pStore: A Secure Peer-to-Peer Backup System

Abstract

In an effort to combine research in peer-to-peer systems with techniques for incremental backup systems, we propose pStore: a secure distributed backup system based on an adaptive peer-to-peer network. pStore exploits unused personal hard drive space attached to the Internet to provide the distributed redundancy needed for reliable and effective data backup. Experiments on a 30 node network show that 95% of the files in a 13 MB dataset can be retrieved even when 7 of the nodes have failed. On top of this reliability, pStore includes support for file encryption, replication, versioning, and sharing. Its custom versioning system permits arbitrary version retrieval similar to CVS. pStore provides this functionality at less than 10% of the network bandwidth and requires 85% less storage capacity than simpler local tape backup schemes for a representative workload.

1 Introduction

Current backup systems for personal and small-office computer users usually rely on secondary on-site storage of their data. Although these on-site backups provide data redundancy, they are vulnerable to localized catastrophe. More sophisticated off-site backups are possible but are usually expensive, difficult to manage, and are still a centralized form of redundancy. Independent from backup systems, current peer-to-peer systems focus on file-sharing, distributed archiving, distributed file systems, and anonymous publishing. Motivated by the strengths and weaknesses of peer-to-peer systems, as well as the specific desires of users needing to backup personal data, we propose pStore: a secure peer-to-peer backup system.

pStore provides a user with the ability to securely backup files in, and restore files from, a distributed network of untrusted peers. Insert, update, retrieve, and delete commands may be invoked by various user interfaces (e.g., a command line, file system, or GUI) according to a user's needs. pStore maintains snapshots for each file allowing a user to restore any snapshot at a later date. This low-level versioning primitive permits several usage models. For example, works in progress may be backed up hourly so that a user can revert to a last-known-good copy, or an entire directory tree can be stored to recover from a disk crash.

pStore has three primary design goals: reliability, security, and resource efficiency. pStore provides reliability through replication; copies are available on several servers in case some of these servers are malicious or unavailable. Since a client's data is replicated on nodes beyond his control, pStore strives to provide reasonable security: private data is readable only by its owner; data can be remotely deleted only by its owner; and any unwanted changes to data can be easily detected. Finally, since backups can be frequent and large, pStore aims to reduce resource-usage by sharing stored data and exchanging data only when necessary.

Section 2 discusses related systems. pStore draws from their strengths while discarding functionality which creates overhead and complexity in the application-specific domain of data backup. Section 3 outlines the pStore architecture, and Section 4 presents our implementation. Section 5 evaluates the design in terms of the goals stated above, and Section 6 concludes.

2 Related Work

A distributed backup system has two major components: a distributed storage system and a backup/versioning framework. While much work has been done in the two fields individually, there is little literature integrating the two.

2.1 Distributed Storage Systems

There has been a wealth of recent work on distributed storage systems. Peer-to-peer file sharing systems, such as Napster [15] and Gnutella [12], are in wide use and provide a mechanism for file search and retrieval among a large group of users. Napster handles searches through a centralized index server, while Gnutella uses broadcast queries. Both systems focus more on information retrieval than on publishing.

Freenet provides anonymous publication and retrieval of data in an adaptive peer-to-peer network [5]. Anonymity is provided through several means including: encrypted search keys, data caching along lookup paths, source-node spoofing, and probabilistic time-to-live values. Freenet deletes data which is infrequently accessed to make room for more recent insertions.

Eternity proposes redundancy and information dispersal (secret sharing) to replicate data, and adds anonymity mechanisms to prevent selective denial of service attacks [1]. Document queries are broadcast, and delivery is achieved through anonymous remailers. Free Haven, Publius and Mojo Nation also use secret sharing to achieve reliability and author anonymity [9, 22, 13].

SFSRO is a content distribution system providing secure and authenticated access to read-only data via a replicated database [11]. Like SFSRO, CFS aims to achieve high performance and redundancy, without compromising on integrity in a read-only file system [8]. Unlike complete database replication in SFSRO, CFS inserts file system blocks into a distributed storage system and uses Chord as a distributed lookup mechanism [7]. The PAST system takes a similar layered approach, but uses Pastry as its distributed lookup mechanism [10]. Lookups using Chord and Pastry scale as $O(log(n))$ with the number of nodes in the system. Farsite is similar to CFS in that it provides a distributed filesystem among cooperative peers [2], but uses digital signatures to allow delete operations on the file data.

Several systems have proposed schemes to enforce storage quotas over a distributed storage system. Mojo Nation relies on a trusted third party to increase a user's quota when he contributes storage, network, and/or CPU resources to the system. The PAST system suggests that the same smart cards used for authentication could be used to maintain storage quotas [10]. The Tangler system proposes an interesting quota scheme based on peer monitoring: nodes monitor their peers and report badly behaving nodes to others [21].

2.2 Versioning and backup

The existing distributed storage systems discussed above are intended for sharing, archiving, or providing a distributed file system. As a result, the systems do not provide specific support for incremental updates and/or versioning. Since many file changes are incremental (e.g., evolution of source code, documents, and even some aspects of binary files [14]), there has been a significant amount of work on exploiting these similarities to save bandwidth and storage space.

The Concurrent Versioning System, popular among software development teams, combines the current state of a text file and a set of commands necessary to incrementally revert that file to its original state [6]. Network Appliances incorporates the WAFL file system in its network-attached-storage devices [3]. WAFL provides transparent snapshots of a file system at selected instances, allowing the file system data to be viewed either in its current state, or as it was at some selected instance in the past.

Overlap between file versions allows reduction of network traffic required to update older versions of files. Rsync is an algorithm for updating files on one node so that they are identical to those on another node [20]. Its rolling, two-level checksum scheme is attractive as the use of checksums eliminates the need for a byte-by-byte file comparison and allows binary files to be compared. LBFS uses content-dependent chunk boundaries to achieve similar goals and additional benefits in the context of an entire file system [14].

3 System Architecture

Before discussing the details of the pStore architecture, we present an overview of how the system works for one possible implementation. A pStore user first invokes a pStore client which helps him generate keys and mark files for

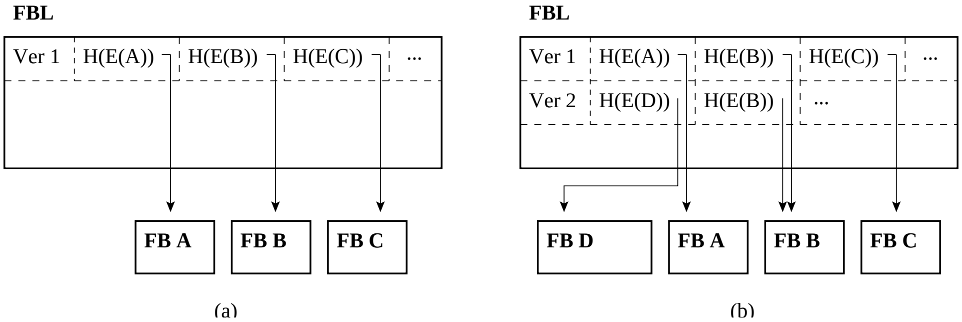

Figure 1: File Block List and File Blocks: (a) shows a file with three equal sized blocks, (b) shows how a new version can be added by updating the file block list and adding a single new file block.

backup. The user notes which files are most important to him, and the client uses his choices to decide how often to backup the file and how many replicas to make.

To insert a file, pStore computes an identifier specific to the user's file. The identifier is chosen in such a way that it will not conflict with identically named files owned by other users. The file is encrypted using symmetric encryption to prevent members of the pStore network from seeing its potentially private contents. The file is broken into digitally signed blocks, and signed metadata is assembled which indicates how the blocks can be reassembled. The metadata and blocks are inserted into a peer-to-peer network. If the file changes and is backed up again, only the changes to the file are stored.

To retrieve the file, the user specifies its name and version, or browses the pStore directory hierarchy. The metadata is retrieved, and it indicates where to look for blocks belonging to the desired version. When the file blocks are retrieved, their signatures are examined to ensure file integrity.

3.1 Data Structures

This section describes the data structures used to manage files, directories, and versions. These data structures were designed for reliability, security, and resource efficiency.

3.1.1 File Blocks Lists and File Blocks

A pStore file is represented by a *file block list* (FBL) and several *file blocks* (FB). Each FB contains a portion of the file data, while the FBL contains an ordered list of all the FBs in the pStore file. The FBL has four pieces of information for each FB: a file block identifier used to uniquely identify each FB, a content hash of the unencrypted FB, the length of the FB in bytes, and the FB data's offset in the original file. Figure 1a illustrates the relationship between a FBL and its FBs. A traditional file can be converted into a pStore file by simply breaking the file into several fixed size FBs and creating an appropriate FBL. File attributes such as permissions and date of creation can also be stored in the FBL.

The FBL contains version information in the form of an additional ordered list of FBs for each file version. An FB list for a new version is created with an adaptation of the rsync algorithm [20]. The revised algorithm uses the FB hash, length, and offset information to make an efficient comparison without the need to retrieve the actual FBs. Our adaptation extends rsync to support varying FB lengths that will arise from the addition of new FBs as the pStore file evolves; care is taken to match the largest possible portion of the previous version. If overlapping FBs are detected, then duplicates are not created. Instead, they are simply referenced in the FBL. This saves both storage space and network traffic (which can be particularly advantageous when using metered or low bandwidth network connections). New or changed portions of the file are broken into new FBs of appropriate size, referenced in the FBL, and inserted into the network. The final result, depicted in Figure 1b, is a pStore file which can be used to reconstruct either version with no duplicate FBs. Another advantage of the pStore versioning scheme is that corruption of an FB does not necessarily preclude the retrieval of *all* versions as it would if version information were unified with the data (as in CVS). Versions that do not include the corrupt FB will remain intact.

pStore also allows files to be grouped into directories. A pStore directory is simply a text file listing the name of each file and subdirectory contained in the directory. Since a directory is represented as a pStore file, the same mechanism described above can be used for directory versioning.

File block lists and file blocks are encrypted with symmetric keys to preserve privacy. This is in contrast to peer-to-peer publishing networks where encryption is used to aid anonymity and deniability of content for node owners [5, 22]. File blocks are encrypted using convergent encryption [2]. Convergent encryption uses a hash of the unencrypted FB contents as the symmetric key when encrypting the FB. This makes block sharing between different users feasible since all users with the same FB will use the same key for encryption. Convergent encryption makes less sense for FBLs, since FBLs are not shared and the unencrypted content hash would have to be stored external to the FBL for decryption. FBLs are instead encrypted with a symmetric key derived from the user's private key. This means that all FBLs for a given user are encrypted with the same key, simplifying key management.

3.1.2 Data Chunks

For the purposes of pStore file insertion and retrieval in a distributed peer-to-peer network, FBLs and FBs are treated the same. Both can be viewed as a *data chunk*, denoted $C(i, p, s, d)$, where i is an identifier, p is public metadata, s is a digital signature, and d is the actual data. Any data chunk in the network can be retrieved by specifying its identifier. The public metadata is signed with the owner's private key and the owner's public key is included with each data chunk. This allows anyone to verify and view public metadata, but only the owner can change the public metadata.

As mentioned above, pStore uses a hash of the encrypted FB contents for the file block identifier. Thus, when the FB is retrieved, one can compare a rehash of the data chunk to the identifier to verify the data has not been tampered with. The public metadata for a FB chunk contains this identifier and is used for authentication when deleting FB chunks. A FB chunk can be specified more formally as follows:[1]

$$\begin{aligned}
i &= H(d \circ \mathrm{salt}) \\
p &= \mathrm{type} \circ i \\
s &= E^p_{K'_A}(H(p)) \circ K_A \\
d &= E^s_{H(FB)}(FB) \\
C_{FB} &= C(i, p, s, d)
\end{aligned}$$

The type is an indicator that this is a FB chunk as opposed to an FBL chunk. Type is included to ensure correct sharing and deletion of chunks. The identifier salt is used for replication and is discussed in Section 3.2.1.

A hash of the filename makes a poor FBL chunk identifier since it is likely multiple users will have similarly named files creating unwanted key collisions. Although two users have a file with the same name, they might have very different contents and should be kept separate to maintain consistency and privacy. A hash of the actual FBL also makes a poor chunk identifier since it will change after every version and cannot be easily recovered from the current local copy of the file. pStore uses a namespace-filename identifier which is similar to Freenet's signed-subspace keys [5]. Every pStore user has a private namespace into which all of that user's files are inserted and retrieved, eliminating unwanted FBL chunk identifier collisions between different users. A pStore user creates a private namespace by first creating a private/public key pair. pStore provides the flexibility for a user to have multiple namespaces or for several users to share the same namespace. A namespace-filename identifier is formed by first concatenating the private key, pStore pathname, filename, and salt. The results are then hashed to form the actual identifier. In addition to providing a unique namespace per user, this allows immediate location of known files without the need to traverse a directory structure. A FBL chunk

[1]We use the following notation for cryptographic primitives.

$H(M)$	: one-way hash of M
$E^t_K(M)$	: M encrypted with key K
$D^t_K(M)$	: M decrypted with key K
K_A	: public key belonging to A
K'_A	: private key corresponding to K_A

The superscript for encryption and decryption indicates whether a symmetric scheme ($t = s$) or public key scheme ($t = p$) is used.

can be specified more formally as follows:

$$\begin{aligned} i &= H(K'_A \circ \text{path} \circ \text{filename} \circ \text{salt}) \\ p &= \text{type} \circ \text{timestamp} \circ i \circ H(d) \\ s &= E^p_{K'_A}(H(p)) \circ K_A \\ d &= E^s_{f(K'_A)}(FBL) \\ C_{FBL} &= C(i, p, s, d) \end{aligned}$$

where f is a deterministic function used to derived the common symmetric key from the user's private key. A timestamp is included to prevent replay attacks. The type and salt are used in a similar manner as for FBs.

The public key contained within the public metadata provides ownership information useful when implementing secure chunk deletion (as described in Section 3.2.3). This ownership information may also be useful when verifying that a user is not exceeding a given quota. Unfortunately, attaching ownership information makes anonymity difficult in pStore. Since anonymity is not a primary goal of pStore, we feel this is an acceptable compromise.

The content hashes located in the public metadata provide a mechanism for verifying the integrity of any chunk. Since the hash is publicly visible (but immutable), anyone can hash the data contents of the chunk and match the result against the content hash in the public metadata. Unlike FBs, there is no direct relationship between the FBL identifier and the FBL contents. Thus an attacker might switch identifiers between two FBLs. Storing the identifier in the FBL public metadata and then comparing it to the requested search key prevents such substitution attacks.

3.2 Using the Data Structures

pStore relies on Chord to facilitate peer-to-peer storage due to its attractive $O(logN)$ guarantees concerning search times [7]. Unlike Freenet, Chord does not assume any specific data access pattern to achieve these search times, and this integrates nicely with pStore's primarily data insert usage model. Since it is a low-level primitive, using Chord does not burden us with extraneous functionality ill-suited for a distributed backup system such as absolute anonymity and file caching at intermediate nodes.

3.2.1 Replication

pStore supports exact-copy chunk replication to increase the reliability of a backup. Chunks are stored on several different peers, and if one peer fails then the chunk can be retrieved from any of the remaining peers. More sophisticated information dispersal techniques exist which decrease the total storage required while maintaining the reliability [18]. Although pStore uses exact-copy chunk replication to simplify the implementation, these techniques are certainly applicable and may be included in pStore in the future.

To distribute chunk replicas randomly through the identifier space, salt is added when creating the identifier. The salt is a predetermined sequence of numbers to simplify retrieval of replicas. This replication technique differs from "chain replication" in which the user sends data to one node, requesting that it store a copy of the data and pass it along to another node until a counter has expired [8, 5]. Malicious or broken nodes in the chain can reduce the effectiveness of chain replication by refusing to pass the data along, potentially preventing replicas from reaching benign nodes. pStore's replication technique avoids this problem by sending replicas directly to the target nodes.

Many systems rely on caching frequently accessed files along the retrieval path to further increase data replication [5, 8, 10], but a data backup application is poorly suited to this form of caching. File insertion is much more common than file retrieval for pStore and other data backup applications. Although FBLs are accessed on every backup, they are never shared. Instead of caching along the lookup path, FBLs can be cached on the owner's local machine. FBs are shared but are rarely accessed.

3.2.2 Sharing

Unlike file sharing peer-to-peer networks, a pStore user cannot directly make use of the local storage space he contributes to the system. To encourage fair and adequate donation of storage space, pStore would require a quota policy where the amount of space available to a user for pStore backups is proportional to the amount of personal

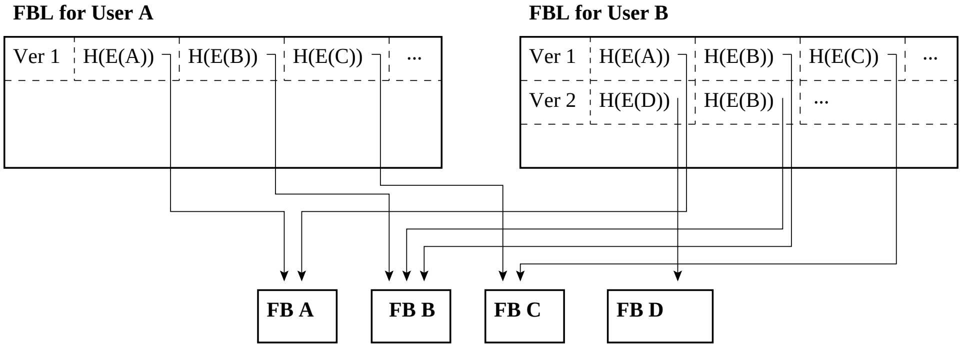

Figure 2: Sharing File Blocks: Blocks are shared between user A and user B and are also shared between two of different versions of the file for user B.

storage space contributed to the system. Users then have a vested interested in making sure their data is stored efficiently. Techniques which decrease the space required to store a given file in pStore increase the effective pStore capacity for the owner of that file. To address this issue pStore permits sharing at the file block level to capture redundancy both within a single file and between different versions, files, and users.[2] FB sharing occurs explicitly during versioning as described in Section 3.1.1. By exploiting the similarity between most versions, this technique saves storage space and reduces network traffic which could be significant when performing frequent backups. FB sharing also occurs implicitly between duplicate files owned by the same or different users.[3] Implicit FB sharing is a result of using a hash of the encrypted contents as the file block identifier. Convergent encryption ensures that identical file blocks are encrypted with the same key, allowing multiple users to securely share the same FBs. File sharing between users can be quite common when users backup an entire disk image since much of the disk image will contain common operating system and application files. A recent study showed that almost 50% of the used storage space on desktop computers in Microsoft's headquarters could be reclaimed if duplicate content was removed [2]. Figure 2 illustrates an example where two users have inserted an identical file. Notice how although user B has modified the file and inserted a new version, most of the FBs can still be shared.

3.2.3 Deletion

If we assume a policy where users can only insert an amount data proportional to the amount of storage contributed, then pStore users may reasonably demand an explicit delete operation. A user may want to limit the number of versions per FBL or remove files to free space for newer and more important files.

Explicit delete operations in peer-to-peer systems are rare, since there is the potential for misuse by malicious users. Systems such as Freenet, Chord, and Free Haven rely instead on removing infrequently accessed files or using file expiration dates [5, 7, 9]. Removing infrequently accessed files is an unacceptable solution for pStore, since by definition backups are rarely accessed until needed. Expiration dates are also less applicable to pStore, since a user may be unable to refresh his data due to a hardware crash - and this is exactly the case when a backup is needed.

Exceptions include Publius, which attaches an indicator to each file that only acceptable users can duplicate to indicate file deletion, and Farsite, which uses digital signatures to authorize deletions. pStore also uses digital signatures in the form of the public metadata mentioned in Section 3.1.2. This public metadata can be thought of an *ownership tag* which authorizes an owner to delete a chunk.

[2] Future work might investigate the benefit of other techniques such as data compression and information dispersal algorithms (as in [18, 4]) to further increase effective capacity for a user.

[3] We adopt the terminology presented in [2]: *replicas* refer to copies generated by pStore to enhance reliability, while *duplicates* refers to two logically distinct files with identical content.

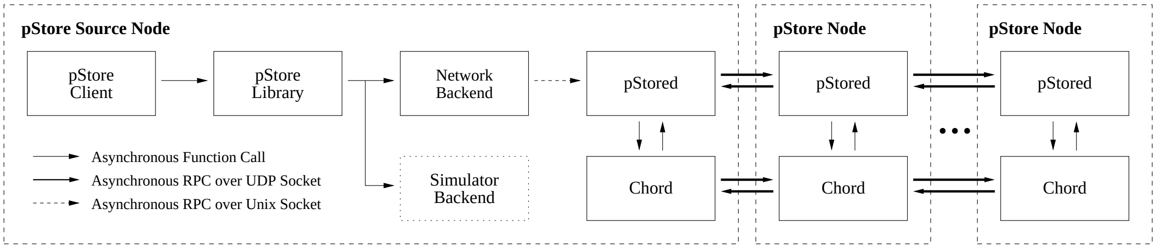

Figure 3: pStore Implementation

Each storage node keeps an ownership tag list (OTL) for each chunk. While FB chunks may have many ownership tags, each FBL chunk should only have one ownership tag. A user can make a request to delete a chunk by inserting a *delete chunk* into the network. The delete chunk has the same identifier as the chunk to delete, but has no data.

When a storage node receives a delete chunk, it examines each ownership tag in the OTL associated with the chunk. If the public keys match between one of the ownership tags and the delete chunk, then the delete is allowed to proceed. The appropriate ownership tag is removed from the OTL, and if there are no more ownership tags in the OTL then the chunk is removed. The OTL provides a very general form of reference counting to prevent deleting a chunk when other users (or other files owned by the same user) still reference the chunk.

Notice that a malicious user can only delete a chunk if there is an ownership tag associated with the malicious user on the OTL. Even if a malicious user could somehow append his ownership tag to the OTL, the most that user could do is remove his own ownership tag with a delete command chunk. The chunk will remain in the network as long as the OTL is not empty.

4 Implementation

We implemented pStore using both original C++ code and third-party libraries for encryption algorithms and asynchronous programming. pStore uses RSA for public key encryption [19], Rijndael (AES) for symmetric key encryption [17], and SHA-1 for cryptographic hashing [16]. Figure 3 shows the components involved in the pStore implementation. pStore clients are linked with the pStore library to provide various user interfaces. The current implementation provides a command line interface for inserting, updating, retrieving, and deleting single files, as well as public and private key generation. In addition to the Chord backend for peer-to-peer storage, a simulator backend was developed which stores chunks to the local disk for debugging and evaluation purposes. pStored runs on each node in the network and maintains a database of chunks which have been inserted at that node.

The current system acts as a proof-of-concept implementation. As such, certain features in the pStore architecture have been omitted. pStore directories are not implemented and no effort is made to handle very large files efficiently. Although quotas would be important in a production pStore system, this implementation does not provide a mechanism for quota enforcement.

5 Evaluation

Three realistic workloads were assembled to test various aspects of pStore and evaluate how well it meets its goals of reliability and minimal resource-usage. The first workload, *softdev*, models a software development team that backs up their project nightly. At any time during the development process they must be able to revert to an older version of a file. It consists of ten nightly snapshots of the development of pStore and includes a total of 1,457 files (13MB). The *fulldisk* workload models individual users who occasionally back up their entire hard disk in case of emergency. We chose four Windows 2000 machines and five Linux-based machines (four RedHat 7.1 and one Mandrake 8.1) and collected 5% of the files on each disk. Files were chosen with a pseudo-random number generator and the resulting workload includes a total of 23,959 files (696MB). The final workload, *homedir*, contains 102 files (13MB) taken from a user home directory. These workloads provide a diverse testset with different file types, file sizes, number of files, and amounts of versioning.

To test reliability, we brought up a network of 30 nodes across five hosts and backed up both *softdev* and *homedir*. Ordinarily, each pStore node would be on a separate host, but five hosts allowed for a manageable test environment and the logical separation between nodes allowed for a fair reliability study. Nodes were killed one-by-one, and we measure how many files could be retrieved from the remaining nodes as a percentage of the entire dataset.

The results in Figure 4 show backup availability as a function of failed nodes. r denotes number of instances of a file inserted into the network. As expected, increasing r increases the availability of the backup set. For *homedir* which consists of fewer, larger files than *softdev*, the reliability is usually less for each number of failed nodes. Even when seven nodes (23% of 30) have left the network, four replicas are sufficient to bring back 95% of one's files for both workloads. The observed reliability of the pStore system can be generalized as follows. Given a fraction, f, of failed nodes and a uniform assignment of chunks to nodes due to SHA-1 hashing, the probability that a block is not available is f^r. A file with k blocks is not retrievable if any of the k blocks are not available – a probability of $(1 - f^r)^k$. Thus, for a given r (which can be small), similar availability may be maintained as the network scales as long as f remains unchanged.

The remaining experiments, which do not rely on the properties of pStore's underlying network, were performed using the pStore simulator. The simulator allowed analysis to be performed prior to integration with Chord. Figures 5 and 6 show the efficiency of pStore when working with *softdev*. The figures compare four techniques for maintaining versioned backups: local tape backups (*tape*), pStore without versioning (*nover*), pStore with versioning (*withver*), and pStore using CVS for versioning (*cvs*). The *tape* test was simulated by using the `cp` command to copy files to a new directory each night. The *nover* test

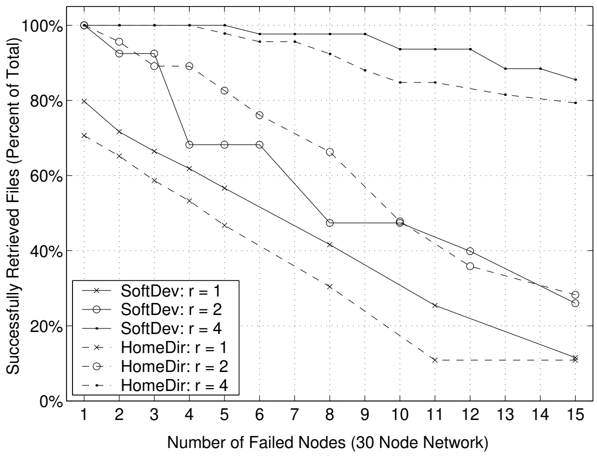

Figure 4: File Availability vs. Node Failures

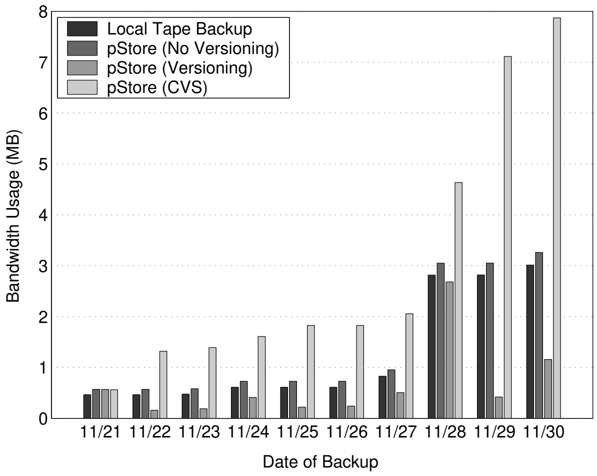

Figure 5: Bandwidth Usage - *softdev* Backup

does not use the versioning algorithm described in Section 3.1.1. Instead it simply inserts each additional version under a different pStore filename (creating a separate FBL for each version). The *withver* test uses the pStore versioning algorithm. To evaluate pStore's versioning method against an established versioning scheme, we used pStore to back up CVS repository files. These files can be retrieved and manipulated with the CVS program rather than relying on pStore versioning. In our tests, the entire CVS repository file is retrieved and then deleted from the network. Next, CVS is used to produce a new version, and the new repository file is inserted into the network.

Figure 5 shows the network bandwidth usage for each versioning technique when inserting each nightly snapshot. Notice that for the first snap-

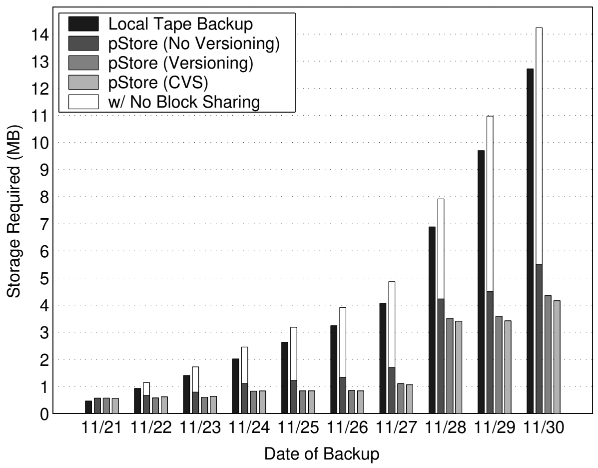

Figure 6: Required Storage for *softdev*

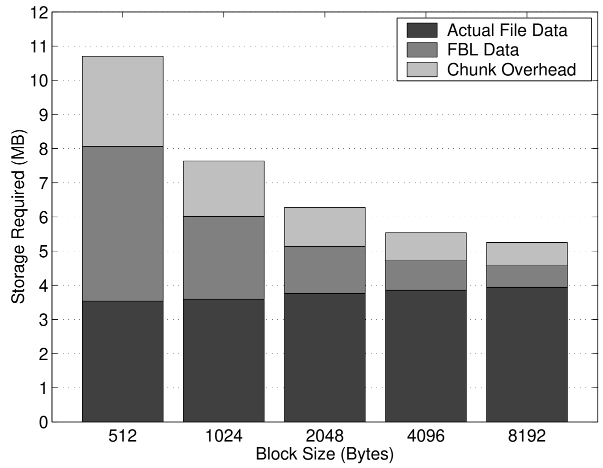

Figure 7: Storage vs. Block Size:

shot, pStore FBLs and chunk overhead (public metadata) cause the pStore tests to use more bandwidth than the *tape* test. For later versions, the *withver* test requires less than 10% the bandwidth of *tape* and *nover* tests since pStore versioning avoids inserting identical inter-version data into the network. The *cvs* test uses drastically more bandwidth since each new version requires retrieval of the entire CVS repository and consequently all of the data associated with each previous version.

The amount of bandwidth for restoring versions was also analyzed. The *tape*, *nover*, and *withver* tests were similar since each test must retrieve the entire version. The *cvs* test again used drastically more bandwidth since all versions must be retrieved at once and not just the the desired version.

Figure 6 shows how much storage is required in the network using each of the four methods. The stacked white bar indicates the additional amount of storage required if implicit block sharing is turned off. Without versioning or block sharing, pStore is nearly equivalent to the local tape backup with the exception of the storage required for FBLs and chunk overhead. Since subsequent versions in *softdev* are similar or identical, implicit block sharing can reduce the required storage by 3-60% even without any sophisticated versioning algorithms. The *withver* and *cvs* tests reap the benefit of block sharing as well. Since this is due to explicit block sharing during versioning, there is no increased storage requirement when implicit block sharing is turned off (as illustrated by the lack of visible white bars for these tests). With implicit block sharing enabled, the *withver* and *cvs* tests are able to reduce the storage required by another 1-10% over *nover*.

Clearly, the rsync algorithm on which pStore is based will find more overlap between file versions when it is comparing smaller blocks. Thus, decreasing block size appears to be a simple way to decrease the required storage of pStore versioning. In practice, while the storage required for the actual data decreases due to more effective versioning, the size of each FBL and the amount of public metadata increases due to the greater number of blocks. This increase in overhead outweighs the modest decrease in the actual stored data (see Figure 7). The 11/30 *softdev* snapshot is shown; other days exhibit similar trends.

It is reasonable to expect common blocks between users running the same operating system. To test this, the amount of block sharing was measured while inserting the *fulldisk* workload into pStore. Unique public/private key pairs where used for each user to test convergent encryption. The Linux datasets saved 3.6% of the total inserted bytes through block sharing, while the Windows datasets saved 6.9%. When both sets were inserted simultaneously, 5.3% was saved. This number is much lower than the 50% found in the Farsite study [2], probably owing to the homogenous nature of the Farsite workload compared to the heterogenous nature of the *fulldisk* workload.

6 Conclusion

pStore uses chunk replication, various cryptographic techniques, and a revised versioning algorithm to achieve its three primary design goals of reliability, security, and resource efficiency. We have described a proof-of-concept implementation of the pStore architecure which provides a command line interface for file insertion, update, retrieval, and delete in a network of untrusted peers. We show that a small number of replicas is sufficient to provide adequate reliability for a modest network size and that pStore can provide significant bandwidth and storage savings. These savings are useful in the context of a quota system where users are concerned about their effective capacity in the network and also in situations where users have disk or network bandwidth constraints. pStore is a novel system which brings together developments in peer-to-peer storage with those in the domain of data backup and versioning.

References

[1] R. Anderson. The eternity service. In *Proceedings of the 1st International Conference on the Theory and Applications of Cryptology*, 1996.

[2] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Measurement and Modeling of Computer Systems*, pages 34–43, 2000.

[3] K. Brown, J. Katcher, R. Walters, and A. Watson. SnapMirror and SnapRestore: Advances in snapshot technology. Technical report, TR3043, Network Applicance, 2001.

[4] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, Berkeley, CA, Jul 2000. International Computer Science Institute.

[6] Concurrent versions system. http://www.cvshome.org.

[7] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan. Building peer-to-peer systems with Chord, a distrubuted location service. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, pages 71–76, 2001.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[9] R. Dingledine, M. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, Berkeley, CA, Jul 2000. International Computer Science Institute.

[10] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, May 2001.

[11] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 181–196, Oct 2000.

[12] The gnutella protocol specification v0.4. Distributed Search Services, Sep 2000.

[13] Mojo nation. http://www.mojonation.net.

[14] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Oct 2001.

[15] Napster. http://www.napster.com.

[16] National Institute of Standards and Technology. Secure hash standard. Technical Report NIST FIPS PUB 180, U.S. Department of Commerce, May 1993.

[17] National Institute of Standards and Technology. Advanced encryption standard (AES). Technical Report NIST FIPS PUB 197, U.S. Department of Commerce, Nov 2001.

[18] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Jounral of the ACM*, 36(2):335–348, Apr 1989.

[19] R. Rivest, A. Shamir, and L. Adelman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb 1978.

[20] A. Tridgell and P. Macherras. The rsync algorithm. Technical report, TR-CS-96-05, Australian National University, Jun 1996.

[21] M. Waldman and D. Mazières. Tangler: A censorship resistant publishing system based on document entanglements. In *8th ACM Conference on Computer and Communcation Security*, Nov 2001.

[22] M. Waldman, A. Rubin, and L. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, Aug 2000.