ABS: The Apportioned Backup System

Joe Cooley, Chris Taylor, Alen Peacock
MIT 6.824 Final Project
cooley@ll.mit.edu,taylorc@mit.edu,peacock@ll.mit.edu

Many personal computers are operated with no backup strategy for protecting data in the event of loss or failure. At the same time, PCs are likely to contain spare disk space and unused networking resources. We present the Apportioned Backup System (ABS), which provides a reliable collaborative backup resource by leveraging these independent, distributed resources. With ABS, procuring and maintaining specialized backup hardware is unnecessary. ABS makes efficient use of network and storage resources through use of coding techniques, convergent encryption and storage, and efficient versioning and verification processes. The system also painlessly accommodates dynamic expansion of system compute, storage, and network resources, and is tolerant of catastrophic node failures.

1 Introduction

Typical backup systems suffer from several constraints imposed largely by centralized architectures. Centralization prohibits scalability; adding capacity is rarely as easy as adding another computer to the system with more disk storage. Today's backup systems require dedicated hardware resources, and these resources are usually geographically localized, making them susceptible to simultaneous failure. Finally, typical backup systems ultimately require administrative resources in addition to those of maintaining the client systems. Some systems compensate for the locality inherent in a centralized design by providing offsite backup, but such compensation comes at the expense of administrative and maintenance costs.

At the same time, many of the client machines which rely on backup systems contain unused storage and computational resources [1], exhibit geographic diversity, and are already maintained by their users or administrators. The utility of these features is invariant with the number of client machines considered. In other words, these features scale.

These observations have resulted in the creation of several self-managed, distributed systems which take advantage of peer-to-peer relationships to provide backup services [2, 3, 4, 5] (see Section 8). In this paper we describe ABS, which presents several novel features not seen in distributed backup systems thus far, including an rsync-based versioning scheme, a novel storage verification process, and storage/recovery guarantees in the face of failures.

2 Goals

A number of trade-offs can be made when designing a distributed backup system, dependent on how the system is expected to be deployed. The implementation of ABS that we describe in this paper focuses on the use case of 10s of PCs connected via a LAN or through broadband Internet connections. We believe the architecture presented can support other types of environments with little modification. Though users of an ABS cluster are expected to generally trust one another, the collaborative nature of the system dictates that safeguards must be in place to prevent unauthorized access to stored data and to prevent abuse of the system by "greedy" users. ABS is designed to: 1) allow nodes to join, leave, and fail with some frequency, without losing data or preventing backup and restore operations, 2) provide resiliency to data loss on nodes that have not failed, 3) store data securely, so that only authorized users may access content, 4) provide archival capabilities for both current and previ-

ous versions of the same file, 5) provide some level of resistance to misbehavior, including modified nodes which may wish to "cheat" by storing less data than they claim, and 6) provide efficient, load-balanced storage of data,

The system must also be easy to use and administer. The backup system will be used to recover corrupted or accidentally deleted files and access previous versions of existing files. If the user's local storage fails, complete recovery should be possible regardless of whether the user has a copy of the list of files that were previously stored.

3 System Architecture

The design of ABS addresses the goals outlined in Section 2 by employing: 1) coding techniques to deal with storage node transience and data loss, 2) encryption to provide privacy and security, 3) convergent encryption and convergent storage (also called single instance store [6]) for efficient storage of data, 4) file versioning [2] for efficient archival of data, 5) a novel, resource-efficient data verification scheme for misbehavior prevention, and 6) a distributed hash table [8] to provide storage placement, load balancing, and keyspace management.

ABS is logically separated into two parts: a server and a client.

Every machine continuously runs an instance of the server which provides a distributed block store view to the client. This store allows data of arbitrary length (*fragments*) to be stored under 20-byte *keys*, along with fragment related metadata (*fragment metadata*). Key/data/metadata tuples are accompanied by *signatures*, which are private key signatures over the data/metadata. These signatures are used as 1) an integrity check on the data, and 2) as a statement of ownership of the data. These same keys are used to sign delete requests, so the signature allows a server to verify that a delete request is legal and authentic.

The block store hides the complexity of dealing with server transience and distributed storage behind an API used by the client. This allows the client to view the block store as a single monolithic entity that will perform best effort put/get operations, guaranteeing fragment retrieval if a fragment exists on an active server. The details of the block store are covered in more detail in section 5.

The client's responsibility is to use the properties provided by the block store to provide meaningful backup services to a user. The client chooses, given a file, what fragments should be stored, under which keys, and what metadata should accompany those key/data pairs. The client also monitors the status of fragments it has added to the system in the past, and is responsible for ensuring that the correct fragments are available; for example, if a server crashes, the client is responsible for determing what data was lost and replacing it in the network if necessary.

The client provides several services to users. The most important service is recovery. In the event of a catastrophic failure, the user needs only his public/private key pair for the client to completely recover the user's files. This service is provided using the signature infrastructure of the block store. The second service is versioning, which provides efficient backup archiving and protection from accidental file overwrites. The details of the client operations are contained in section 4.

The client uses a coding library to generate fragments for storage. This provides the client with some resilience to fragment losses in the block store (which occur unavoidably due to downtime of participating nodes). These fragments are also compressed to minimize network bandwidth and storage requirements. Furthermore, fragments are encrypted to protect the privacy of users. Finally, the storage keys for the fragments are generated in a canonical way, so that ABS can realize the benefits of *single instance store*: if two users store the same file contents, their clients will generate the same fragments independently, and only one copy of each fragment will be stored in the network.

The client and server also implement a *verification* operation, a lightweight method by which the client can confirm that the server is making a good faith effort to store a fragment.

4 The ABS Client

This section is structured functionally; section 4.1 covers the operations involved in generating and storing fragments, section 4.2 covers the reciprocal operation of recovering users' files from the block store, and section 4.3 discusses the actions taken by the client to maintain the availability of fragments in the block store and realize the

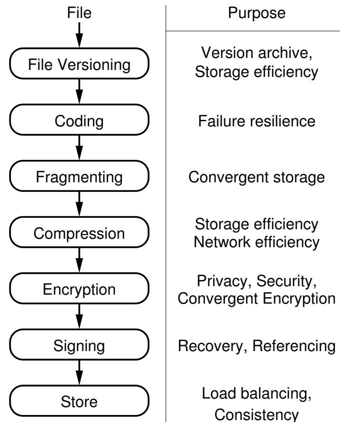

Figure 1: File Storage in ABS

storage benefits of *single instance store*.

4.1 Generation and Storage of Fragments

The user initiates the storage process by selecting files through a GUI or commandline interface, or by configuring a daemon such as `cron` to perform the task automatically. Once files are selected for storage in ABS, the client process performs a sequence of operations on the files to generate fragments for storage. These fragments are then inserted into the block store in a canonical way. The steps involved in storage are shown in Figure 1. The input to this pipeline is a file, with its associated file system metadata.

4.1.1 File Versioning

The ABS client leverages the capabilities of the block store and the *rsync* [9] library to offer file versioning services to users, which helps maintain efficient storage use. It takes a disk file (with associated file system metadata) and produces a new (possibly different) file and metadata for the next stage in the pipeline.

The first time a file is stored, the client uses *rsync* to generate a difference signature over the file (which is called the *basis* file). A difference signature is a compact, hash-based representation of a file that supports fast, file block comparisons between two file versions. Signatures are saved on the client, and can optionally be backed up to the block store as a normal file. Once the difference signature is generated, the file is stored.

If the file being stored already has a difference signature on file, the client uses rsync to generate a *delta file* that encodes the difference between the file being stored and the original basis file. This delta file replaces the modified file for the remainder of the pipeline, and a note is added to the file metadata indicating which file is the basis for the delta.

These deltas can often be smaller than the whole modified file; thus, difference versioning can provide a space savings. However, it may also decrease the availability of the modified file by creating dependencies on prior versions.

4.1.2 Coding and Fragmenting

After versioning is complete, the next step is to split the incoming file into some number of fragments. This split is accomplished following *coding*.

Coding techniques allow for recovery of data in the face of failure by producing data parity. Coded data can then be divided into fragments, where only some number of fragments less than the total number are needed to reconstruct the file.[1] The design of ABS calls for using a version of the large block FEC described in [10], which produces blocks of parity to be appended to the file. The guarantee for retrieval in the face of failure is adjustable.

In order to maintain the invariant that no two fragments belonging to the same file are stored on the same node (which is necessary to provide the above guarantees), the content of the file is hashed to produce a *file collision key*. The block store helps the client guarantee that no two fragments with the same file collision key are placed on the same physical machine. This helps preserve independence between fragment losses, since the fragments from a file are each placed on different servers.

Thus, the coding step takes a file and some metadata and produces n fragments with n associated pieces of metadata. This metadata is a complete copy of the file metadata, plus the information required to identify this fragment's place in the file (e.g. "fragment 3/6, large block FEC").

4.1.3 Compression

Next, fragments may be optionally compressed to reduce their size. Compression occurs before encryption to take advantage of regularity

[1] in our preliminary implementation, we use replication instead of a low-density parity-check codec. Replication is a subset of coding, providing protection for x-1 failures where the number of replicas is x.

within the fragment and occurs after the coding stage to minimize dependencies among stored fragments. Experimental results (detailed in Section 7.1) confirm that this compression strategy yields significant storage reductions.

Compression only covers the fragment, leaving the metadata intact. It does, however, append a note to the metadata indicating whether the fragment was compressed.

4.1.4 Encryption

After compression, each fragment is encrypted using a symmetric encryption function. The key for this encryption is the hash of the fragment's contents. This encryption is meant to prevent nodes storing a fragment from determining the fragment's contents.

The key is chosen to be the fragment hash for several reasons. First, it preserves *single instance store* [6], since identical fragments will remain identical after encryption, even when encrypted by different users. This is also known as *convergent encryption* [11]. Second, it suggests that only users with access to the fragment's unencrypted content can generate the key needed to decrypt the encrypted fragment.

The key used in the encryption is added to the fragment metadata, and the fragment metadata is finalized by encrypting it using the user's public key. This ensures the privacy of the metadata and protects the encryption key added to the metadata during the encryption step.

4.1.5 Signing

The final step prior to storage is the generation of a signature for the fragment. The signature is taken over the entire encrypted fragment and the finalized metadata using the user's private key.

The signature is significant for several reasons. First, it is an integrity check on the data and metadata. Second, it acts as a statement of ownership: a server in the block store can now tell that the user owns the fragment. During recovery from a catastrophic failure, the user may only have access to his public key pair: the signature allows the server to provide the user with any fragments he has stored. It also allows the user to validate the data he receives during the recovery. The signature's statement of ownership is also important when a user wishes to remove data from the backup system; only a removal request signed by the same key as the data/metadata signature will be honored by the block store.

4.1.6 Storage

Now the fragments are ready for storage. First, the client generates a *storage key*, which is the hash of the key used to encrypt the fragment. This method of storage key generation provides two desirable benefits. First, it maintains the *single instance store* property, since identical encrypted fragments will have the same storage key. Second, the storage key is easily recovered from the unencrypted fragment, so a user cannot lose a storage key while retaining the corresponding data.

The client can now send the storage key, the encrypted fragment, the finalized fragment metadata, and the fragment signature to the block store. In the normal case, the block store will save the fragment, metadata, and signature under the storage key. If two identical fragments are stored by different users, only one instance of the fragment is stored, but separate metadata and signatures are saved for each user. When a fragment is removed, the corresponding metadata and signature are deleted; the fragment itself is only removed if there are no more metadata/signature pairs present. Thus, the signature and metadata stored by the block store serve as a crude form of reference counting on the fragments.

In some cases, the block store will be unable to save a fragment and its auxiliary data under a particular storage key. This could happen because the file collision key generated during coding collides with another fragment stored on the same physical machine. It could also occur if the physical machine for the storage key is unavailable, or if the machine has insufficient disk space to hold the fragment.

In these cases, the client simply generates a new storage key by computing the hash of the old storage key. It then retries the operation using the new key. This process continues recursively until an acceptable storage key is found or the client has tried all hosts on the network, in which case the store operation simply fails.

After the operation is complete, the original storage key and the hash depth required to store the fragment are recorded in a client side database to simplify retrieval. This database may also be stored as a file in the block store. Note that this database is completely optional, since

the fragment can always be recovered by polling all the nodes in the block store with the user's public key.

4.2 Retrieval of Fragments and File Recovery

The file retrieval operation is simply the reverse of the storage operation. Given a file to retrieve, the client looks in its database to determine which fragments it needs and the storage keys corresponding to those fragments, then requests the fragments from the block store. If the database is lost, then the client uses the user's public key to recover the database from the block store; or, failing that, simply recovers as many fragments as possible from the block store by exhaustive search on the user's public key until the necessary fragments are found.

Once the fragments are retrieved, the client verifies their signatures, then decrypts the finalized metadata. It then decrypts the fragments using the keys from the corresponding fragment metadatas. If necessary, it also decrypts the fragments. Next, it uses the information in the metadata to invert the coding step and reassemble the complete file data from its fragments. Finally, it adds the file to the local file system, using the file system metadata in the fragments' metadata.

4.3 Maintenance of the Block Store

The client is also responsible for ensuring that files stored in the block store remain available. The client runs two processes to do this: merge high and data verification.

4.3.1 Merge High

In some cases, the initial storage key for a fragment may be unacceptable due to load-balancing requirements, unreachability of destination storage node, or file collision key requirements. Relocating a fragment is accomplished by rehashing a fragment's storage key and reinserting it into the system.

To ensure that convergent storage occurs to the largest extent possible for such fragments, clients are responsible for moving fragments as close to the initial storage key as possible using a process we call *merge high*. As part of the periodic verification process (explained in the next section), a client attempts to move key/data pairs 'up' the hash chain defined by the storage key. This assures that identical fragments stored by separate clients will eventually converge in storage if possible.

4.3.2 Data Verification

Periodically, the client verifies that storage servers still hold the correct fragments by employing a technique that maintains *O(1)* network operations between any client/server pair, independent of the number of fragments stored by the server on behalf of the client.

The algorithm is as follows: prior to storing an encrypted fragment, the client generates a random number *seed* and uses it to generate a string of successive random indices into the fragment data. Byte values from these offsets are concatenated into a string; this string is hashed to create a *fingerprint*.

After the fragment is stored, when the client wishes to verify that the server still holds the fragment, it transmits the seed to the server. If the fragment is present, the server uses the seed in conjunction with the fragment to compute the fingerprint, which it returns to the client. If the fingerprint sent by the server matches the client's saved fingerprint, the client can assume with confidence that the fragment exists within the system.

Cheating is difficult since servers cannot be sure which seed the client will ask for, and would thus need to invest extensive resources creating and storing an exhaustive fingerprint list.

This algorithm may be *chained*: the fingerprint from one fragment becomes the seed for a fingerprint in the next fragment. Doing this allows a client to verify multiple fragments stored on the same server in one compact request/reply pair.

When a verification operation fails, the client can assume that its fragment was lost, regenerate it (provided the corresponding file can still be retrieved), and store it again, perhaps on a different node.

This verification scheme is a lightweight way of providing modest protection against fragment loss. It assumes the compare-by-hash risks described in [13]. It does not provide strong guarantees. Further study is warranted.

5 The ABS Block Store

The backend for ABS is a distributed block store that offers DHT-like semantics with support for file collision keys and signature reference counting. Essentially, the goal of the block store is

twofold: 1) if a fragment exists on any ABS machine in the cluster, it should be possible to find it, and 2) if the fragment exists in the ABS system, it should be easy to locate, using relatively few network queries. In every design decision, the former trumps the latter. The most crucial thing about a store for a backup system is that it recover data, if there is data to recover.

5.1 Key slices, the slice table

In order to efficiently support storage and retrieval of fragments, the space of all possible storage key values is divided into equal-sized discrete regions called *slices*. A system-wide table, the *slice table*, maps ABS nodes to hash slices. In the steady state, each slice has exactly one node that is used for both reading and writing of fragments. However, as nodes enter and leave the system, it is possible that several nodes will end up holding keys for a slice, resulting in multiple read nodes. In a well-formed table, therefore, each slice is mapped to at most one *write node*, and possibly several *read nodes*. Read nodes slowly transfer fragments to the write node, so that steady state eventually converges to one read/write node per slice.

To insert data into the fragment store under a particular key, the client looks up the write node associated with the key's hash slice, and sends the data to that node. When attempting to recover data for a key, the client checks the read nodes in sequence until it finds a match.

Some slices may not have a write node. Clients treat slices without write nodes as though the write node for the slice was offline; the fragment key is rehashed, and this new value is used for storage. If a write node later comes online, the merge high process migrates the data to the new node (see Section 4.3.1). If a read node has no keys for its slice, the node removes itself from the list of read nodes for that slice. In steady state, each slice has exactly one node which is both its read and write node.

Slices and the slice table guide the client to the correct server quickly during key lookups and insertions. Given a storage key, a glance at the slice table allows a client to determine the correct server to query.

5.2 File collision keys and signature reference counting

The fragment store extends the traditional DHT semantics by providing support for file collision keys and signature reference counting/single instance store. Nodes guarantee that only one key/value pair is held for any particular fragment collision key. An attempt to store a pair with a duplicate collision key results in an error. The client can respond by rehashing the key and therefore placing the data on a different node.

Nodes also keep track of signature/metadata information for each logically distinct client that stores a key/value pair. The key/value pair is only removed when all the signatures/metadata pairs have been deleted.

5.3 Maintaining the slice table

Preserving the correctness of the slice table across node failures and network partitions is essential to the correctness of the backup system. The ABS cluster elects a leader as soon as it is brought online. This leader is responsible for maintaining the authoritative copy of the slice table. All ABS nodes keep local copies of the slice table, but periodically refresh from the leader. Changes to the slice table are always made and propagated from the leader. If the leader goes offline, an election is held to find a new leader, and all nodes download copies of the slice table from that leader. This strategy is sufficient, since the merge high and join processes ensure that the new table evolves to correctly describe the system. The slice table is effectively soft state.

To join the ABS cluster, a new node locates the leader, and asks to be added. The node then retrieves a copy of the modified slice table and begins operating. The leader changes the table as follows:

- if there is a slice without a write node, the leader assigns the new node as the write node for the slice that was least recently held by a node. If no slices are empty, the process is more complicated (see Section 5.4).
- the new node is added as a read node for any slices for which the new node already has key/value pairs. This could occur when nodes recover from downtime.

Over time, the new node takes on additional responsibility, since data is migrated from read

nodes for the slice to the new node. Furthermore, merge high operations will most likely begin shifting keys to the new node. The new node also takes responsibility for new writes to that key space.

Whenever a node retrieves a copy of the slice table, it checks to make sure that it has write responsibility for at least one slice and that it has read responsibility for all of the slices for which it has key/value pairs. If it does not have write responsibility, it rejoins the network with a join call. If it does not have a read responsibility, it informs the current leader, so that the slice table can be updated appropriately.

To leave the system, a node informs the leader that it wishes to leave. The leader then marks the node as no longer holding write responsibility for any slices. A well behaved exiting node will remain online until all of its data is transferred to new nodes. However, this is not required for correctness.

Unreachable nodes do not necessarily lose their write slices. Unreachable nodes are marked in the slice table as *inactive*. Inactive nodes are bypassed for purposes of storing keys, just as though the slice were unassigned. However, when a down node returns, it is likely to keep its old slice, minimizing the amount of data relocation required.

This system does not guarantee that all nodes have the same view of the slice table. However, different slice tables on different hosts is acceptable, as long as the tables eventually converge, since there is a mechanism for reshuffling data (e.g. the merge process). We envision that node joins and leaves will be fairly infrequent, with several hours between events, so a gradually converging process is likely to be successful.

5.4 Empty slices

The system is likely to work best if there are usually hash slices empty. Joins are thus trivial and involve minimal copying. Empty hash slices cost storage and lookup very little time, since an empty hash slice can be bypassed without a network query. We suspect that having roughly twice as many hash slices as physical hosts is a good balance. In the event that the cluster grows more than anticipated, it is possible to resize slices. Each slice can simply be partitioned into two slices. The old write node for the large slice becomes the write node for only one of the new slices, but the read node for both. If such slice resizes are to be done, it most likely makes sense to do them rarely, and only in the circumstance of a cluster expanding to twice its originally envisioned size.

5.5 Network partitions

Network partitions are manageable under this scheme. After a partition heals, members of the cluster will notice that there are two leaders (possibly by periodically asking other nodes who the leader is). The two leaders can choose a new leader between themselves, and the two slice tables can be merged. Since the slice tables are essentially soft state, the precise mechanics of the merge are not important. Even the trivial merge, where the new leader executes a join operation for each node from the other side of the partition converges to a well formed slice table. We did not implement code to handle network partitions due to time constraints; however, doing so in the future should be straightforward.

6 Implementation

Our current implementation of ABS is built using BerkeleyDB as the backing store for the client and storage processes. Storage servers and clients communicate via RPC, and both rely on library functionality provided by SFS, gcrypt, and gpgerror. Versioning is accomplished with the help of rsync. A coding library wrapped around LDGM/LDPC also forms part of our implementation, though storage and retrieval operations do not currently use it. Client utilities include command-line programs as well as a GUI interface. In addition to these tools, we have also built a visualization tool to aid in demonstration and development of ABS, and which allows us to monitor the amount of data being stored on nodes, view logs from nodes, and stop, fail, and restart nodes.

6.1 Interfacing

ABS aims to provide a design that enables uncomplicated, cross-platform access for its users. Multiple modes coexist in support of these goals. A set of command line and GUI utilities allows the user to store, retrieve, and view information. Perhaps more interestingly, a NFS and Samba client could be created to allow the system to maintain typical filesystem semantics, thus enabling deployment in a wide variety of environ-

ments with a broad range of operating systems. In all interface modes, clients maintain a default configuration which users can manipulate.

7 Analysis

Storage efficiency in ABS is provided by file compression, convergent storage, and file versioning. The presence of these is justified by the measurable benefit as weighed against its cost, as examined in Sections 7.1, 7.2, and 7.3. Section 7.4 explores availability guarantees as a function of node failures and coding protection.

7.1 File Compression

Initial trials with a small number of files shows that data which has been coded with the large block FEC mentioned in Section 4.1.3 maintains compressibility with the same attributes as unencoded data; although the added parity blocks are not generally compressable, the non-parity remainder of the file achieves roughly 99% of its non-coded reduction in size across a range of file types and sizes when the file (including coded parity data) is compressed in its entirety. In other words, the storage benefit is essentially the same as if the non-coded fragments of the file were compressed while the parity fragments were not.

As for the cost of compressing file fragments instead of files in their entirety, trials with gzip were performed on files of different types (postscript, C++ source code, a binary executable, and an mp3 audio file) ranging in size from 44K to 11M. Measurements were taken using gzip on the whole files, on the files split into 10 fragments, and on the files split into 100 fragments. All examples (except for the already compressed mp3 file) compressed to 22.7%-41.4% of their original size. Compressibility of the fragmented files matched that of the whole file with an extra storage cost of between 0.5%-7%, and an extra computation cost of between 4%-18%. The slight loss of efficiency due to fragmenting is a valid tradeoff for increased efficiency in convergent storage.

7.2 Convergent Storage

According to [11], over 90% of machines share common files, justifying the inclusion of convergent storage in the system. Its use will help maintain efficient storage use by reducing stored volume on initial system insertion. File versioning, analyzed below, will maintain storage efficiency as files change over time.

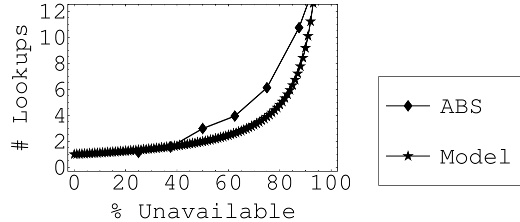

Figure 2: Average number of lookups required to avoid coding collisions

The major cost associated with ABS' implementation of convergent storage is that of ensuring that fragments belonging to the same file end up being stored on different physical nodes. This requires that when intra-file fragment collisions occur, the colliding fragment must have a new storage key generated, requiring further lookup(s). .

7.2.1 Networking

Figure 2 depicts these measured and modeled lookup costs. We forego a model discussion here for purposes of brevity. The x-axis denotes the percent of total system, $\frac{m}{n}(100)$, machines currently storing a fragment from a given file. The y-axis denotes the number of storage operations necessary to find a storage node that avoids collisions.

Each experimental data point represents 100 write operations to 8 storage servers. The operation consists of writing multiple replicas of randomly generated 1KB files. The number of replicas stored per point increases from 2 to 8 (25% to 100% unavailability) with increasing x. The figure positively validates the analytical model.

Interestingly, until file fragments from one file have been stored on approximately 80% of the system's storage nodes, collisions rarely occur. Even more importantly, as the size of the system increases, the operating region of the graph approaches the y-axis, i.e. $\lim_{n\to\infty} \frac{m}{n} = 0$ assuming m remains some fixed integer. System performance improves as system size increases.

7.3 File Versioning Efficiency

Versioning efficiency was investigated by conducting four experiments. Experiments were divided into versioning and non-versioning classes. Both experiment classes were begun by writing 320 randomly generated files of size 32KB into ABS

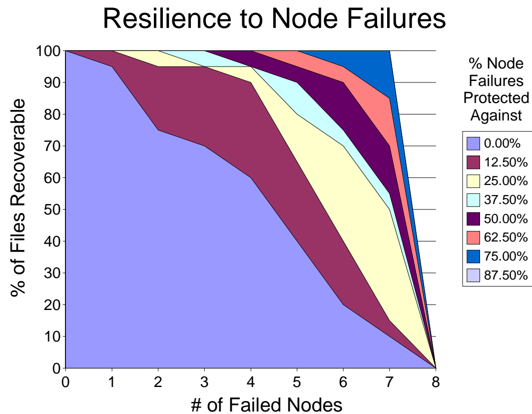

Figure 3: ABS Resilience to Node Failures

using 2 replicas.

In the first two experiments, nine rounds of the following procedure was executed. First, every byte in each file was randomly mutated. In the versioning class, changes between new and old versions were written into ABS. In the non-version class, the entire new version was written into the system.

Both classes consumed 240MB of storage. Because in each experiment entire files were randomly regenerated, one would expect subsequent versions to drastically differ. As such, the versioning experiment should and does approach the non-versioning in storage consumption because each new version is a more-or-less completely new file.

In the second two experiments, nine rounds of changes were appended to each file. Changes consisted of 32KB of randomly generated bytes. Upon completion, the versioning class consumed 240MB of storage, while the non-version class consumed greater than 1.1 GB. These two experiments highlight the benefits of versioning when file changes do not include entire files.

7.4 Resiliency to Failures

ABS guarantees survival of data in the face of some amount of node failures. The amount of failure ABS protects against can be chosen by the user, though standard values are provided by coding defaults. The graph in Figure 3 demonstrates how failure resiliency varies with node failure and coding protection.[2] The data presented here was collected on an 8 node ABS cluster by storing and attempting to retrieve 20 random files. Each file was protected against the percentage of node failures listed in the legend. Interestingly, ABS allows a user to retrieve 90% of his files even with 6 failed nodes, so long as 4 replicas of each are stored. In other words, the system provides reasonable best-effort recovery when more nodes fail than coding protects against.

8 Related Work

Since storing files for backup is similar in many regards to storing files for live use, the earliest work on distributed backup systems built heavily on ideas and mechanisms developed for use in distributed file systems such as PAST [14], Farsite [6], OceanStore [15], and CFS [16]. Each of these systems has the capability to serve as a backup device, but their designs were intended for more general use, with none specifically tailored to the backup task.

The first work specifically targeted at distributed backup that we are aware of is the pStore cooperative backup system [2]. pStore used Chord [8] to store file object replicas to a number of nodes on the backup cluster in a manner similar to that proposed in ABS, but operated at the file block level to achieve convergent storage. pStore exhibited some resilience to node failures, with preliminary experiments showing that roughly 5% of data was lost when 7 out of 30 nodes failed. The Phoenix Recovery System [4] built on ideas in pStore to build a backup system which provides some resiliency to malicious Internet epidemics by ensuring that backups are stored on a set of systems that are diverse (in terms of the operating system) from that of the client. Pastiche [3] took the opposite approach to maximize storage efficiency; backup peers are selected mainly by the criteria that they are already similar (in terms of native files). Pastiche incorporated several useful mechanisms for further increasing storage efficiency, including use of content-based indexing to find common data across different files and convergent encryption to ensure that identical data stored by multiple

[2] Coding protection in this experiment was provided through replication, though more efficient codecs can reduce the space requirements for the same amount of protection.

users would not be duplicated for each storage operation. In Samsara [5], the authors of Pastiche explored solutions to the problem of malicious nodes and greedy use through enforcement of consumption symmetric with contribution.

9 Conclusions

In this paper we have presented a design for a distributed backup system which combines a collection of known techniques in a novel way, employing convergent storage and file versioning to minimize storage. A novel verification process adds consistency checking with minimal server load. Cryptographic techniques have been employed to maintain data integrity and privacy. We have developed a membership strategy that allows the system to store and retrieve information in a transient environment characterized by changing storage server availability. We described an implementation and made observations concerning aspects of system scalability and analyzed system performance, showing that our storage scheme remains network and space efficient while maintaining reliablility. Experimental evidence suggested that collisions and network retries are rare when less than 80% of block servers contain a file fraction. Fragment compression was shown to save between 22.7%-41.4% over no compression. File versioning consumed similar volume to non-versioning in the worst case and four times less in the best. Coding techniques allow data to be stored with 100% guarantee of retrieval, provided that a certain percentage of nodes in the cluster remain available, and also provide graceful, best-effort recovery when large numbers of nodes failed.

References

[1] J. R. Douceur and W. J. Bolosky, "A large-scale study of file-system contents," in *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 59–70, ACM Press, 1999.

[2] C. Batten, K. Barr, A. Saraf, and S. Trepetin, "pStore: A secure peer-to-peer backup system," Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002.

[3] L. P. Cox, C. D. Murray, and B. D. Noble, "Pastiche: Making backup cheap and easy," in *The Fifth ACM/USENIX Symposium on Operating Systems Design and Implementation*, (Boston, MA), December 2002.

[4] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker, "The phoenix recovery system: Rebuilding from the ashes of an internet catastrophe," in *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, (Lihue Hawaii), May 2003.

[5] L. P. Cox and B. D. Noble, "Samsara: Honor among thieves in peer-to-peer storage," in *The 19th ACM Symposium on Operating Systems Principles*, (Bolton Landing, NY), October 2003.

[6] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: federated, available, and reliable storage for an incompletely trusted environment," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 1–14, 2002.

[7] R. C. Burns and D. D. E. Long, "Efficient distributed backup with delta compression," in *I/O in Parallel and Distributed Systems*, pp. 27–36, 1997.

[8] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan, "Building peer-to-peer systems with chord, a distributed lookup service," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, (Schloss Elmau, Germany), IEEE Computer Society, May 2001.

[9] A. Tridgell and P. Mackerras, "The rsync algorithm," Technical Report TR-CS-96-05, The Australian National University, June 1996.

[10] V. Roca, Z. Khallouf, and J. Laboure, "Design and evaluation of a low density generator matrix (LDGM) large block FEC codec," in *Fifth International Workshop on Networked Group Communication (NGC'03)*, (Munich, Germany), October 2003.

[11] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs," in *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 34–43, ACM Press, 2000.

[12] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, April 1965.

[13] V. Henson, "An analysis of compare by hash," in *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, (Lihue Hawaii), Sun Microsystems, May 2003.

[14] A. I. T. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Symposium on Operating Systems Principles*, pp. 188–201, 2001.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, 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*, pp. 190–201, ACM Press, 2000.

[16] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *ACM SOSP 2001*, (Banff, Canada), October 2001.