A Cooperative Internet Backup Scheme

Mark Lillibridge   Sameh Elnikety   Andrew Birrell   Mike Burrows
Michael Isard
HP Systems Research Center[*]
Palo Alto, CA

Abstract

We present a novel peer-to-peer backup technique that allows computers connected to the Internet to back up their data cooperatively: Each computer has a set of partner computers, which collectively hold its backup data. In return, it holds a part of each partner's backup data. By adding redundancy and distributing the backup data across many partners, a highly-reliable backup can be obtained in spite of the low reliability of the average Internet machine.

Because our scheme requires cooperation, it is potentially vulnerable to several novel attacks involving free riding (*e.g.*, holding a partner's data is costly, which tempts cheating) or disruption. We defend against these attacks using a number of new methods, including the use of periodic random challenges to ensure partners continue to hold data and the use of disk-space wasting to make cheating unprofitable. Results from an initial prototype show that our technique is feasible and very inexpensive: it appears to be one to two orders of magnitude cheaper than existing Internet backup services.

1 Introduction

Traditional data backup techniques work by writing backup data to removable media, which is then taken off-site to a secure location. For example, a server might write its backup data daily onto tape using an attached tape drive; at the end of each week, the resulting tapes would then be picked up by a truck and driven to a guarded warehouse. The main drawback of these techniques is the inconvenience for system owners of managing the media and transferring it off-site, especially for small installations and PC owners.

In contrast, Internet backup sites (*e.g.*, `www.backuphelp.com`), avoid this inconvenience by locating the tape or other media drive in the warehouse itself and by using the Internet instead of a truck to transfer the backup data. Customers need only install the supplied backup software to be assured that, so long as their system remains connected to the Internet, their data will be automatically backed up daily[1] without any further action on their part. These sites charge by the month based on the amount of data being backed up. For example, a typical fee today to backup up one gigabyte of data is fifty US dollars a month (see Section 5.1).

In this paper we propose a new Internet-based backup technique that appears to be one to two orders of magnitude cheaper than existing Internet backup services. Instead of relying on a central warehouse holding removable media, we use a decentralized peer-to-peer scheme that stores backup data on the participating computers' hard drives.

To provide for off-site storage, we arrange for pairs of geographically-separated participating computers (*partners*) to swap equal amounts of disk space—a fair trade. To compensate for the fact that Internet PCs are much less reliable than a tape stored in a secure facility, we have each computer partner multiple times so it can spread its backup data in a redundant manner across many machines. By using a large number of partners per computer, we can ensure high reliability with low space overhead.

Our scheme requires the cooperation of the participating computers: computers depend on their partners to hold their data and make it available when needed. In an uncontrolled environment like the Internet, such cooperation cannot be taken for granted. Non-cooperation must be discouraged by making it unprofitable. We use several novel methods to do this, including the use of periodic random challenges to ensure partners continue to hold data (partners that fail are abandoned in favor of new partners) and the use of disk-space wasting to make fake crashes unprofitable.

The remainder of this paper is organized as follows: Section 2 describes a simplified version of our scheme that assumes cooperation can be taken for granted. It is well suited for systems that are intended to be deployed within a single company. Section 3 tells how to extend the simplified scheme to an environment where cooperation cannot be assumed, such as the Internet, by adding various security mechanisms. Section 4 presents results from an initial prototype. Section 5 compares our system to existing Internet backup sites as well as traditional backup techniques. Section 6 covers related work. Finally, we present our concluding remarks in Section 7.

---

[*]Current affiliations: Elnikety, Rice University; Birrell, Burrows, Isard: Microsoft Research.

2 The simplified scheme

Each computer that wishes to participate in our backup scheme runs special software. Under the software's direction, these computers link up and form a peer-to-peer system over the Internet or a corporate intranet. The same software, performing the same functions, runs on each computer—except for a single external matching server (see Section 2.1), the system is decentralized and functionally symmetric. Like most peer-to-peer systems, computers are free to join or quit the system at any time.

Each participating computer has some number of backup partners. For example, $\mathcal{A}$ might have partners $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$. Partnership is a symmetric relation: $\mathcal{A}$ is also a partner of $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$. Partnership is not, however, transitive: $\mathcal{B}$ and $\mathcal{C}$ need not be partners, and in general $\mathcal{B}$ and $\mathcal{C}$ may share no partners other than $\mathcal{A}$.

How many and which partners a given computer has varies over time. Computers start with zero partners on joining and quickly add enough partners to handle their current backup needs. As their backup needs change, they may want to add or remove partners. Partners may also be changed if an existing partner is found to be wanting (*e.g.*, due to excessive downtime) or a new computer needs partners.

Each pair of partners agrees at partnership-formation time to an amount of storage to be swapped and a level of uptime (time that they are running and connected) that they must maintain. Different pairs may reach different agreements. Suppose $\mathcal{A}$ and $\mathcal{B}$ agree to swap s blocks. Then each must reserve s blocks of their local disk for use by each other. The software, running in the background, performs reads from and writes to this space on behalf of requests from the other partner.

2.1 Finding partners

We suggest using a simple central server to keep track of the computers in the system and their partner needs. Many other methods of finding partners are possible—for example, a Gnutella-like flooding approach could be used—but the central-server method has the advantage of being very simple to implement.

Each computer should periodically update the server with its identity and what partners it needs and has, including uptime and storage-swapping levels. When a computer needs a new partner, it contacts the server with its needs and obtains a list of candidate partners; it can then contact those computers directly and find out if they are still compatible.

Sometimes there may be no other computers looking for new partners. In that case, a computer looking for new partners needs to step between two existing partners that have an agreement similar to the one it desires: if $\mathcal{A}$ and $\mathcal{B}$ are partners, $\mathcal{N}$ can step between them so that $\mathcal{A}$ now has $\mathcal{N}$ for a partner instead of $\mathcal{B}$ and $\mathcal{B}$ now has $\mathcal{N}$ as a partner instead of $\mathcal{A}$. This leaves $\mathcal{A}$ and $\mathcal{B}$ with the same number of partners, but gives $\mathcal{N}$ two new partners of the type it wants. By having $\mathcal{N}$ copy $\mathcal{A}$ and $\mathcal{B}$'s data beforehand, this can be done atomically with no data loss.

To avoid complicated negotiation, we suggest appropriately quantizing uptime and storage-swapping levels. Ideally, to work well, the system should have many (at least a hundred, preferably more than ten thousand) members spanning the range of possible agreements. Computers wishing to swap huge amounts may still be out of luck finding compatible partners, but can compensate (with somewhat lower reliability) by using multiple partners swapping less each.

It is important that each partner in a pair be located at different sites in order to ensure all backups are stored off-site. Accordingly, computers should reject candidate partners that are co-located. This means that our scheme cannot be used safely within a single site. Additional reliability can be obtained by further diversification: a single computer should choose its partners from as many different sites and using as many different operating systems (to guard against viruses) as it can. To allow this, the identity information supplied to the central server should include a computer's "location" and operating-system type. Location information can either be obtained directly from the computer owner or estimated via IP ranges or domain-registry information.

Although the central server forms a single point of failure for finding new partners, it need keep no permanent state and is thus easily replaced or replicated should it fail or become a bottleneck. Its failure does not prevent backups or restorations from occurring; thus, as long as it is repaired within a reasonable amount of time (*i.e.*, weeks), no real harm is done.

2.2 Creating a reliable logical disk

We use Reed-Solomon erasure-correcting codes [13] to create a highly-reliable logical disk from a large number of partners. A $(k+m, m)$–Reed-Solomon erasure-correcting code generates m redundancy blocks from k data blocks in such a way that the original k data blocks can be reconstructed from any k of the $k+m$ data and redundancy blocks. By placing each of the $k+m$ blocks on a different partner, the logical disk can survive the failure of any m of the $k+m$ partners without any data loss, with a space overhead of m/k.

Erasure-correcting codes are more efficient than error-correcting codes because they handle only *erasures* (detectable losses of data) rather than the more general class of *errors* (arbitrary changes in data). Block errors can be turned into block erasures by attaching a checksum and version number to each stored block; all but the blocks

$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{E}$	$\mathcal{F}$	$\mathcal{G}$
0	1	2	3	$R_{0,1,2,3}$	$R'_{0,1,2,3}$
4	5	6	7	$R_{4,5,6,7}$	$R'_{4,5,6,7}$
8	9	10	11	$R_{8,9,10,11}$	$R'_{8,9,10,11}$
⋮	⋮	⋮	⋮	⋮	⋮

Figure 1: Sample block layout using 6 partners ($\mathcal{B}$-$\mathcal{G}$) with $k = 4$ and $m = 2$. $R_{w,x,y,z}$ denotes the first redundancy block and $R'_{w,x,y,z}$ the second redundancy block generated from data blocks w, x, y, and z.

k	m	n	Reliability	Overhead
6	0	6	53.144%	0%
6	1	7	85.031%	17%
6	2	8	96.191%	33%
6	3	9	99.167%	50%
6	4	10	99.837%	67%
6	5	11	99.970%	83%
6	6	12	99.995%	100%

Figure 2: Reliability and overhead for increasing values of m, holding k constant at 6, and assuming an individual computer reliability of 90%.

with correct checksums and the highest version number are considered erased.[2]

So that we can use a small fixed block size, we stripe blocks across the partners using the erasure-correcting code. See Figure 1 for an example block layout for a computer $\mathcal{A}$ with 6 partners that is using $k = 4$ and $m = 2$. We place corresponding outputs of the erasure-correcting code on the same partner (*e.g.*, $\mathcal{F}$ holds all the first redundancy blocks in Figure 1) to make reconfiguration easier (see Section 2.6).

Each computer decides for itself how to tradeoff reliability against overhead by choosing values for k and m; these choices in turn determine the number of partners $n = k+m$ it needs. To get a feel for how these tradeoffs work in practice, see Figures 2 and 3. Figure 2 shows how reliability rapidly and overhead slowly increase as the number of redundancy blocks (m) is increased while holding the number of data blocks (k) constant. Figure 3 shows that the overhead can be decreased for a given level of reliability by increasing the number of partners (n=k+m). (Unlike traditional RAID systems, we can use high values of m and n because backup and restoration are relatively insensitive to latency.)

These figures were calculated via the binomial distribution assuming that individual Internet computers fail independently and are 90% reliable. More precisely, they assume that when a computer tries to restore its data, the probability that a particular one of its partners still has its data, uncorrupted, and is sufficiently available during a limited restoration time window to supply that data is 90%.

k	m	n	Reliability	Overhead
6	6	12	99.995%	100%
8	7	15	99.997%	88%
10	8	18	99.998%	80%
12	9	21	99.999%	75%
14	9	23	99.997%	64%
16	10	26	99.999%	63%
18	10	28	99.996%	56%

Figure 3: Reliability and overhead for increasing values of k, using the minimum value of m necessary to get a reliability of at least 99.995% and assuming an individual computer reliability of 90%.

This number is meant to be conservative; we expect from our personal experience that the real number will be considerably higher. Indeed, the only empirical study we know of on PC availability, Bolosky *et al.* [3], found that over half of all Microsoft's corporate desktops were up over 95% of the time when pinged hourly. As that study included some PCs that are shut down at night or over the weekend and a reasonable restoration window would probably be at least 24 hours, their numbers underestimate restoration availability for machines promising 24-hour availability. Nonetheless, even when using our conservative number, our calculations show that high reliability can be achieved with low overhead.

We expect randomly-chosen Internet PCs to fail independently except in cases of widespread virus damage, sustained loss of Internet connectivity, and (in the case of uncooperative environments) coordinated attacks involving multiple computers. See Section 3.3 for discussion of why we believe the later is unlikely to be a problem in practice.

2.3 Backing up data

Each computer backs up its data on the reliable logical disk it has constructed from its partners' disks. Exactly how this is done—*e.g.*, incrementals vs. full backups, compression, ignoring caches and application binaries, *etc.*—is orthogonal to our scheme; we assume here only that backups occur at most daily. Because our backup space is of limited size and somewhat expensive compared to removable media, it may be useful to conserve space. In particular, one may not want to require backup space for two full snapshots so that a crash while writing a new snapshot does not leave the system without a viable backup. We demonstrate one way of doing this in Section 4.1.

The following procedure can be used to stream a snapshot to logical disk starting on a block stripe boundary: For each k data blocks of the stream, perform the follow-

ing operations in turn: use the erasure-correcting code to generate the m redundancy blocks from the data blocks, generate and attach a checksum and the same new version number to each of the $k+m$ blocks, send a request to write each block to the appropriate partner in the appropriate place, and, finally wait for acknowledgments from at least $w \geq \max(k, m+1)$ partners. It is important that we write at least k blocks to the current block stripe before starting to write the next one to ensure that a crash while writing will leave at most one block stripe unreadable; we need to write at least $m+1$ blocks to ensure that the old version is overwritten.

The parameter w here represents a tradeoff. Smaller values of w allow the backup to proceed faster because it is not necessary to wait for as many partners to be up, but the resulting written data has lower reliability than normal because some of the blocks are missing: only $w-k$ failures can be tolerated before some of the written data is unrecoverable. w should be chosen based on empirical data and the uptime-level agreements being used. Note that if more than $n-w$ partners fail, it will no longer be possible to make new backups with this procedure until some of the failing partners have been replaced. Alternatively, w could be updated based on the number of partners scheduled to be replaced.

We run a *cleaner* in the background on each computer to help limit how long recently written data has less than the maximum redundancy available. The cleaner scans its logical disk looking for incompletely-written block stripes. Each time it finds such a block stripe, it reads as many blocks as it can from it and tries to decode the stripe. If it succeeds, it generates the missing blocks and writes them to the appropriate partners, thus increasing the stripe's redundancy. Note that both the cleaner and streaming procedure use only a block stripe worth of extra local storage, avoiding the need for an extra snapshot worth of temporary disk space during backing up.

2.4 Restoring data

Restoration can be done from any computer in the event of the backed-up machine's total destruction. The backed-up computer's logical disk can be recovered to the new computer's local disk given a list of the original computer's current partners by using the following procedure: Contact each partner and ask for all of the backed-up computer's data. For each block stripe, attempt to decode using the erasure-correcting code the blocks with valid checksums and the highest version number in that stripe. If you succeed, write the resulting data blocks to local disk in the appropriate places. Keep repeating this process, retrying partners that were down, until additional blocks cannot result in more stripes being successfully decoded or time runs out.

It is the responsibility of the backed-up–computer maintainer to keep one or more copies of the list of current partners off-site in a security box or the like. This list is generated shortly after joining once the initial set of partners has been determined and updated occasionally as partners change.

To limit how often this list must be updated, we store the list of current partners in a special block (the *master block*) that is replicated on each partner and not part of any block stripe. This means that the list can be retrieved from any current partner so that the off-site list actually needs to be updated only every $k-1$ partner changes under the assumption that we must tolerate m partners failing. If this is still too frequent, it is possible to add many additional partners that we only swap master blocks with.

2.5 Handling downtime

In the real world computers are often unavailable: they may be connected via a dialup line or suffer from frequent soft failures (*e.g.*, Windows crashes). Partners must agree on a level of required uptime (*e.g.*, "up 90% of the time" or "up during California business hours").

Lower levels of partner uptime decrease performance: backups and restores take longer because the computer must wait for partners to become available. For example, if a machine's partners are up only during business hours and it crashes during the weekend, no restore will be available until Monday morning. Efficient backups require most partners to be up simultaneously during some period of the day. This limits the ability of computers with low and unpredictable uptime to participate in our scheme.

Agreements are subject to being broken. For the simplified scheme, we assume that owners are not out to take advantage of or hurt others. We do not, however, assume that owners are reliable about maintaining uptime agreements. Owners might forget to leave their computer on as much as planned, underestimate how often their machine crashes, or change their computer-usage policy without remembering to tell the backup-system software.

To guard against this, each computer keeps track of its uptime and warns its owner when it is failing to live up to its end of its agreement. For the simplified scheme, we assume this reminder is sufficient to make the owner take any needed steps to correct the problem. In the full scheme (see Section 3), we actively police agreements (both uptime and storage swapping) and abandon partners who fail to live up to their end of an agreement.

2.6 Resizing the amount of backup space

Consider a computer currently swapping s blocks with each of $n = k+m$ partners. In return for $n \times s$ blocks

of local disk, it has access to a logical disk of size $k \times s$ blocks. If it needs additional logical-disk space, it can either add more partners swapping s blocks each (presumedly maintaining a similar ratio of k and m) or switch to n new partners willing to swap $s' > s$ blocks each. Adding partners increases the amount of overhead due to per-partner costs (especially under the full scheme where we must periodically check on each partner), but requires issuing a new current partner list less often.

The same methods run in reverse can be used to shrink the logical disk. Partners holding redundancy blocks can also be added or removed to adjust the reliability level. Most of these changes require moving data around to maintain a sequential image (*i.e.*, adding partners adds blocks to every stripe row, rather than just adding a bunch of blocks at the end of the disk). By using the master block and version numbers, this can be done using no extra space in a restartable way with restoration always possible.

3 Security

In the previous section, we described a simplified scheme that assumes system members can be relied on to cooperate with each other, either because of substantially-similar interests or some external enforcement regime. We believe this assumption is likely to hold for systems deployed within a single company. Care should be taken, however, if our scheme is used within a single company to ensure sufficient site diversity so that all partnerships can be between sites.

In this section we describe how to extend the simplified scheme so that it can function in an environment such as the Internet where cooperation cannot be assumed because computer owners have different and possibly conflicting interests. Systems operating in such environments must be able to defend against members attempting to read or alter other members' data, to unfairly take advantage of other members, and to shut down or impair the system.

3.1 Confidentiality and integrity

To ensure the confidentiality of its backup data, each computer should encrypt its data before sending it to its partners using symmetric cryptography with a secret key known only to it. Because this and the other keys described below are needed for restoration, they should be added to the current-partners list that is manually taken off-site.

Ensuring backup integrity requires three steps. First, third parties must be prevented from impersonating a computer to one of its partners so that they can overwrite that computer's data. This requires pairwise authentication. At partnership formation time, the two partners can use Diffie-Hellman to establish a shared secret key, which they can then use later to authenticate write messages by attaching a sequence number and keyed cryptographic hash to each message.

Second, partners must be stopped from modifying a computer's data by altering a block's data then fixing up its checksum. This can be prevented by substituting a keyed cryptographic hash for the simple checksum used by the simplified scheme. So long as a computer keeps this hash key (an *integrity key*) secret, no other party will be able to modify or generate new valid blocks. Like with the encryption key, there is no need to have separate integrity keys for each partner.

Third, the ability of a computer's partners to conspire to replace one valid block with another must be limited. Computing a block's cryptographic hash over the partner ID and block offset where that block is stored in addition to its portion of the backup data and version number will prevent all substitutions except those involving an earlier version of the same logical-disk block. By storing the date and version number of each snapshot in the master block and refusing to accept earlier versions at restoration time, a computer can ensure that conspirators can not selectively revert parts of a snapshot. A conspiracy of at least k partners can still revert the entire snapshot to a previous version; the only possible defense is to print the date of the actual snapshot being restored in the hope that the owner will notice the reversion.

The order in which encryption and checksum attachment are done matters. The correct order is to (1) generate the redundancy blocks, (2) encrypt each block, (3) attach the version number, (4) compute a cryptographic hash for each block (over the encrypted data, version number, partner ID, and block offset), and (5) attach the appropriate cryptographic hash to each block. This order allows a block's validity to be checked without having to decrypt it. More importantly, it ensures that there is no exploitable redundancy available to attackers: if encryption was done before generating redundancy blocks, a partner could save space by using the erasure-correcting code to reconstruct the data he was supposed to store from the data stored at the other partners. While good for him, that leaves the backup with lower redundancy.

3.2 Free-rider attacks

Peer-to-peer systems, including ours, are potentially vulnerable to *free-rider* attacks. A free-rider attack is one where an attacker, called a free rider, benefits from the system without contributing their fair share. The classic example of a free rider is a person who watches the US Public Television System (PBS) without donating any money. (PBS is supported largely by viewer donations.) Systems vulnerable to free-rider attacks either run at reduced capacity or collapse entirely because, as more and more users free ride, the costs of the system weigh more

and more heavily on the remaining honest users, encouraging them to either quit or free ride themselves.

3.2.1 Agreement violations

The most basic free-rider attack against our scheme is for an attacker to intentionally fail to uphold his end of his agreements. For example, under our simplified scheme a computer could free ride by letting its partners backup its data but refuse to hold their data in turn; this would give the computer backup service at essentially no cost to itself.

To prevent such attacks, participating computers should police their agreements by verifying whether or not their partners are honoring their promises. Each computer can periodically challenge each of their partners to make sure that the partner in question is up when promised and continuing to hold the data it agreed to hold.

Such a challenge might consist of a request for the block of the challenger's data stored at a challenger-chosen random offset; the answer would then be checked to make sure it is a valid block that belongs on that partner at that offset. Optionally, the block's version number could also be checked to make sure it is the most recent version. Because a partner who holds only fraction d of the challenger's data will pass c challenges with probability d^c, by challenging frequently enough, the challenger can be assured with high probably that its partner is still holding almost all of its data.

While a challenge remains pending (*i.e.*, not yet answered correctly), the challenger should keep retrying it until either it is answered correctly or the partner claims data loss (aka, needs restoration). After a computer has restored its data, it signals its partners, who then reload their data (empty blocks until their cleaners have a chance to run) on it and resume challenging it. The time the challenger spends waiting for an answer should be counted as downtime for that partner. Partners who are down too often (relative to their uptime-level agreement) or need restoration too often should be forever abandoned in favor of a new partner.

Unfortunately, a crashed computer looks just like a cheating one that is trying to dodge a challenge it cannot answer—both do not respond to requests. If computers using our scheme abandoned partners (discarding their backup data) as soon as they were clearly down more than their uptime agreements allow (say, 8 hours for a strict 100% agreement), our backup service would not be very useful because backups would likely be discarded before computer owners realized they were needed.

Accordingly, we suggest allowing a *grace period* of two weeks: partners should not be abandoned until two weeks have passed since they first went down excessively. We recommend two weeks to cover the case where a machine crashes, losing data, just after the owner leaves on a two week vacation.

3.2.2 Exploiting the grace period

A more sophisticated free-rider attack involves taking advantage of the grace period to obtain backup service for free. The attacker joins the system, forming partnerships and exchanging data as normal. He then pretends to crash, throwing away all the data he has been given. For the next 2 weeks, he has free backup service because of the grace period. Just before the end of the grace period, when his partners will stop giving him backup service, he switches to a new set of unsuspecting partners and starts again. So long as he can find new partners (peer-to-peer systems can have millions of members), he can continue to receive backup service without cost.

Another free-rider attack involving the grace period has the attacker refusing to wait out the grace period before abandoning partners that are excessively down. This hurts his partners because they are left with a less redundant backup, or even no backup at all if enough of their partners free ride this way; however, the attacker benefits because his backup has better redundancy for the next two weeks because of the additional new partner.

The only way to deter these attacks is to make them unprofitable: we need to arrange things so that the attacker pays more for the privilege of using the grace period than it is worth to him and so that the attacker saves money by waiting out the grace period without abandoning his partners. (Free riders are motivated by the chance to save money, not the opportunity to hurt others.)

Payment If our scheme was modified to use a hybrid peer-to-peer model where a company charged money to use the software, collecting payment would be easy: just add an extra amount to the monthly bill. Likewise, if low-cost electronic financial transactions were available (*e.g.*, digital cash), payment could consist of a straightforward transfer of money between the attacker and the hurt parties or a suitable charity.

Because we wish to allow for a decentralized peer-to-peer system with almost no administration beyond keeping the matching server running and do not believe low-cost transactions are likely to become available anytime soon, we consider here a different way to make attackers pay: disk-space wasting.

The idea behind disk-space wasting is that we need to impose a cost that balances out the benefit the attacker stands to gain, in this case 2 weeks of a partner holding s blocks of his data. How much is this benefit worth? Clearly, no more than it would cost to obtain it from the next cheapest source, namely our scheme used honestly: the effort required to host s blocks for 2 weeks. Thus forcing someone to waste s blocks of disk space for 2

weeks cancels any benefit they might receive from abusing the grace period.

Assuming computers $\mathcal{A}$ and $\mathcal{B}$ are already swapping s blocks, $\mathcal{A}$ can pay such a cost to $\mathcal{B}$ by holding an additional s blocks for $\mathcal{B}$ for two weeks, by holding an additional $2 \times s$ blocks for one week, or by allowing $\mathcal{B}$ to hold s fewer of $\mathcal{A}$'s blocks for two weeks. The previously described protocols are used to allow $\mathcal{B}$ to read, write, and check on her additional blocks. Note that in the fewer blocks case that $\mathcal{B}$ is no longer holding $\mathcal{A}$'s backup data for the duration, leaving $\mathcal{A}$ with one fewer effective backup partner. Also notice that unlike the monetary-payment cases, disk-space payments take time to make—one or two weeks in the examples here.

Such payments may benefit $\mathcal{B}$ because she has more storage available to her, and thus represent a transfer of value from $\mathcal{A}$ to $\mathcal{B}$. A different kind of payment is a *commitment-cost* payment where $\mathcal{B}$ ensures that $\mathcal{A}$ either pays an unrelated third party (*e.g.*, a charity in the monetary-payment case) or else simply wastes resources (the disk-space–wasting case). We can convert the previous transfer examples into commitment-cost payments by using *low-utility blocks*.

A low-utility block is a normal block whose access has been sufficiently restricted so that it is of little or no utility to its lessee. For example, the lessee might be allowed to write any time, but be able to read only an occasional random one of its low-utility blocks for checking purposes. More draconian would be allowing only reads of random *words*. A transfer can be turned into a commitment-cost payment by either making the additional blocks held be low-utility blocks or by instead of having the partner hold s fewer normal blocks, have her convert those s normal blocks to low-utility blocks for the duration. The low-utility–block lessee stores uncompressible (*i.e.*, encrypted) data in those blocks and checks a random block (or word) periodically to ensure that the data is being kept. The parties must mutually agree on the random block (or word) to be read—see Section 3.3.1 for a protocol to do this—to guard against the lessor always presenting the same block for inspection.

Because we believe that most PCs will not have enough space to hold a large number of extra blocks beyond the ones already needed for backup space, we will assume for the rest of this paper that the "allow your partner to hold s less normal blocks" cases are used for disk-space wasting. While this requires no extra space, it does have the drawback of leaving a PC with no backup while it is paying multiple partners. As an optimization, PCs paying commitment costs may continue to backup their data onto their (temporarily) low-utility blocks so that it is immediately accessible once payment is complete and the access restrictions are removed; this optimization is incompatible with restricting access to random words because only entire backup blocks are verifiable.

Prepayment We can make these free-rider attacks unprofitable by requiring prepayment for the right to abuse the grace period: each time a computer gets a new partner, it pays a commitment cost of "3 weeks" to it and after being restored after d days of downtime, it must pay "$d{+}1$ days" to its partners to keep them. Here, a cost of "1 day" is shorthand for disk-wasting for 1 day with the same amount of storage swapped or the equivalent via monetary transfers to a clarity or central billing authority. Note that if we do not use commitment-cost payments here, the payments between two new partners would cancel out.

This scheme clearly makes the backup-service-for-free attack unprofitable. The case for the refusing-to-wait-out-the-grace-period attack is more subtle: If the attacker switches immediately, he pays "3 weeks" for the new partner. If he waits instead, he might have to pay up to "2 weeks" for the grace period plus a possible additional "3 weeks" if the partner does not resume swapping data with him after restoration. So long as the probability of his partner resuming swapping is more than $2/3$, it will be cheaper for him to wait. Should the probability (q) turn out in practice to be less than this (unlikely), a larger new-partner fee of "2 weeks"$/q$ will still make the attack unprofitable.

The prepayment scheme has the advantages of being very simple and robust, requiring no assistance from the central server or any assumptions about the difficulties of changing computer identities. Its main disadvantage is that when disk-space wasting is used it interferes with backup service: backup service is not available for the first 3 weeks after joining the system, for up to 2 weeks after a restoration, and additional backup space takes 3 weeks to become available (new partners are needed). While growth in the backup space needed can usually be anticipated, the growth-speed limitation may be problematical in some cases.

Post-payment An alternative scheme is based on requiring payment *after* using the grace period (with or without a restoration). Here the central computer keeps track of each computer's partners. Each computer is supposed to honor the grace period and pay "$d{+}1$ days" to its partners after being restored after d days of downtime. If it does not, its partners will complain to the central server, causing the server to sever those partnerships and impose a fine of "3 weeks" on all of the parties.

The fine must be imposed on everyone because, in general, there is no way for the central server to know who is truly at fault. This is unfortunate because it introduces a new free-rider attack: don't bother to complain,

letting others shoulder the burden of deterring attackers alone. We believe that this last attack will not be serious because it is only really tempting when there are a lot of attackers exploiting the grace period, which should not happen if most computers act to deter them by complaining.

The central server must impose the fine (*e.g.*, it supplies the data and does the challenging) because an attacker's partners (new or old) may be accomplices that will not fine it. Should a computer refuse or try to cheat the fine, it is exiled by the central server from the system: the central server tells the machine's partners to abandon it and refuses to authorize any new partnerships for it ever again.

In order for the threat of exile to be an effective deterrent, rejoining the system under a new name must cost more than "3 weeks" times the maximum number of partners. A possible way to do this in a decentralized manner is by requiring joining computers to possess a class 2 personal digital certificate that has never been seen by the server before. Such certificates can currently be purchased from companies like GlobalSign (`www.globalsign.net`) for 16 Euros.

When disk-space wasting is used, this scheme provides better backup service availability than the prepayment one because it limits backup service only immediately after a restoration. It may, however, require the user to pay for membership somehow and requires much more effort from the central server per system member.

3.2.3 Bandwidth theft

Another free-rider attack involves attackers using participating computers to broadcast information. For example, a cracker might wish to make his pirated-software collection available to his friends but lacks the bandwidth to do this from his home PC. He can join our system and place the data to be broadcast on each of his partners in the clear; he then gives out his partners' IP addresses via email to his friends, who then download the data from the partners, subjecting them to unfair high traffic.

This attack is easily thwarted, however, by imposing a quota on how many reads or writes a partner can do per day, say three times the number needed for daily backup. A somewhat larger quota may be needed during restorations.

3.3 Disrupter attacks

Another kind of attack that we must guard against is *disrupter* attacks. A disrupter is an attacker motivated to disrupt, impair, or destroy a system or particular user. They might want to do this for prestige (any system whose disruption would make the front page is a target for some attackers) or to hurt a hated company or person.

1. $\mathcal{A}$ chooses a random number $r1$
2. $\mathcal{A}$ sends *Commit*($\mathcal{A}, r1$) to $\mathcal{B}$
3. $\mathcal{B}$ chooses a random number $r2$
4. $\mathcal{B}$ sends $r2$ to $\mathcal{A}$
5. $\mathcal{A}$ sends opening information to $\mathcal{B}$
6. $\mathcal{B}$ sends *block*[$r1 \oplus r2$] to $\mathcal{A}$

Figure 4: The random-read protocol where $\mathcal{A}$ is reading a block from $\mathcal{B}$. *Commit*(-) generates a nonmalleable commitment that is revealed in step 5; it hides $r1$ from $\mathcal{B}$ until step 5 while preventing $\mathcal{A}$ from changing her mind after seeing $r2$.

Unlike free riders, disrupters cannot be deterred simply by making such attacks unprofitable; the attacks must be made ineffective or beyond the attackers' budget.

3.3.1 Blocking restoration systemwide

The basic disrupter attack against our scheme is to attempt to block restoration of as many machines as possible by controlling enough of their partners: any machine which has more than m attacker machines as partners will believe its backups succeed, but come restoration time will find that it is unable to recover its data because the attacker machines refuse to cooperate.

In order to attack a large number of machines (presumedly for publicity) this way, the attacker will have to swap disk space with thousands of machines. The disk storage to do this directly, however, is beyond the budget of any likely attacker. However, by using a man-in-the-middle attack, a smart attacker can appear to be storing data for thousands of machines without using any local disk space. He simply steps between pairs of partners (see Section 2.1) and passes blocks back and forth, leaving each original partner to believe the attacker is backing up its data, when in fact they are still storing each other's data. If the attacker encrypts data going one way and decrypts data going the other way, he can ensure that when he bows out the stored data will be useless to the original partners.

We can prevent such man-in-the-middle attacks by changing our block-access protocols slightly. Intuitively, the idea is to provide only the following read operation: read a completely-random block. Unlike the normal read-specified-block operation, this operation cannot be passed through: if $\mathcal{A}$ does a random read (say block r is chosen) of $\mathcal{B}$, there is no way for $\mathcal{B}$ to read block r from $\mathcal{C}$ efficiently because each random read he does has only a $1/s$ chance of returning block r. On average $\mathcal{B}$ must read $s/2$ blocks to fulfill each random-read request of $\mathcal{A}$. This means that a small number of challenges by $\mathcal{A}$ will quickly force $\mathcal{B}$ to exhaust his read quota with $\mathcal{C}$ (see Section 3.2.3), leaving him unable to answer all of the challenges.

This restriction, of course, must be lifted while a restoration is in progress and may be lifted systemwide periodically so that cleaners can run efficiently. Figure 4 shows one way to formalize the random-read operation into a protocol. Steps 1–5 here, modulo the inclusion of $\mathcal{A}$'s name in the commitment, are a standard variant of the flipping-coins-by-telephone protocol [2]; step 6 returns the resulting chosen block, the one at offset $r1$ xor $r2$.

We include $\mathcal{A}$'s name—either $\mathcal{A}$'s current IP address, or if that is spoofable, a public key—in the commitment in step 2 to prevent the random-number-choosing protocol itself from being passed through: If $\mathcal{B}$ simply passes the commitment along to $\mathcal{C}$, he will be caught after he reveals the commitment in step 5 and $\mathcal{C}$ sees that it does not have $\mathcal{B}$'s name in it. He is prevented from fixing up the commitment to have his name while still containing $r1$ by the non-malleability of the commitment scheme [9]. Because he is unable to pass through the choice of $r1$, he is unable to trick $\mathcal{A}$ and $\mathcal{C}$ into agreeing on the same random number.

3.3.2 Blocking one machine's restoration

The preceding defense prevents attackers from disrupting a large portion of our system; it does nothing, however, to prevent attackers from targeting one or two hated machines. It is not clear that much can be done to defend against such a focused attack. By requiring that all a computer's partners be located on different IP subnets, forcing partners to be chosen randomly from the eligible subset, and charging a fee for (excessive) switching the cost of such an attack could be raised somewhat, but probably not enough to deter determined attackers.

The attack is unlikely to be used, however, because it is not very effective at damaging the target machine compared to alternative attacks such as targeted viruses and denial-of-service attacks: it only blocks restoration; actual data loss requires an independent event that might take years to happen, if it happens at all. There is little point in combining this attack with a method for destroying a computer's data because it is usually easier to modify such methods to destroy any backup directly by corrupting the backup software.

4 The Prototype

We are building a prototype using our scheme, but it is not yet fully operational. Enough functionality is working, however, to allow measurement of the prototype's backup performance, which we report on here along with some interesting aspects of the prototype's data-layout choices.

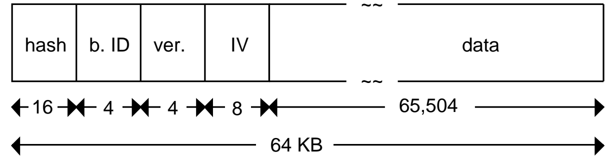

Figure 5: The format for blocks stored at partners, which includes a 16-byte HMAC-MD5 cryptographic hash, a block ID, a version number, an initialization vector (encryption-added random padding), and 65,504 bytes of backup data.

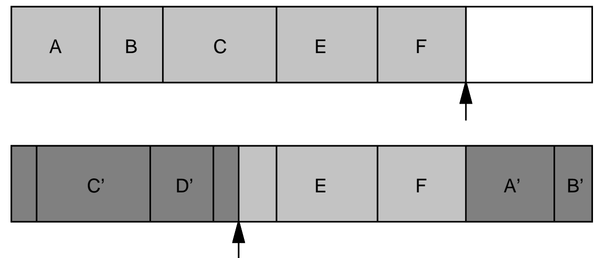

Figure 6: Sample partial overwrite of old snapshot (light grey) by new one (dark grey). Files A, B, and C are overwritten with new versions (A', B', C') and a new file D' is written. Only part of the new E is written, but it and F have old versions that are not yet overwritten.

4.1 Data layout

The prototype divides the logical disk into 64 KB blocks. Figure 5 shows the format used for these blocks. The necessary header fields use 32 bytes, leaving 65,504 bytes free for data, an overhead of only 0.05%. The prototype uses the IDEA block cipher for encrypting the data and the cryptographic-hash HMAC MD5 to generate checksums in the order described in Section 3.1.

The prototype treats the logical disk as if it were a large circular tape: each snapshot is written starting just after the last one, using ascending block offsets (with wrap around at the end of the disk). Snapshots use a format similar to archival file formats (*e.g.*, tar), which store a sequence of files by writing, for each file, a file header followed by that file's data. The file header contains a synchronizing sequence, the file name, date stamp, file length, and a checksum. Because of the synchronizing sequence, it is possible to start reading a snapshot in the middle and still extract all the files whose headers come after that point.

This property of the snapshot format can be useful when backup space is limited, and as a result the next snapshot necessarily overwrites the last full snapshot: Should the computer crash while writing the new snapshot, it will be left with two incomplete snapshots at restoration time, the beginning of the new one and the

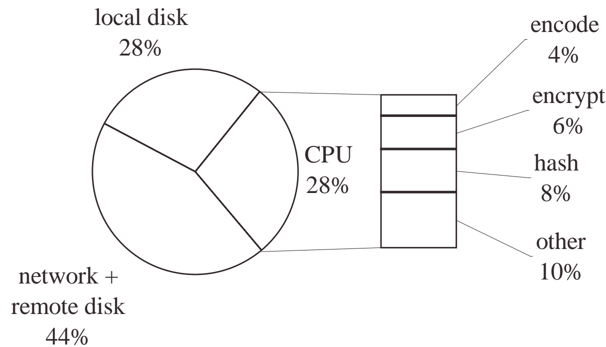

Figure 7: Breakdown of backup time.

end of the old one. The start-in-the-middle property allows reading and recovering all the complete files in both partial snapshots. If there is extra space available beyond that needed to hold a full snapshot and the set of files being backed up has not changed greatly, then most files should be recoverable, although some of them may be restored to the version saved in the old snapshot; see Figure 6 for an example.

4.2 Performance

To measure the prototype's backup performance, we used five personal computers running Microsoft Windows NT, each with a 200 Mhz Pentium Pro processor, 64 MB of RAM, and a 10 Mbps Ethernet card. The computers were connected via a 10 Mbps Ethernet hub. We used the (8,2)-Reed-Solomon erasure-correcting code, which tolerates the failure of any 2 of 8 partners at the cost of using $2/6 = 33\%$ extra space, and simulated 10 participating computers by running two instances of the prototype software on each PC. Each instance was partnered with the eight instances located on different PCs from it, so that all communication between partners went through the network.

We instructed one partner to write either 100 MB or 1 GB of test data stored on its local disk to its logical disk to simulate saving a snapshot; during this time, the other partners were idle except for processing write requests. The prototype uses the procedure described in Section 2.3 with w=7 to write snapshots: writes to partners occur in parallel, but are not pipelined with the reading and preparing of the blocks to be written, and at most one write is in progress at each partner at a time.

Using this unoptimised procedure, the prototype is able to write 100 MB in 12 minutes and 1 GB in 2 hours. This corresponds to a backup rate of 1.0 Mbps and a write rate of 1.3 Mbps (larger because of the redundancy blocks that must be written). Optimization would improve these rates, but since they are already larger than the bandwidth many Internet connections provide, it is not clear how useful this would be.

space	100 MB	1 GB	10 GB	100 GB
mean	$16.46	$53.84	$232.28	$1773.33
min	$4.50	$12.71	$72.21	$720.00

Figure 8: Monthly fees in US dollars required by existing Internet Backup services to store a given amount of data; excludes one-time startup fees but includes discounts for annual contract.

Figure 7 breaks down the contributions to the backup time made by the local disk (mostly reads), the remote-writing step (mostly network and partner delays), and the various CPU-intensive tasks. The remote-writing step consumes the largest portion (44%) of the total time, presumedly due to our allowing only one outstanding write to a given partner at a time. Hashing requires more time (8%) than encryption (6%) because it must be done twice for each block: once to generate the stored-block checksum and once to authenticate the write request containing that block to the partner. (Separate keys, and hence hashes, are required because the integrity key must be kept secret from the partner.)

5 Comparison with existing services

5.1 Cost

We did a survey of Internet backup sites on October 28, 2002. Figure 8 shows the average and minimum monthly fees for various amounts of storage for the 15 sites (out of 28 surveyed) that list prices on the web for given amounts of data to be stored (as opposed to the amount to be backed up, which differs due to assumptions about compression and how much data actually needs to be backed up). The cheapest marginal cost found was US $7.20 per gigabyte.

In estimating the cost of using our scheme, we assume that the user's computer hardware, base power, and any needed bandwidth are already paid for by existing uses. This seems reasonable for a home PC that is already on enough for reasonable predictable uptime and that has an Internet connection with flat-rate bandwidth pricing. We furthermore assume no cost for the software or central-server operation based on an open-source model and the extremely low overhead on the central server per participating computer; if a commercial model of software development was used instead, a small one-time fee for the software might be also be required. Under these assumptions, the cost of our scheme is determined by the marginal cost of storage for a PC and the marginal cost of the extra power needed to operate a disk drive to answer challenges and backup data.

Based on the cost of a 60 GB internal IDE hard drive as of October 2002—US $75 according to `www.pricewatch.com`—depreciated over 2 years, we conservatively estimate the marginal cost of storage

at no more than 5.2 US cents per gigabyte per month. The cost of the extra power required is much harder to estimate. One conservative approach is to assume that the disk is turned on 1.5 extra hours per day per gigabyte to be backed up, on the assumption that the average backup takes 25% of the time of a full one (2 hours/GB for the prototype in each direction) and that challenges take half an hour of disk time total per day. Desktop disk drives appear to consume about 10 watts extra power when active [11] so at a conservative electricity cost of 15 cents per kilowatt hour, our scheme should use 7.5 US cents of power per GB backed up per month.

If we assume 100% storage overhead for redundancy (*e.g.*, $k{=}m$) and room for 2 full snapshots on the logical disk (a factor of 2.2 should suffice), we will need to trade 4.4 GB of local disk in order to back up 1 GB of data, resulting in our scheme costing no more than 26 US cents per gigabyte per month. A less conservative estimate (3 years, 50% overhead) gives a figure of 18.6 US cents/GB/month.

Thus, our scheme appears to be 30 to 100 times cheaper than existing Internet backup services. We do not fully understand why this is, but believe it largely stems from our scheme's lack of administrative costs and use of marginal resources (*e.g.*, most of the resources we use are already paid for by other uses). A small part of the difference may be due to limitations of our scheme as compared to existing Internet backup services (see below).

Traditional backup methods can be comparable in cost to our scheme, but are inconvenient for users: if a home-computer user weekly writes a snapshot to 700 MB CD-Rs (6 US cents each in quantities of 100) then takes them to work and leaves them there, he will incur a cost of 35.1 US cents/GB/month ignoring the cost of the CD burner.

5.2 Limitations

Our scheme does have some limitations as compared to existing Internet backup services. Perhaps the biggest limitations are the limited grace period (2 weeks) and the need for sizable amounts of predictable Internet connectivity. Unlike the existing services, which provide long grace periods (months if not years) and only require a computer to be connected to the Internet when a snapshot needs to be saved, our scheme leaves computers with no backup at all in case of excessive downtime. Unlike schemes based on off-line media, our scheme offers little protection against catastrophic viruses that suddenly erase most PCs' hard drives.

Also, our scheme is somewhat less convenient than the best Internet backup services: we provide no tech support line to hold users hands and will not Fed Ex a CD copy of a user's data to them so they don't have to wait for a restore over a slow Internet connection. Depending on the uptime-level agreement, restoration may take longer in our scheme than with the existing services. In theory, Internet backup sites could offer insurance ("if we lose your data, we'll pay you a million dollars"). Insuring users in our system seems problematic due to possible fraud using the disrupter attack of Section 3.3.2.

If disk-space wasting is used, there are the additional limitations that backup service will not be available for 2 weeks after a crash, and if prepayment is used, backup service will not be available for the first 3 weeks and growing backup-storage space will take 3 weeks.

5.3 A hybrid model

It is possible to remove most of these limitations by combining our scheme with the existing Internet-backup-service model: A company would run the central server and bill users periodically. Backup would be done as per our scheme except that before abandoning a partner, a computer would upload the partner's data to the central site, where it would be stored temporarily. When the abandoned computer found a new partner, it would move its data from the central server to the new partner.

This model allows preserving backups even in the presence of excessive downtime. Members would be billed at a basic rate similar to our scheme but with surcharges for centrally storing data (charged presumedly at the existing Internet-backup-services rate) and for abusing partners. In essence, customers with good uptime would pay our cheap rates while still being guaranteed a backup even if they exceed the 2 week grace period or are down excessively, albeit at a slightly higher price.

6 Related work

The first wave of peer-to-peer systems included several systems (The Eternity Service [1], Archival Intermemory [4], Free Net [5], and Free Haven [7]) designed to provide *uncensorable storage*: documents once deposited in these systems cannot be altered or destroyed by attackers, even national governments. Like ours, these systems use cryptography and redundancy (Archival Intermemory uses erasure-correcting codes) to protect data. It was soon suggested that one of these systems would make a good base on which to build a backup system.

Unfortunately, however, these systems do not have working defenses against free riders when used in this way: Freenet keeps only the most popularly requested documents, Archival Intermemory simply assumes cooperation (it is intended for research libraries, which are believed to have substantially similar interests), and the Eternity Service paper only suggests a possible line of direction for a defense.

The Free Haven project has proposed an elaborate scheme based on trading storage, where nodes are allowed to insert only as many *shares* (the unit of trade in Free Haven) as they hold for others. However, unlike in our system, these shares are constantly traded between nodes, leading to effectively non-symmetric partnerships. This prevents a node $\mathcal{A}$ from directly punishing a node $\mathcal{B}$ that drops one of $\mathcal{A}$'s shares by dropping one of $\mathcal{B}$'s shares that $\mathcal{A}$ holds in return. Instead an elaborate reputation system is needed to punish nodes that are complained against too many times because they drop data. Unfortunately, building a working reputation system is very hard and it is far from clear if their system works [8].

More recent peer-to-peer storage systems include PAST [10, 14] and OceanStore [12, 15]. These systems do not attempt to provide uncensorability, and are thus simpler than the previous systems.

PAST relies on trusted third parties and smartcards to broker requests between clients so that clients cannot use more remote storage than they are providing locally. Unfortunately, insufficient details are provided about PAST's defenses to properly evaluate PAST's resistance to free-rider attacks—*e.g.*, nodes are supposed to be randomly audited, but no details of how or with what consequences are provided. PAST appears to encrypt data before replicating, which may allow a free-rider attack where malicious nodes obtain data from the redundant data that their peers store, rather than storing it themselves (see Section 3.1).

OceanStore is a federated system where utility companies pool their resources to provide storage to users. Each user contracts with a single company, the *responsible party*, to receive storage for a fee. That company then exchanges storage with the other companies for greater reliability and geographic range. Because of the companies' large size and deep pockets, legal contracts and enforcement can be used to punish companies that do not keep their end of the bargain, based on planned billing and auditing systems. OceanStore, because of its need to support concurrent updates, is very complicated (*e.g.*, it uses Byzantine agreement) and requires a great deal of central resources, making it likely more expensive than our scheme if only simple backup service is needed.

The only other peer-to-peer system we know of whose primary purpose is backup service is Pastiche [6]. Like in our scheme, Pastiche nodes form partnerships with other nodes. However, rather than using erasure-correcting codes, each Pastiche node stores a copy of all of its data on each of its partners. By sacrificing a fair amount of privacy (observers can tell if a node's filesystem includes a given large byte sequence), this backup data can be greatly compressed: by choosing partners with similar software installations, most files being backed up will be already present on the partner, thus requiring no extra storage. No data is available yet about whether this savings compensates for the greater overhead of using full replication rather than erasure-correcting codes in practice. Pastiche has not yet adopted defenses against free-rider attacks; the authors merely sketch, in a paragraph each, three approaches that they are considering.

7 Conclusions

In this paper, we have described a scheme for a peer-to-peer Internet backup system that appears to be one to two orders of magnitude cheaper than existing Internet backup services. We believe the cost savings stem largely from savings on administrative expenses and the use of cheaper resources, much of whose operating cost is paid for by other uses. Preliminary experiments with a prototype show that the scheme's performance is acceptable in practice.

The most difficult part of the scheme design was guarding against the inevitable free rider and disrupter attacks to which a cooperative system is vulnerable. We came up with several novel mechanisms—periodic random-block challenges, disk-space wasting, and limiting reads to mutually-chosen random blocks—to address these attacks.

Acknowledgments

We wish to thank Marcos Aguilera, the anonymous referees, and our shepherd, Steve Gribble, for their valuable comments and suggestions.

Notes

1. Increased convenience encourages more frequent off-site backups.

2. This assumes all blocks (of a block stripe) are written as a unit; we do not discuss the more complicated partial-write case in this paper.

References

[1] R. Anderson. The eternity service. In *Proceedings of Pragocrypt '96*, pages 242–252, Prague, Czech Republic, 1996. CTU Publishing House.

[2] Manuel Blum. Coin flipping by telephone: A protocol for solving impossible problems. In *Advances in Cryptology: A Report on CRYPTO 81*, pages 11–15. Department of Electrical and Computer Engineering, U. C. Santa Barbara, August 1982. ECE Report 82-04.

[3] William J. Bolosky, John R. Douceur, David Ely, and Marvin 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 2000)*, pages 34–43, Santa Clara, June 2000.

[4] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the fourth ACM Conference on Digital libraries (DL '99)*, 1999.

[5] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability, LNCS 2009*, New York, 2001. Springer.

[6] L. P. Cox and B. D. Noble. Pastiche: making backup cheap and easy. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[7] Roger Dingledine. The free haven project: Design and deployment of an anonymous secure data haven. Master's thesis, MIT, June 2000.

[8] Roger Dingledine, Nick Mathewson, and Paul Syverson. Reputation in privacy enhancing technologies. In *Proceedings of Computers, Freedom, and Privacy*, April 2002.

[9] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.

[10] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of HotOS VIII*, May 2001.

[11] Keith Farkas. Personal communication, March 2003.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Nov. 2000.

[13] James S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software, Practice and Experience*, 27(9):995–1012, 1997.

[14] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of 18th ACM SOSP'01*, October 2001.

[15] Chris Wells. The OceanStore archive: Goals, structure, and self-repair. Master's thesis, U.C. Berkeley, May 2000.