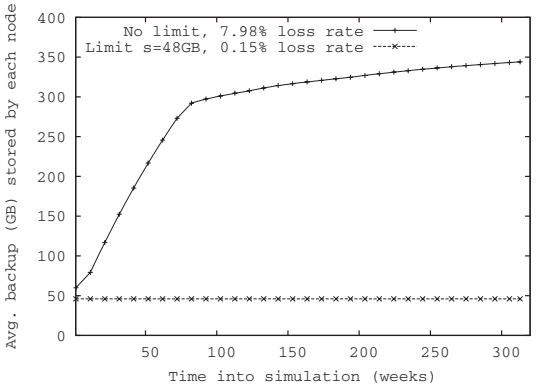

Fig. 10. The amount of data backed up per node as a function of time. Without any limit, nodes can store more data into the system over time, causing a high loss-rate of 7.89% as opposed to 0.15% with the s_{max} limit.

owner's upload bandwidth increases to 750kbps, s_{max} increases to 240 GB. Coding is able to achieve similar low loss-rate with a slight reduction in maintainable capacity. However, as Figure 9 reveals, coding results in increased restore time. The increase in restore time is due to node unavailability, as helpers must wait for owners to come back online before decoding check blocks. In our experiments, the XOR(1,2) and XOR(2,3) schemes increase the restore time by approximately 60% and 18% respectively.

To ensure data reliability, a node must limit the amount of backup data it stores in the system. We simulate a scenario where nodes do not attempt to limit the amount of backup data stored on remote helpers. These nodes simply transfer additional backup data whenever their upload links become idle. Figure 10 shows that without any limit, a node manages to store an increasing amount of backup data over time. The rate of increase flattens out after the first 70 weeks, as a node has fewer opportunities for uploading fresh data due to the increased burden of helping others restore, and maintain its remote replicas. After six-years of simulated time, 7.98% of data backed up five years ago are lost. In contrast, if nodes refrain from storing more than $s_{max} = 48$GB of backup data, the lost rate is less than 0.15%.

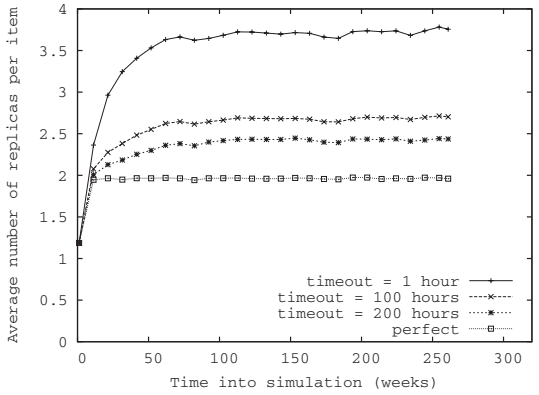

Fig. 11. The average number of replicas per data item as a function of time. A system with perfect knowledge creates exactly two replicas. A system using a timeout of 200 hours to detect permanent failures ends up creating 2.4 replicas on average.

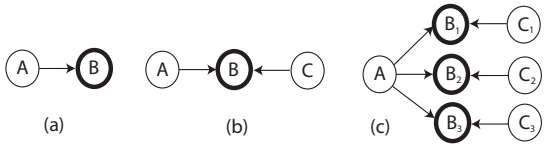

Fig. 12. Microbenchmark setup. In (a), owner A stores backup data on helper B without coding. In (b), helper B codes owner A and C's data together and store only check blocks. In (c), owner A stores backup data on three helpers B_1, B_2, B_3 that code A's data with owners C_1, C_2, C_3.

We proceed to explore the impact of different timeout values used to mask transient node offline events. Figure 11 shows the average number of replicas created per data item as a function of time. A system with perfect knowledge about transient failures never creates redundant replicas. In contrast, without perfect knowledge, nodes end up creating more replicas than needed over time. Fortunately, with the creation of each redundant replica, a node is less likely to regenerate replicas in response to a future timeout event [Chun et al. 2006]. Thus, the average number of replicas eventually stabilizes. Using a timeout of 200 hours, the system stabilizes at 2.4 replicas.

7.5. Microbenchmarks

We evaluate our prototype's performance using microbenchmarks. The machines used in the experiments have Pentium 3.4GHz CPU and 1GB memory and are connected to each other on a 100Mbps LAN. The experimental setup without and with coding is shown in Figure 12(a) and (b). Figure 13 shows our software's backup and restore time without coding and with coding. If the upload bandwidth at each node is not throttled (100Mbps), the prototype can achieve backup throughput of 300MB/28.3s = 10.6 MB/s without coding. The restore throughput is similar. With coding, the restore time takes longer, causing throughput to drop to 300M/92.6s = 3.2 MB/s because decoding check blocks becomes the bottleneck. When the available upload bandwidth is less than $<$ 25Mbps, there is no difference in backup and restore performance with or without coding.

For an asymmetric access link whose download bandwidth is larger than upload bandwidth, Friendstore achieves better restore time if an owner can restore from more than one helper simultaneously. Figure 14 shows the backup and restore times. The experimental setup is shown in Figure 12(c). Owner A restores data from 1, 2, or

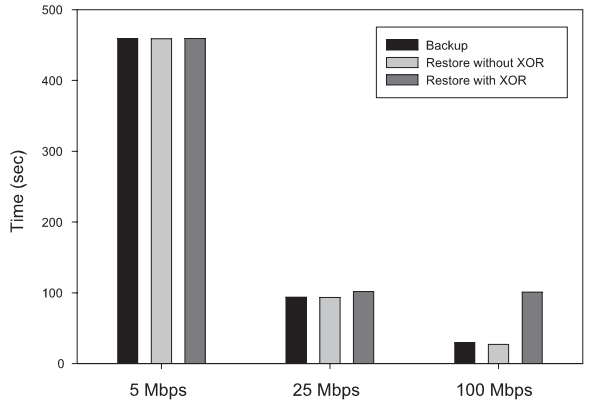

Fig. 13. Backup and restore times with different upload throttled bandwidths. The download bandwidth is 100Mbps.

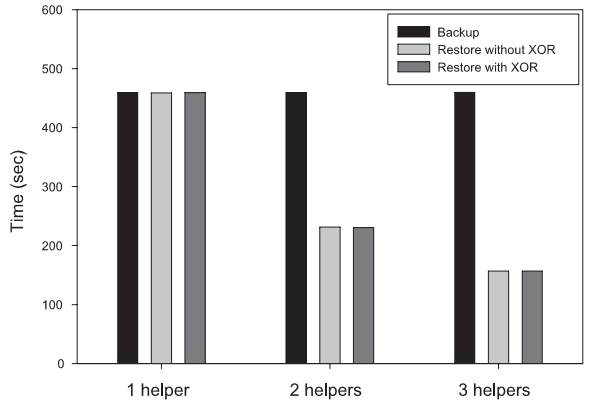

Fig. 14. Backup and restore time with 1, 2, or 3 helpers with 5Mbps throttled upload rate and 100Mbps download rate.

3 helpers involving up to seven nodes with coding. Each node throttles its upload bandwidth at 5Mbps. Backup time remains unchanged when the number of helpers increases because it is limited by *A*'s own upload speed at 5Mbps. On the other hand, restore completes three times faster when using three helpers to take advantage of node *A*'s larger download bandwidth (100Mbps).

7.6. Deployment Statistics

We have been running Friendstore in a small scale deployment involving 17 users and 21 nodes since August 1, 2007. The 21 nodes are a mixture of university desktops, home desktops, and laptop nodes running Windows, Mac, and Linux. Unfortunately, the deployed version of Friendstore did not automatically configure it to start across reboots. Many users manually changed Windows registry or edited startup scripts while some did not. We manually filtered out six offline events longer than a week that we suspected might be due to Friendstore's failure to launch after a reboot. Table II summarizes various statistics from the deployment. There is a wide range of upload bandwidths. We are encouraged to find that the median upload bandwidth usable by Friendstore is quite high (624Kbps) and that nodes are fairly available (median availability is 75%). A recent study of MSN Video clients also shows that many users have

Table II. Two Months Deployment Statistics of Friendstore from 08/01/2007 to 10/01/2007
Apart from the number of users, nodes, and maximum nodes per user, all statistics are calculated on a per-node basis and the resulting distribution is shown by the median number followed by 20- and 80-percentiles in parentheses.

Number of users	17
Number of nodes	21
Maximum nodes per user	3
Fraction of time online	75.3% (28.6%, 98.6%)
Max consecutive hours online	175 hours (53, 692)
Max consecutive hours offline	53 hours (13, 120)
Upload link bandwidth	624 kbps (211, 3744)
Number of neighbors per node	3 (1, 7)
Total amount of data backed up	578MB (275, 3077)

high access link speeds [Huang et al. 2007], suggesting that Friendstore's maintainable capacity is likely to be high in practice.

The pilot deployment has revealed a number of practical issues for which our prototype lacked good solutions.

—The deployed software displays a warning sign for users whenever a helper could not be reached during the past five days. We intended for a user to contact her friend to fix the problem when noticing these warnings. Instead, our users often just ignored warnings altogether. The software could be more useful if it could automatically identify the source of the problem and email the responsible user to suggest a fix.
—Our deployed software used existing social relationships collected by Google Talk and Facebook to help users configure trusted nodes. We are surprised to find out that many users do not have accounts with either of these popular services. This suggests that we will have to provide our own trust relationship registration service for a wider deployment.
—A few users expressed interest in using Friendstore to back up data for a large pool of machines that they administer. Since the deployed software lacks the notion of a group, these users cannot write a single backup policy to configure a large collection of machines easily.
—Many users prefer storing some files without encryption on trusted nodes so their friends can browse and view these files. This suggests that there is potential synergy between backup and file-sharing since both could use Friendstore as a generic replicated storage infrastructure.

The Friendstore software is currently undergoing its second major revision to address pitfalls observed in the deployment.

8. RELATED WORK

There are many proposals for distributed cooperative backup systems [Batten et al. 2002; Cox et al. 2002; Lillibridge et al. 2003; Aiyer et al. 2005; Cox and Noble 2003]. The need to deter nodes from behaving selfishly or maliciously is widely recognized in these systems [Cox and Noble 2003; Aiyer et al. 2005; Lillibridge et al. 2003; Rowstron and Druschel 2001b; Ngan et al. 2003]. A typical solution [Cox and Noble 2003; Lillibridge et al. 2003] is to follow the principle of tit-for-tat, which has worked well for peer-to-peer file sharing applications [Cohen 2002]. However, backup systems are different from file-sharing and a node that requests restore service following a disk crash does not have anything valuable to guarantee that others will grant her request. BAR-B [Aiyer et al. 2005; Li et al. 2006] relies on a central authority to enforce external disincentives for denying restore service, for example, expelling the offending node from the system. In

contrast, Friendstore uses the decentralized trust relationships among users to admit trustworthy nodes and enforce external disincentives.

Many systems have exploited the use of social relationships. For example, LOCKSS [Maniatis et al. 2005], a peer-to-peer digital preservation system, uses a friend list to bias weights for different peers' votes. Maze [Yang et al. 2004] and Turtle [Popescu et al. 2004] use social networks to provide better searches for peer-to-peer file sharing. Re [Garriss et al. 2006] lets users expand their whitelist for spam filtering to include the friends of friends. Peerspective [Mislove et al. 2006] provides better quality and more context-aware Web search results by aggregating friends' browsing histories. Many online reputation systems also use social networks to improve their accuracy [Hogg and Adamic 2004; Sabater and Sierra 2002; Marti et al. 2004]. In contrast to these systems, Friendstore uses social relationships for choosing a set of trusted nodes for reliable storage. Friendstore's idea of having a node explicitly pick out other trustworthy nodes resembles that in SPKI/SDSI [Ellison et al. 1986] and PGP certification chain. However, our notion of trust is different from that in certification systems. In Friendstore, trust reflects the expectation that a node belonging to a different administrative domain will maintain minimal availability and provide restore service as needed. CrashPlan [Crashplan] is recently released commercial software that allows users to back up data on friends' machines. Friendstore shares a similar structure but focuses on addressing two technical challenges: calculating maintainable capacity so that nodes with low bandwidth links do not back up more data than what can be maintained reliably, and storing more information in the limited disk-space using coding. These designs are not present in our earlier workshop paper [Li and Dabek 2006].

Using coding to improve space efficiency has been explored in RAID [Patterson et al. 1988; Gibson and Patterson 1993]. In particular, XOR(1,2) is similar to a level-5 RAID. Myriad [Chang et al. 2002] is an online disaster-recovery system resembling a cross-site distributed version of RAID. Plank et al. [2005] show some optimal coding schemes designed for a small number of original data blocks. This setting is similar to Friendstore's setting because the number of friends a node has is small. The difference is that all the checked blocks in Friendstore are at the same helper that leads to a different failure analysis compared with traditional erasure codes in which each block is at a different site. POTSHARDS [Storer et al. 2007] is a system that provides secure long-term archival storage. It uses the secret splitting technique, which is similar to erasure coding, to avoid reliance on encryption for long-lived data. In order for this technique to work, the number of colluding adversaries must be smaller than the number of shards required to reconstruct the data. POTSHARDS targets data that is read-only after being created, while Friendstore can support mutable data. HAIL [Bowers et al. 2009a] and PoR [Bowers et al. 2009b] are cryptographic frameworks that allow a set of servers to prove to a client a stored file is intact and retrievable. In the context of Friendstore, this problem is simpler because an owner in Friendstore keeps the original file. It is easy for the owner to check the integrity of any segment of the file. The communication in HAIL and PoR is different from that of Friendstore. In HAIL and PoR, the client can send a request to all the servers. In Friendstore, in order to check the integrity of a chunk, the owner only communicates with one helper, which stores the chunk. This helper may need to contact another owner when she cannot contact the first owner in order to produce the correct response if it coded the chunk.

There is a vast body of previous work on ensuring data reliability in replicated storage systems, especially for high churn and low bandwidth environments [Haeberlen et al. 2005; Bhagwan et al. 2004; Kotla et al. 2007; Chun et al. 2006; Rhea et al. 2003]. All systems go through similar tradeoff analyses to decide between replication and erasure coding and to choose a suitable replication level or coding rate. SafeStore [Kotla

et al. 2007] clients store data on autonomous storage service providers and periodically audit the providers to detect damage [Baker et al. 2006]. In contrast, Friendstore is a cooperative system where a node stores data on others as well as storing others' data [Cox et al. 2002]. Limited wide-area upload bandwidth is often a major concern for such cooperative storage systems [Blake and Rodrigues 2003]. Our analysis in Section 4 is directly inspired by Chun et al. [2006] and similar in spirit to Patterson et al. [1988]; Ramabhadran and Pasquale [2006]; Tati and Voelker [2006]. Using timeouts to delay responding to possibly transient failures is similar to lazy replication in TotalRecall [Bhagwan et al. 2004]. The timeout in Friendstore is to reduce wasted bandwidth in repairing a temporary offline node and hence has a different tradeoff than timeout in failure detectors [So and Sirer 2007]. Unlike massive replication in Glacier [Haeberlen et al. 2005], we find a small replication factor (e.g. $r = 2$) is sufficient to achieve low data loss in Friendstore.

A summary of this work has been published as a workshop paper [Tran et al. 2008]. That paper gives a more detailed description of Friendstore and presents the analysis and evaluation of the proposed techniques.

9. DISCUSSION AND FUTURE WORK

One concern about using a cooperative backup system is the possible loss of data privacy when a user stores her data on other nodes, even in encrypted form. As CPUs get faster and algorithms to break existing encryption schemes become better, it is only a matter of time before old data can be decrypted by an adversary. We believe that giving users the choice of where to store its backup data mitigates the risk of this potential privacy loss.

We are planning to deploy Friendstore on our departmental desktops many of which store large trace files in currently unbacked-up local scratch directories. The source code will be publicly available.

10. CONCLUSION

This article presents Friendstore, a cooperative backup system that gives users the choice to store backup data only on nodes they trust. Using trust based on social relationships allows Friendstore to provide a high assurance for reliable backup. Friendstore limits how much data a node stores according to its maintainable capacity and uses coding to store more information when disk space is the more limiting resource. Our initial deployment suggests that Friendstore is a viable solution for online backups. Friendstore is available online.[1]

APPENDIX

Choose Optimal Coding Schemes

In this section, we elaborate upon the discussions in Section 5 to show the optimality of coding schemes XOR(1,n) and XOR(n-1,n) and to argue that XOR(1,2) and XOR(2,3) achieve particularly desirable trade-offs.

Suppose a helper computes and stores k check blocks out of n original data blocks from n distinct owners. In order to restore data upon the failure of any one of the n owners, the helper needs to transfer at least $n - k$ original data blocks from the rest of the owners. The optimal coding scheme, OPT(k,n), transfers exactly $n - k$ blocks during restore. Note that OPT(k,n) is achievable using algebraic codes (e.g. Reed-Solomon) for arbitrary n and k. However, in order to apply homomorphic hash function to verify check blocks, we can only use linear codes whose check blocks are computed using the

[1]http://www.news.cs.nyu.edu/friendstore.

addition operation. In particular, XOR(1,n) and XOR(n-1,n) are optimal linear codes for $k = 1$ and $k = n - 1$.

Although XOR(1,2) and XOR(2,3) represent only two specific trade-off points, we argue that their trade-offs are particularly desirable among those achievable by all optimal codes. First, only codes OPT(k,n), where $k \leq 2$ are attractive for Friendstore. The alternative scheme, OPT(2+x,n+x), where $x \geq 0$, transfers the same amount of original data blocks $(n-k)$ during restore, and has a similar restore failure probability bounded by the probability that the helper fails during restore. However, OPT(2,n) has a smaller storage overhead; its ratio of check blocks over original blocks is $2/n$ compared to $(2+x)/(n+x)$ for OPT(2+x,n+x). When $x \geq 0$, $2/n \leq (2+x)/(n+x)$. Second, since many Friendstore owners have a small number of neighbors, practical coding schemes should compute check blocks from a small number of owners. For example in our deployment, the median node degree is only three. Taking into account both factors, we believe XOR(1,2) and XOR(2,3) achieve the most useful tradeoff points.

ACKNOWLEDGMENT

We thank Frank Dabek who greatly helped us to improve this article. We are grateful to Robert Morris, Frans Kaashoek, Jinyuan Li and Friendstore's early users for their encouragement and insightful comments. We also owe our gratitude to the authors of Adya et al. [2002] for making their trace available.

REFERENCES

Adya, A., Bolosky, W., Castro, M., Cermak, G., Chaiken, R., Douceur, J., Howell, J., Lorch, J., Theimer, M., and Wattenhofer , R. 2002. FARsite: Federated available and reliable storage for incompletely trusted environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*.

Aiyer, A., Avisi, L., Clement, A., Dahlin, M., Martin, J., and Porth, C. 2005. Bar tolerance for cooperative services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*.

Baker, M., Shah, M., Rosenthal, D., Roussopoulos, M., Maniatis, P., Giuli, T. J., and Bungale, P. 2006. A fresh look at the reliability of long-term digital storage. In *Proceedings of the SIGOPS European Conference on Computer Systems (Euro-Sys)*.

Batten, C., Barr, K., Saraf, A., and Trepetin, S. 2002. pstore: A secure peer-to-peer backup system. Tech. rep. MIT-LCS-TM-632, Massachusetts Institute of Technology.

Bhagwan, R., Tati, K., Cheng, Y., Savage, S., and Voelker, G. M. 2004. Totalrecall: System support for automated availability management. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Blake, C. and Rodrigues, R. 2003. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*.

Bolosky, W., Douceur, J., Ely, D., and Theimer, M. 2000. 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)*.

Bowers, K. D., Juels, A., and Oprea, A. 2009a. Hail: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (*CCS*). ACM, New York, NY, 187–198.

Bowers, K. D., Juels, A., and Oprea, A. 2009b. Proofs of retrievability: Theory and implementation. In *Proceedings of the ACM Workshop on Cloud Computing Security (CCSW)*. ACM, New York, NY, 43–54.

Chang, F., Ji, M., Leung, S.-T., Maccormick, J., Perl, S., and Zhang, L. 2002. Myraid: Cost-effective disaster tolerance. In *Proceedings of the 1st USENIX Conference on File and STorage Technologies*.

Chun, B.-G., Dabek , F., Haeberlen, Sit, E., Weatherspoon, H., Kaashoek, M. F., and Morris, R. 2006. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd Symposium on Networked System Design and Implementation (NSDI)*.

Cohen, B. 2002. Incentives build robustness in bitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*.

Cox, L. P., Murrary, C., and Noble, B. 2002. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*.

Cox, L. P. and Noble, B. 2003. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*.

CRASHPLAN. Crashplan: Automatic offsite backup. http://www.crashplan.com//.

DABEK, F., KAASHOEK, M. F., LI, J., MORRIS, R., ROBERTSON, J., AND SIT, E. 2004. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st ACM Symposium on Networked Systems Design and Implementation (NSDI)*.

ELLISON, C., FRANTZ, B., LAMPSON, B., RIVEST, R., THOMAS, B., AND YLONEN, T. 1986. Spki certificate theory. Internet RFC 2693 http://www.cis.ohio-state.edu/htbin/rfc/rfc2693.html.

FORD, B., STRAUSS, J., LESNIEWSKI-LAAS, C., RHEA, S., KAASHOEK, F., AND MORRIS, R. 2006. Persistent personal names for globally connected mobile devices. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

FU,Y., CHASE, J. S., CHUN, B., SCHWAB, S., AND VAHDAT, A. 2003. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*.

GARRISS, S., KAMINSKY, M., FREDMAN, M. J., KARP, B., MAZIRES, D., AND YU, H. 2006. Re: reliable email. In *Proceedings of the 3rd Symposium on Networked System Design and Implementation (NSDI)*.

GIBSON, G. AND PATTERSON, D. 1993. Designing disk arrays for high data reliability. *J. Parallel Distrib. Comput. 17,* 1–2, 4–27.

HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. 2005. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*.

HOGG, T. AND ADAMIC, L . 2004. Enhancing reputation mechanisms via online social networks. In *Proceedings of the 5th ACM Conference on Electronic Commerce*.

HUANG, C., LI, J., AND ROSS, K. 2007. Can Internet video-on-demand be profitable? In *Proceedings of the ACM SIGCOMM Data Communications Festival*.

KAMVAR, S. D., SCHLOSSER, M. T., AND GARCIA-MOLINA, H. 2003. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th International Conference on World Wide Web (WWW)*. ACM, New York, NY, 640–651.

KOTLA, R., ALVISI, L., AND DAHLIN, M. 2007. Safestore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference*.

KROHN, M., FREEDMAN, M., AND MAZIRES, D. 2004. On-the-fiy verification of rateless erasure codes for efficient content distribution. In *Proceedings of the IEEE Symposium on Security and Privacy*.

LI, H., CLEMENT, A., WONG, E., NAPPER, J., ROY, I., ALVISI, L., AND DAHLIN, M. 2006. BAR gossip. In *Proceedings of USENIX Operating Systems Design and Implementation (OSDI)*.

LI, J. AND DABEK, F. 2006. F2f: reliable storage in open networks. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS)*.

LILLIBRIDGE, M., ELNIKETY, S., BIRREL, A., AND BURROWS, M. 2003. A cooperative Internet backup scheme. In *Proceedings of the USENIX Annual Technical Conference*.

MANIATIS, P., ROUSSOPOULOS, M., GIULI, T., ROSENTHAL, D. S. H., AND BAKER, M. 2005. The LOCKSS peer-to-peer digital preservation system. *ACM Trans. Comput. Syst. 23*.

MARTI, S., GANESAN, P., AND GARCIA-MOLINA, H. 2004. DHT routing using social links. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*.

MISLOVE, A., GUMMADI, K. P., AND DRUSCHEL, P. 2006. Exploiting social networks for Internet search. In *Proceedings of the 5th Workshop on Hot Topics in Networks (HotNets)*.

MISLOVE, A., MARCON, M., GUMMADI, K. P., DRUSCHEL, P., AND BHATTACHARJEE, B. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC)*. 29–42.

MUTHITACHAROEN, A., CHEN, B., AND MAZIRES, D. 2001. A low-bandwidth network file system. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*.

NGAN, T.-W., WALLACH, D., AND DRUSCHEL, P. 2003. Enforcing fair sharing of peer-to-peer resources. In *Proceedings of the 5thInternational Workshop on Peer-to-Peer Systems (IPTPS)*.

PATTERSON, D., GIBSON, G., AND KATZ, R. 1988. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

PINHEIRO, E., WEBER, W.-D., AND BARROSO, L. A. 2007. Failure trends in a large disk drive population. In *Proceedings of the 5th Usenix Conference on File and Storage Technologies (FAST)*.

PLANK, J. S., BUCHSBAUM, A. L., COLLINS, R. L., AND THOMASON, M. G. 2005. Small parity-check erasure codes-exploration and observations. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*.

POPESCU, B. C., CRISPO, B., AND TANENBAUM, A. S. 2004. Safe and private data sharing with turtle: Friends team-up and beat the system. In *Proceedings of the 12th Cambridge International Workshop on Security Protocols*.

RAMABHADRAN, S. AND PASQUALE, J. 2006. Analysis of long-running replicated systems. In *Proceedings of the 25th IEEE Conference on Computer Communications (INFOCOM)*.

RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. 2003. Pond: The oceanstore prototype. In *Proceedings of the USENIX Conference on File and Storage Technologies*.

RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. 2005. OpenDHT: A public DHT service and its uses. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*.

ROWSTRON, A. AND DRUSCHEL, P. 2001a. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*.

ROWSTRON, A. AND DRUSCHEL, P. 2001b. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*.

SABATER, J. AND SIERRA, C. 2002. Social ReGreT, a reputation model based on social relations. *ACM SIGecom ExChanges* 3, 1, 44–56.

SCHROEDER, B. AND GIBSON, G. 2007. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean too you? In *Proceedings of the 5th Usenix Conference on File and Storage Technologies (FAST)*.

SO, K. AND SIRER, E. G. 2007. Latency and bandwidth-minimizing failure detectors. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. 2007. Potshards: Secure long-term storage without encryption. In *Proceedings of the USENIX Annual Technical Conference*. 142–156.

TATI, K. AND VOELKER, G. 2006. On object maintenance in peer-to-peer systems. In *Proceedings of the 5th International Workshop on Peer-to-peer systems (IPTPS)*.

TOIVONEN, R., ONNELA, J.-P., SARAMÄKI, J., HYVÖNEN, J., AND KASKI, K. 2006. A model for social networks. *Physica: Statis. Mech. Appl. 371*, 2, 851–860.

TRAN, D. N., CHIANG, F., AND LI, J. 2008. Friendstore: Cooperative online backup using trusted nodes. In *Proceedings of the 1st International Workshop on Social Network Systems (SocialNet)*.

YANG, M., CHEN, H., ZHAO, B. Y., DAI, Y., AND ZHANG, Z. 2004. Deployment of a large-scale peer-to-peer social network. In *Proceedings of USENIX WORLDS*.

Received February 2010; revised June 2010, November 2010, February 2011, June 2011, October 2011; accepted January 2012