pStore: A Secure Distributed Backup System

Ken Barr, Christopher Batten, Arvind Saraf, Stanley Trepetin
6.824 Distributed Operating Systems Project
{kbarr|cbatten|arvind_s|stanleyt}@mit.edu

Abstract

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. We propose pStore, a secure distributed backup system based on a peer-to-peer network. pStore exploits the growing amount of unused personal hard drive space attached to the Internet to provide the distributed redundancy needed for reliable and effective data backup. pStore includes support for file encryption, replication, versioning, and sharing.

1 Introduction

Current peer-to-peer systems focus on file-sharing, distributed archiving, distributed file systems, and anonymous publishing. Motivated by the strengths and weaknesses of such systems, as well as the specific desires of users needing to backup personal data, we propose pStore: a distributed personal backup system.

pStore provides a user the ability to securely backup files in and restore files from a distributed network of untrusted peers. Insert, lookup, retrieve, delete, and update commands may be invoked by several possible user interfaces (e. g. a command line, file system, or GUI) according to a user's needs. For example, works in progress may be backed up hourly so that a user can revert to a last-working-good copy; crucial files can be replicated aggressively; or an entire directory tree can be stored in case of a disk crash.

The design of pStore evolved to achieve several goals. The system aims to provide reliability through replication, making sure that copies are available on several servers in case some of these servers become 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 or replace only by its owner; and any changes to data can be detected. Finally, pStore aims to reduce resource-usage by sharing data when possible and retrieving only as much as necessary.

Section 2 dicusses the related systems. pStore draws from their strengths while discarding any functions that would appear as overhead and complexity in the application-specific domain of data backup. The pStore design is explained in detail in section 3 and pStore's layered implementation is discussed in Section 4. Section **??** describes experiments used to evaluate the design in terms of the goals stated above. Results of these experiments are presented in section **??**.

2 Related Work

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

2.1 Distributed Storage Systems

Network file systems, such as Sun NFS and AFS, were some of the earliest network storage systems. These systems, however, are intrinsically centralized. With the developement of the Internet, a large amount of of storage space became interconnected. Napster and Gnutella first harnessed this storage space via file sharing mechanisms. As web publishing became more rampant, issues such as anonymity and prevention of censorship ("preserving the Guttenberg inheritance" [1]) became more important. Freenet attempted to guarantee publisher anonymity through routing. Systems such as Free Haven, Publius and Mojo Nation inspired by Eternity used cryptography and information dispersal to address this problem. Later systems tried to use exploit load balancing, performance, and reliability through replication achieved on a distributed system to build file system on it.

Napster [14] Napster is an MP3 file sharing system. Users indicate files accessible by others, and they are added in a keyword based centralized index.

Gnutella [12] Gnutella also provides a file sharing service. Unlike Napster, Gnutella does not have a central server to do the indexing. Instead, each query is broadcasted to a set of nodes ("friends"), and each node is free to interpret it as it likes.

In both of above systems, once the data source is located, data is tranferred directly from the source to the requestor. This does not allow author or reader anonymity. To guarantee anonymity, the system must preclude any relation between data location and the author. It can be done by having dedicated servers as data storehouse, and the author uploading data onto them. In a system with a peer-to-peer process running on the source node, data must be exchanged among with peers. Anonymous publishing systems such as Freenet, Free Haven, Publius and Mojo Nation use these schemes for anonymity.

Freenet [5] Freenet provides adaptive peer-to-peer network storage of files permitting publication, replication, and retrieval of data while protecting the anonymity of both authors and readers. Each node caches the last few $< key, sourcenode >$ pairs it has seen for successful requests. Document insertions and requests are forwarded along the path corresponding to the nearest key value. Forwarded files are cached along the path from source to the requestor. Having intermediate nodes change source field for both document insertions and search requests makes it difficult to track the original author or readers.

Eternity [1] It uses redundancy and information dispersal (secret sharing) to replicate data, and adds anonymity mechanisms to prevent selective denial of service attacks. An author breaks the documents into fragments, uploads them along with the requested file duration and digital cash. Document queries are done via broadcast, and delivery achieved through anonymous retailers.

Like Eternity, Free Haven, Publius and Mojo Nation use secret sharing to achieve reliability and author anonymity. These systems differ though in identification and location of documents.

Free Haven [9] In Free Haven, an author generates a pulic key-private key pair, signs the document fragments using it, and upload them into the servers. These fragments are indexed by hash of the public key. Trading of these fragments adds to author anonymity.

Publius [18] Publius uploads the document fragments using anonymous channels. Fragment identifiers are combined to give a URL. Document retrieval is implemented by running a local web proxy on the client machine. This local proxy fetches each share independently, reconstructs the key, and then decrypts the document. By attaching ownership tags with each document, the system allows the original owner to update or delete these documents.

Mojo Nation [13] Mojo Nation uses a central server to locate document fragments. A novel aspect of this system is an accounting scheme that provides incentive for users to contribute resources to the system. Each peer-to-peer interacation in the system costs some credits. Credits are earned by by contributing storage, network, and CPU resources to the system. Corresponding bookkeeping and transactions are carried by a trusted third party.

Intermemory [4] Intermemory intends to provide highly available digital data, that could be used both for archiving as well as publishing. Like above schemes, it uses information dispersal algorithms to achieve reliability. In order to improve the common case performance however, it also stores the full data blocks in the network. All meta information regarding data location is stored at all nodes, and propagated during periodic database exchange and synchronization mechanism between neighbors.

PAST/Pastry [10] PAST names files using random strings tied to their contents. Replicas of files are placed at diverse set of nodes by a fault tolerant and self-organizing infrastructure called Pastry. A novel use of the system is with smart-cards. The smart card contains an ID and a unique public key-private key pair signed by the distributor. The smart card generates and verifies various certificates, in addition to possibly maintaining usage quota of each user.

SFSRO [11] SFSRO is a content distribution system providing secure and authenticated access to read only data. The publisher creates a signed database out of file system's contents. This signature is in the form of hierarchy of content hashes parallel to the file system hierarchy. The root block is signed by the source's private key. The signed database is replicated across multipe distribution servers. Desired documents are retrieved from nearby distribution server, and verified for the integrity using the above hash hierarchy.

Cooperative file system (CFS) [7] Like SFSRO, CFS aims to achieve high performance and redundancy, without compromising on integrity in a read only file system. Unlike simple database replication in SFSRO, CFS distributes file system blocks over distributed storage system (DHash over Chord [6]). File system format and integrity checks remain

the same as in SFSRO.

Farsite [2] Farsite guarantees highly available and reliable file system service using replication. Logically, a single hierarchical file system is visible from all access points. Underneath, encrypted and signed files are distributed across several network nodes. The digital signatures allow update or write operations on the file data.

2.2 Versioning and backup

As we saw above, most existing distributed systems are meant for publishing. Some are meant to be archival or distributed file systems. As a result, they do not provide good support for incremental updates and/or versioning.

Most file system changes are incremental. If different file versions are to be stored, sharing between versions can reduce disk storage (Network Appliance WAFL file system, Concurrent versioning system). Sharing can also be relied upon to reduce the network traffic required to update older versions of files in the network (Rsync).

Network Appliance WAFL file system [3] WAFL file system allows snapshoting of 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 instants in the past. It does so by having a block map file for the file system. The block map file contains the list of file blocks either in active use or belonging to one of the previous snapshots. At snapshot instant, the active block map list is copied onto the snapshot list. For any further changes made to the file blocks, a new block is allocated and the active file system block map made to include this.

Rsync [17] Rsync is an algorithm for updating files on one node to be identical to that on another node. The node having older copy of the file sends the checksums of file blocks to the source node. The source node compares these with the checksums of same sized blocks at different offsets in the more recent local file. Finally, it sends out a sequence of instructions for the reconstructing the copy using the older copy. For blocks that match, this instruction is a pointer to existing old file block. For others, it is literal data.

3 Approach

pStore makes use of three cryptographic primitives: public key encryption, symmetric key encryption, and one-way hashing. We chose RSA for public key encryption owing to its widespread acceptance and simplicity [16]. pStore uses the Rijndael algorithm for symmetric encryption due to its recent acceptance as the Advanced Encryption Standard (AES) and its relatively fast encryption and decryption time [8]. For one-way hashing, pStore uses SHA-1 which produces a fixed length hash from an arbitrary length input with an extremely low probability of collisions [16].

We use the following notation throughout the remainder of this paper. The superscript for encryption and decryption indicates whether symmetric key encryption/decryption ($t = s$) or public key encryption/decryption ($t = p$) is used.

$H(M)$	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

3.1 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 three pieces of information for each FB: a file block identifier (I_{FB}) used to uniquely identify each FB, a content hash of the unencrypted FB ($H(FB)$), and the length of the FB in bytes (l_{FB}). 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 also contains version information, by including an additional ordered list of FBs for each file version. For example, assume we have a traditional file f which we then convert into the pStore file f^p. We now modify f to create f' and would like to record these modifications as a new version in the pStore file f^p. This can be done by first retrieving the FBL associated with f^p, and using the $H(FB)$ and l_{FB} information to determine if any portions of f' match a FB in f^p (a process similar to that used in rsync will be used to perform this comparison [17]). If so, then there is no reason to create duplicate FBs, and instead they are just referenced appropriately in the FBL. New portions of f' are broken into new FBs and are also referenced appropriately in the FBL. The final result is the pStore file f^p which can be used to reconstruct either version

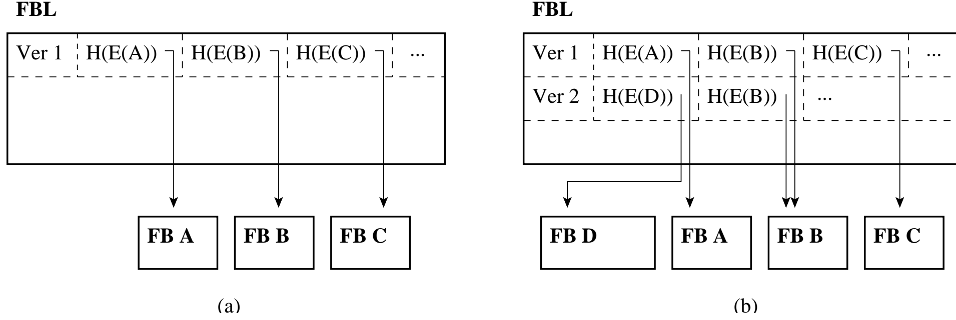

Figure 1: File Block List and File Blocks: (a) shows a file with three equal sized blocks, (b) shows how a new version can be added by updating the file block list and adding a single new file block.

with no duplicate FBs. Figure 1b illustrates how pStore handles file versioning.

pStore also allows pStore files to be grouped into virtual directories. A virtual directory is just a pStore text file listing the name of each file and subdirectory contained in the virtual directory. Since a virtual directory is a pStore file, the same mechanisms listed above can be used for virtual directory versioning.

3.2 Encryption

File block lists and file blocks are encrypted in pStore 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, 18]. File blocks are encrypted using convergent encryption, a technique used in the Farsite system [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 the AES symmetric key which is derived from the user's private key. This means that all FBLs are encrypted with the same key.

3.3 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*, where a data chunk has three parts: data, signed public metadata, and an identifier. The notation $C(i,p,d)$ is used to express a chunk with identifier i, public metadata p, and data d. Any data chunk in the network can be retrieved with the appropriate identifier. The public metadata is signed with the owner's private key and the owner's public key is included with the public metadata. This allows anyone to verify and view public metadata, but only the owner is able to change public metadata.

As mentioned in the previous section, pStore uses a hash of the encrypted FB contents for the file block identifier. The public metadata for a FB chunk contains just this identifier and is used for authentication when deleting FB chunks. A FB chunk can be specified more formally as follows:

$$\begin{aligned} i &= H(d + \text{salt}) \\ p &= E^{p}_{K'_A}(\text{type} + i) \\ d &= E^{s}_{H(FB)}(FB) \\ C_{FB} &= C(i,p,d) \end{aligned}$$

The type is an indicator that this is a FB chunk. Types are necessary since nodes storing a chunk handle delete and replace requests differently based on the chunk type. The identifier salt is used for replication and is discussed in Section 3.5.

A hash of the filename makes a poor FBL chunk identifier since it is likely multiple users 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 under 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. Then a namespace-filename identifier is formed by first concatenating the private key, the virtual pathname, and the filename. The results are then hashed to form the actual identifier. A FBL chunk can be specified more formally as follows:

$$\begin{aligned} i &= H(K'_A + \text{vp} + \text{fn} + \text{salt}) \\ p &= E^p_{K'_A}(\text{type} + T + i + H(d)) \\ d &= E^s_{f(K'_A)}(FBL) \\ C_{FB} &= C(i, p, d) \end{aligned}$$

where $f()$ is some deterministic function used to derived the common symmetric key from the user's private key, *vp* is the virtual path, *fn* is the filename, and T is a timestamp. The type and salt are used in a similar manner as for FBs.

The public metadata provides ownership information useful when implementing secure chunk replacement and deletion (as described in Section **??**.) 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 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.

3.4 Sharing

Although pStore hopes to exploit the vast amount of unused disk space on the Internet, it is naive to assume that this storage space is unlimited. Unlike file sharing peer-to-peer networks, a pStore user cannot directly make use of the storage space he contributes to the system. To encourage fair and adequate donation of storage space, pStore may require a policy where the amount of space available to a user for pStore backups is proportional to the amount of personal 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 between versions, files, and users.[1] One might imagine a system where users recieve 'discounts' on how much storage they must contribute to the system if some of the files they wish to back up can be shared between multiple users.

FB sharing between versions occurs explicitly when adding a new version to a FBL. If the new version contains any of the old FBs, then they are simply referenced in the new version's list of FBs. 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.[2] This is a result of using a hash of the encrypted contents as the identifier for a file block. If we assume that duplicate file blocks are encrypted with the same key, then two identical file blocks will have the same identifier and be stored only once in the network. As mentioned in Section 3.2, convergent encryption ensures that identical file blocks are encrypted with the same key. 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].

FBLs are not shared since they are inserted into the network under a private namespace. Even if two users have identical FBLs, they will have different identifiers and thus be stored separately in the network. Using a content hash of the FBL as the FBL identifier permits FBL sharing when the version histories, file attribute information, and file content are identical. This would be similar to the technique used to store inodes in CFS [7]. Unfortunately, this drastically increases the complexity of updating a file. The entire virtual path must be traversed to find the FBL, and then after the FBL is modified, the virtual path must be traversed again to update the appropriate hash values. Even if pStore allowed FBL sharing, the number of shared FBLs would be small since both the version history and the file attributes of shared FBLs must be identical. pStore

[1] If time permits, we may investigate other techniques such as data compression and information dispersal algorithms (as in [15, 4]) to further increase effective capacity for a user.

[2] 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.

still allows sharing of directory information and actual file data through FB sharing but keeps version information and file attributes distinct for each user.

As an example, assume user A wishes to insert a file f_A and user B wishes to insert a file f_B with identical content. f_A and f_B will be divided into two FBLs and several FBs and then inserted into the network. There will be a FBL for each user, and these FBLs will both point to the same FBs. Now assume user B modifies his copy of the file by adding a new page to the beginning of the file. User B can still share the FBs that are in common with the original, but must insert a new FB which contains the new paragraph. Figure 2 illustrates this example.

3.5 Replication

pStore supports exact-copy chunk replication to increase the reliability of the system. 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 [15]. Although pStore uses exact-copy chunk replication to simplify the implementation, these techniques are certainly applicable and may be provided 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 replication retrieval. This replication technique is in contrast to chain replication. Chain replication involves sending data and a counter to one node and requesting that the node copy the data, decrement the counter, and pass it along to a third node [7, 5]. Malicious nodes can seriously reduce the effectiveness of chain replication at exactly the time when replication is most important. Any node along the chain can refuse to pass the data along, potentially preventing replicas from reaching benign nodes. pStore's replication technique avoids these problems by sending replicas directly to the target nodes. If one of those nodes is malicious, the user can still retrieve the replica from any of the other nodes.

Many systems rely on caching along the lookup path to further increase data replication [5, 7, 10], but pStore is poorly suited to this form of caching. Although FBLs are accessed on every backup, they are never shared. FBLs cached along the lookup path would never be accessed by anyone other than the user. FBLs can instead be cached on the owner's local machine. FBs are shared but are rarely accessed. pStore relies solely on the initial distribution of chunk replicas to provide the necessary level of reliability.

3.6 Deletion and Replacement

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 FLB 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, 6, 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 3.3. This public metadata can be thought of an *ownership tag* which authorizes an owner to replace or delete a chunk.

Each storage node keeps an ownership tag list (OTL) for each chunk. It is an error for there to be more than one ownership tag for each FBL chunk (since FBLs cannot be shared). A user can make a request to delete or replace a chunk by inserting a *command chunk* into the network. A command chunk includes a command (either delete or replace) in the type field of the public metadata. A delete command chunk has no data, while a replace command chunk includes the replacement data and is only applicable for FBL chunks. The command chunk has the same identifier as the chunk to delete or replace.

When a storage node receives a command 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 command chunk, then the command is allowed to proceed. For deletion, 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

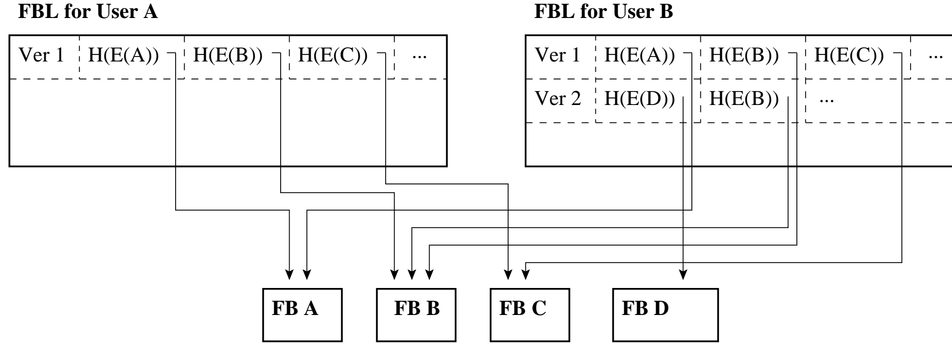

Figure 2: Sharing File Blocks: Blocks are shared between user A and user B and are also shared between two of different versions of the file for user B.

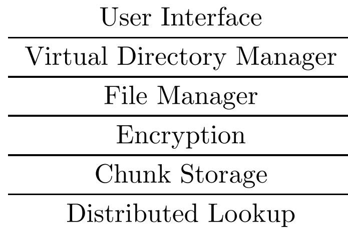

Figure 3: pStore Layered Architecure

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. For replacement, the current chunk is replaced with the data included in the replacement command chunk. It is an error to attempt to replace a FB 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.

4 Layered Architecture

pStore is implemented in a layered architecture with six layers as shown in Figure 3. The *user interface layer* is how the user interacts with the pStore library. Possible pStore user interfaces include a basic command line interface or graphical user interface for backing up a limited number of single files. Layering a file system on top of pStore is another attractive option. Both read/write and read-only file systems are possibilities. A read/write file system would allow files to be inserted into and retrieved from pStore by copying them to and from the file system. A read-only file system might use a daemon to periodically insert a directory snapshot into the network, and then allow a user to browse various snapshot versions through the file system. This would be similar to the NetApp WAFL file system [3].

The *virtual directory manager* handles virtual path traversal, directory creation, and directory removal. When the user interface layer wants to store a file in pStore it does so by identifying a local file to store, and a virtual directory where that file is to be located. Similarly, if a user wants to retrieve a file from pStore it identifies a filename and virtual directory. The directory manager layer is responsible creating and updating virtual directories. Virtual directories may or may not correspond to actual directories on a user's local storage system.

The *file manager layer* inserts a file into the pStore network. The virtual directory layer gives the file manager layer, the file contents and filename. This layer handles breaking a file ianto a FBL and several FBs and then passing them along to the lower levels. The *encryption layer* is responsible for assembling chunks with the appropriate public metadata. This layer handles all public key encryption/decryption, symmetric key encryption/decryption, and one-way hashing. The *chunk storage layer* actually stores chunks for other nodes in the network. It is reponsible for returning the appropriate chunk when asked for a specific identifier and correctly handling delete and replace command chunks.

We plan on using Chord for the *distributed lookup layer* in pStore [6]. Chord offers attractive guarantees concerning search times without assuming any specific file access pattern. 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.

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 snapresore: Advances in snapshot technology. Technical report, TR3043, NetworkApplicance, 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, July 2000. International Computer Science Institute.

[6] 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.

[7] 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.

[8] J. Daemen and V. Rijmen. The rijndael block cipher, 2000.

[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, July 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. Mazi‘eres. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 181–196, October 2000.

[12] The gnutella protocol specification v0.4. distributed search services, September 2000.

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

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

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

[16] B. Schneier. *Applied Cryptography*. John Wiley and Sons, 1996.

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

[18] 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*, August 2000.