Cumulus: Filesystem Backup to the Cloud

Michael Vrable, Stefan Savage, and Geoffrey M. Voelker

University of California, San Diego

Abstract

In this paper we describe Cumulus, a system for efficiently implementing filesystem system backups over the Internet. Cumulus is specifically designed under a *thin cloud* assumption — that the remote datacenter storing the backups does not provide any special backup services, but only provides a least-common-denominator storage interface (i.e., get and put of complete files). While Cumulus can thus use virtually any storage service, we show that it still provides efficiencies comparable to integrated approaches.

1 Introduction

It has become increasingly popular to talk of "cloud computing" as the next infrastructure for hosting data and deploying software and services. Not surprisingly, there are a wide range of different architectures that fall under the umbrella of this vague-sounding term, ranging from highly integrated and focused (e.g., Software As A Service offerings such as Salesforce.com) to decomposed and abstract (e.g., utility computing such as Amazon's EC2/S3). Towards the former end of the spectrum, complex logic is bundled together with abstract resources at a datacenter to provide a highly specific service — potentially offering greater performance and efficiency through integration, but also reducing flexibility and increasing the cost to switch providers. At the other end of the spectrum, datacenter-based infrastructure providers offer minimal interfaces to very abstract resources (e.g., "store file"), making portability and provider switching easy, but potentially incurring additional overheads from the lack of server-side application integration.

In this paper, we explore this *thin-cloud* vs. *thick-cloud* trade-off in the context of a very simple application: filesystem backup. Backup is a particularly attractive application for outsourcing to the cloud because it is relatively simple, the growth of disk capacity relative to tape capacity has created an efficiency and cost inflection point, and there are few entrenched business cases for keeping backup local. For end users there are few backup solutions that are both trivial and reliable (especially against disasters such as fire or flood), and ubiquitous broadband now provides sufficient bandwidth resources to offload the application. For small to mid-sized businesses, backup is rarely part of critical business processes and yet is sufficiently complex to "get right" that it can consume significant IT resources. Finally, larger enterprises benefit from backing up to the cloud to provide a business continuity hedge against site disasters.

However, to price cloud-based backup services attractively requires minimizing the capital costs of data center storage and the operational bandwidth costs of shipping the data there and back. To this end, most existing cloud-based backup services (e.g., Mozy, Carbonite, Symantec's Protection Network) implement integrated solutions that include backup-specific software hosted on both the client and at the data center (usually using servers owned by the provider). In principle, this approach allows greater storage and bandwidth efficiency (server-side compression, cleaning, etc.) but also reduces portability — locking customers into a particular provider.

In this paper we explore the other end of the design space — the thin cloud. We describe a cloud-based backup system, called Cumulus, designed around a minimal PUT/GET/LIST/DELETE interface that is trivially portable to virtually any on-line storage service. Thus, we assume that *any* application logic is implemented solely by the client. In designing and evaluating this system we make several contributions: First, we show through simulation that, through careful design, it is possible to build efficient network backup on top of a generic storage service — one competitive with integrated backup solutions, in spite of having no specific backup support in the underlying storage service. Second, we have built a working prototype of this system, using Amazon's Simple Storage Service (S3), and demonstrated its effectiveness on real end-user traces. Finally, we describe how such systems can be tuned *for cost* instead of for bandwidth or storage, both using the Amazon pricing model as an example as well as for a range of storage to network cost ratios.

In the remainder of this paper, we first describe prior work in backup and network-based backup, followed by a design overview of Cumulus and an in-depth description of its implementation. We then provide both simulation and experimental results of Cumulus performance, overhead and cost in trace-driven scenarios. We conclude with a discussion of the implications of our work and how this research agenda might be further explored.

2 Related Work

Many traditional backup tools are designed to work well for tape backups. The dump, cpio, and tar [9] utilities are common on Unix systems and will write a full filesystem backup as a single stream of data to tape. These utilities may create a full backup of a filesystem, but also support *incremental* backups, which only contain files which have changed since a previous backup (either full or another incremental). Incremental backups are smaller and quicker to create, but mostly useless without the backups on which they are based.

An organization may establish a backup policy establishing at what granularity backups are made, and how long they are kept. A policy might be that backups are made each night and kept for one month, except that one backup each week is kept for six months instead of one. A mixture of full and incremental backups can more efficiently implement a backup policy. Long-term backups may be full backups so they stand alone; short-term daily backups may be incrementals for space efficiency. Tools such as AMANDA [2] build on dump or tar, automating the process of scheduling full and incremental backups as well as collecting backups from a network of computers to write to tape as a group.

Tivoli Storage Manager [7] offers "progressive incremental backup", in which only one full backup of a filesystem is needed. After that, only changes are stored; a database tracks what file data is stored where. If backing up to tape, old tapes may contain a mixture of still-needed data and dead space; a tape reclamation process may compact the useful data onto fewer tapes to recover space.

The falling cost of disk relative to tape makes backup to disk more attractive, especially since the random access permitted by disks enables new backup approaches. As a general rule, though not universal, disk-based backup tools avoid the need for occasional full backups (as Tivoli Storage Manager does). After the first backup, only changes need to be sent. Doing so significantly reduces bandwidth needed for backups and is particularly important if backups are sent over a wide-area network for storage.

A mirror is the simplest disk-based backup approach: keep on a second disk a copy of the data which is to be backed up. Rsync [14] can efficiently mirror a filesystem from one disk to another across a network, transferring only those parts of files that have changed. Users usually want backups at multiple points in time; rsnapshot [11] is a wrapper around rsync that will store multiple snapshots, each as a separate directory on the backup disk. Unmodified files are hard-linked between the different snapshots, so storage is space-efficient, and snapshots are easy to delete.

	Multiple snapshots	Simple server	Incremental forever	Sub-file delta storage	Encryption
rsync			✓	—	
rsnapshot	✓		✓		
rdiff-backup	✓		✓	✓	
duplicity	✓	✓		✓	✓
Brackup	✓	✓	✓		✓
Box Backup	✓		✓	✓	✓
Cumulus	✓	✓	✓	✓	✓

Multiple snapshots: Can store multiple versions of files at different points in time; *Simple server:* Can back up almost anywhere; does not require special software at the server; *Incremental forever:* Only initial backup must be a full backup; *Sub-file delta storage:* Efficiently represents small differences between files on storage; only relevant if storing multiple snapshots; *Encryption:* Data may be encrypted for privacy before sending to storage server

Table 1: Comparison of features among selected tools that back up to networked storage.

rdiff-backup [4] is designed for efficient storage of snapshots, using a mirror plus reverse incrementals approach: the most recent snapshot is a mirror of the files, as in rsnapshot, and the rsync algorithm is used to create compact deltas for reconstructing older versions. The reverse incrementals are more space efficient than full copies of files as in rsnapshot. Snapshots can be deleted, but only starting with the oldest (some snapshots cannot be expired earlier than others).

The previous disk-based backup tools rely on server functionality (for applying the rsync algorithm), and store data unencrypted on the server. Duplicity [5] stores data in compressed, encrypted form on the server; files are grouped together before storage for better compression and to reduce per-file storage costs at the server. No special server support is needed beyond the ability to store files. Incrementals use space-efficient rsync-style deltas, like rdiff-backup, except that forward (traditional) rather than reverse incrementals are used. The use of forward incrementals means that occasional full backups (with their associated large upload cost) must be made to reclaim storage space.

Brackup [6], like duplicity, also stores encrypted backups and needs no special server support. Brackup was specifically designed to work with Amazon S3, though it is not tied to it since the storage interface is generic. Each file backed up is stored separately on the server, so reclaiming space when deleting a snapshot is simple. Later versions may pack small files together for efficiency, though doing so will complicate reclaiming

storage space (as we show in this paper).

Box Backup [13] supports encrypted storage of backup data, rsync-style delta for incrementals, and good reclamation of space on the storage server, but to do so requires specialized software at the server.

Backup-as-a-service providers may offer a similar feature set: relatively-efficient backups, multiple snapshots, no need to repeatedly upload a full backup, and encryption. However, these services tie backups to a particular provider, and precise implementation details may be harder to come by.

Table 1 summarizes differences between some of the tools listed above for backup to networked storage. In relation to existing systems, Cumulus is most similar to duplicity (without the need to occasionally re-upload a new full backup), and brackup (with an improved scheme for incremental backups including rsync-style deltas, and much improved reclamation of storage space).

3 Design

In this section we present the design of our approach for making backups to a thin cloud remote storage service.

3.1 Storage Server Interface

We assume only a very narrow interface between a client generating a backup and a server responsible for storing the backup, consisting of four operations:

Get: Given a pathname, retrieve the contents of a file from the server.

Put: Store a complete file on the server with the given pathname.

List: Get the names of files stored on the server.

Delete: Remove the given file from the server, reclaiming its space.

Note that all of these operations operate on entire files; we do not depend upon the ability to read or write arbitrary byte ranges within a file. Support for reading and setting file attributes, such as permissions and timestamps, is neither required nor used. The interface is simple enough that it can be implemented on top of any number of protocols: FTP, SFTP, WebDAV, S3, or nearly any network file system.

One consequence of a narrow interface is that it becomes natural to adopt a *write-once storage model*, in which a file is never modified after it is first stored except perhaps to delete it to recover space. We do not depend on server interfaces for modifying parts of a file, or copying or renaming files, so the only way to accomplish these operations is to completely re-upload any changed files. Given this, there is little reason not to simply direct all writes to new files (deleting old files later). The write-once model also provides convenient failure guarantees: since files are never modified in place, a failed backup run cannot corrupt old backups. At worst, it will leave a partially-written backup which can be deleted by a garbage-collection process.

The write-once storage model also implies that, since creating a new backup does not modify the files that make up the previous backup, keeping snapshots at multiple points in time simply amounts to not deleting the files that make up old snapshots.

3.2 Storage Segments

When storing a snapshot, Cumulus will often group data from many smaller files together into larger units called *segments*. Segments become the unit of storage on the server, with each segment stored as a single file. There are a number of reasons for consolidating data into segments.

Avoid inefficiencies associated with many small files. The storage server may not store many small files as efficiently as a smaller number of large files. Some file systems will often round up file sizes to be some multiple of a block size, so storage of small files is less efficient. A storage server might also expose a preference for larger files to the user. Amazon S3, for example, has both a per-request and a per-byte cost when storing a file. For files of 100 kB or smaller, the per-file cost dominates; it is most economical to store data in units larger than 100 kB.

Numerous studies of file system metadata have shown that small files predominate. Agrawal *et al.* find that the median file size in their studies has remained at about 4 kB [1]. We find a similar pattern in our own evaluations. With half of files 4 kB or less, it is desirable to treat small files as a common case in the backup system. Even average file size, though larger than median file sizes, may still be relatively small: 20–150 kB in our data sets.

Avoid high per-file costs in network protocols. Unless the protocol used to communicate with the storage server is pipelined to allow multiple outstanding **put** requests, or multiple connections are opened in parallel, the file transfer bandwidth will be limited to one file per network round-trip. Over a high-latency network, this could significantly slow down transfers. Grouping data together into larger files reduces this overhead. We study this effect in more detail in Section 5.4.4.

Take advantage of inter-file redundancy with segment compression. Compression may be used to reduce the storage needed for backups. If all files are compressed individually, however, especially in the case that there are many small files, significant opportunities for compression may be lost. By grouping related files together into segments, compression can eliminate redundancy

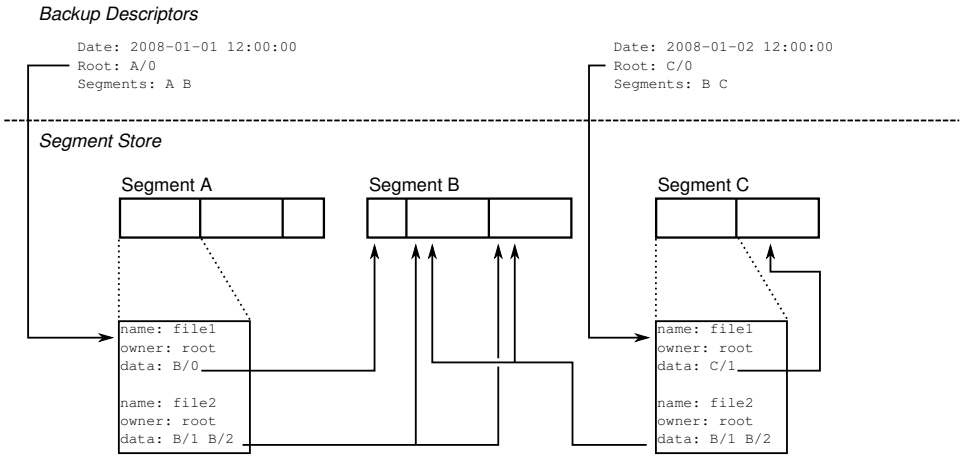


Figure 1: Simplified schematic of the basic format for storing snapshots on a storage server. Two snapshots are shown, taken on successive days. Each snapshot contains two files. `file1` changes between the two snapshots, but the data for `file2` is shared between the snapshots. For simplicity in this figure, segments are given letters as names instead of the 128-bit UUIDs used in practice.

between files as well as within files. Section 5.4.2 evaluates the benefits of grouping files together before compression.

Hide details of the file size distribution from the server. Data may be encrypted before transferring it to the storage server to provide privacy. However, if each file on the client is stored as a file on the server, then even with encryption the server can still learn many details about the number and (approximate) size of the files on the client. Grouping files into segments masks much of this information.

Represent sub-file changes efficiently in incremental backups. Even if the above reasons for choosing to group data into segments did not apply to data at a file level — if the average file size was large enough to make it acceptable to store each file separately — they apply once sub-file incrementals are introduced. Isolated changes in large files can be represented efficiently by dividing each file into small (few kilobyte) chunks which can be re-used or not to represent the modified file. However, these chunks must either be stored separately (producing a very large number of small files), or grouped together for storage purposes — acting once again like the segment model.

3.3 Snapshot Format

Figure 1 illustrates the basic format used to store backup snapshots. A snapshot logically consists of two parts: a *metadata log* which lists all the files backed up, and the file data itself. Both metadata and data are broken apart into blocks, or *objects*, and these objects are then packed together into *segments*, compressed as a unit and optionally encrypted, and stored on the server. Each segment has a unique name — we use a randomly generated 128-bit UUID so that segment names can be assigned independently. Objects are numbered sequentially within a segment.

Segments are internally structured as a TAR file, with each file in the archive corresponding to an object in the segment. Compression and encryption are provided by filtering the raw segment data through `gzip`, `bzip2`, `gpg`, or other similar external tools.

A snapshot can be decoded by traversing a tree (or, in the case of sharing, a DAG) of objects. The root object in the tree is the start of the metadata log. This root may contain some file metadata directly, or it may contain pointers to objects containing other fragments of the metadata log. This support for pointers between portions of the metadata log allows flexibility in how the metadata log is written: it may be written incrementally, may be spread across multiple segments, and portions of the metadata log may even be shared between different snapshots (useful for compactly encoding metadata for files that do not change). Objects containing the actual file contents appear as leaves in the tree.

The metadata log entry for each individual file specifies properties such as modification time, ownership, and file permissions, and can be extended to include additional information if needed. It includes a cryptographic checksum[1] so that file integrity can be verified after a restore. Finally, it includes a list of pointers to objects containing the file data; concatenating the data in all of these objects reproduces the original file data. Metadata is stored in a text, not binary, format to make it more

[1] SHA-1 is currently used, but alternate algorithms may be easily substituted.

transparent. Compression applied to the segments containing the metadata, however, makes the format quite space-efficient.

The one piece of data in each snapshot not stored in a segment is a *snapshot descriptor*, which includes a timestamp for the snapshot and a pointer to the root object of the snapshot.

Starting with the root object stored in the snapshot descriptor and traversing all pointers found, a list of all segments required by the snapshot can be constructed. Since segments may be shared between multiple snapshots, it would be incorrect to delete each of the segments used by a snapshot when deleting the snapshot itself. Instead, a garbage collection process identifies segments still referenced by current snapshots, then deletes unreferenced segments. To simplify this process, each snapshot descriptor includes, though it is redundant, a list of segments on which it depends. This list of segments can also be used if it is necessary to restore the contents of a snapshot, since it allows all needed segments to be downloaded before even beginning to parse the snapshot metadata log.

3.4 Sub-File Incrementals

If only a small portion of a large file changes between snapshots, only the changed portion of the file should be stored. The Cumulus format supports this in two ways. First, the metadata for each backed-up file includes a list objects that make up the file contents. If some of these blocks remain unchanged in the new file version, pointers to these old objects may be used. This may occur if there are a few in-place writes to a file, or new data is appended at the end (at most the last block of the old file will change). Hashes of block contents are used to detect when a block can be re-used. These object hashes also allows for coarse-grained data de-duplication during backup — if the data needed for a new file matches the hash of an old block, that block is re-used.

Second, the Cumulus format allows references to an object to specify a byte range — thus, an object can be re-used even if it overlaps a changed part of a file by using a byte-range to select the unchanged part, and writing the new data out to a new object. We discuss how our implementation of Cumulus identifies unchanged data in Section 4.3.

3.5 Segment Cleaning

When old snapshots are no longer needed, space on the storage server may be reclaimed by deleting the root snapshot descriptors for those snapshots, then deleting any segments which are no longer reachable. It may be, however, that some segments only contain a small fraction of useful data — the remainder of these segments, data from deleted snapshots, is now wasted space. This problem is similar to the problem of reclaiming space in the Log-Structured File System (LFS) [10].

There are differences between our situation and that of LFS, however. Unlike LFS, which must only track the current state of the filesystem, we keep references to the state of the filesystem at multiple points in time. LFS has a fixed target for total space used: the size of the underlying disk, with no real benefit to staying significantly below this limit. We have no hard limits, but an overall preference to avoid wasted space where possible.

There are two approaches that can be taken in the segment-cleaning process. The first, *in-place cleaning*, identifies unused data in a segment and rewrites the segment to eliminate the unneeded data. In-place cleaning can recover wasted space in a snapshot at any time.

This mode of operation has several disadvantages, however. First, it violates the write-once storage model, in that the data on which a snapshot depends is changed after the snapshot is written, which can make reasoning about backups more complex. It requires detailed tracking of which data is used by each snapshot, at the granularity of objects, not just segments, to ensure that cleaning does not delete an object still in use, and thus complicates the garbage-collection process. Second, repacking data from an old snapshot can require the data to be downloaded from the storage server, processed locally, and re-uploaded since there may no longer be local copy of the data. This step may generate extra network traffic during a backup. Third, if data is encrypted before sending it to the storage server, in-place cleaning will require decrypting the backup data. If public-key encryption is used, the backup process only needs access to the public key, but cleaning will require access to the private key, and so may require manual intervention (if the private key is protected).

The alternative to in-place cleaning is to never modify segments in old snapshots, through a multi-step process:

1. Identify segments which are poorly-utilized.
2. Ensure that subsequently created snapshots do not reference the segments identified in the previous step. If necessary, upload copies of still-needed data in new segments.
3. As old snapshots are deleted, the segments identified in the first step can be deleted since all more recent snapshots will not reference them.

This cleaning process avoids the disadvantages listed above of in-place cleaning. However, it will not be as space-efficient as in-place cleaning. Dead space is not reclaimed until snapshots depending on the old segment are deleted. Additionally, during this time data is stored redundantly: old snapshots reference the data in the selected-for-cleaning segment, and new snapshots

reference the compacted version. We analyzed both approaches to cleaning in simulation, but found that the cost benefits of in-place cleaning were not large enough to outweigh its disadvantages, and so our Cumulus prototype does not clean in-place.

3.5.1 Selecting Segments for Cleaning

The simplest policy for selecting segments to clean is to set a minimum utilization threshold, α. We define utilization as the fraction of bytes within the segment which are referenced by a current snapshot. Any segment with a utilization below α will have its data repacked. So, $\alpha = 0.8$ will ensure that at least 80% of the bytes in segments are useful, and that at most 20% of the bytes within segments is wasted. Letting $\alpha = 0$ will disable segment cleaning altogether. The storage overhead can be bounded by $1/\alpha$, so values of α closer to 1 will tend to decrease the storage overhead of a single snapshot in isolation.

On the other hand, a high threshold for α, which will clean aggressively, will transfer more data than necessary. In the worst case, each segment will be cleaned just at the threshold. In aggregate, over all snapshots uploaded, the network overhead can be bounded by $1/(1-\alpha)$.

3.6 Restoring from Backup

Restoring data from previous backups may take several forms:

- Perform a *complete restore*: restore data from all files on a given date.
- Perform a *partial restore*: restore data from a specified file, or a specified set of files, on a given date.
- Report on a file history: list all the available versions (different dates) of a specified file, or set of files.

Cumulus is primarily optimized for the first form of restore — recovering all files, such as in the event of the total loss of the original data. In this case, the restore process will look up the root snapshot descriptor at the date to restore, then download all segments referenced by that snapshot. Since segment cleaning seeks to avoid leaving much wasted space in the segments, the total amount of data to be downloaded will not be much more than needed.

For partial restores, Cumulus needs to download those segments that contain metadata for the snapshot, to locate the files requested, and then locate each of the segments containing file data. It is possible that the quantity of data downloaded significantly exceeds the size of the data to restore, since the selected files may be scattered across many segments. For example, if a request is made to restore the contents of a directory as of a given date, and files in that directory had been added gradually, those files are likely scattered across segments initially created on many different days.

Cumulus is not optimized for tracking the history of individual files — the only way to determine the list of changes to a file or set of files is to download and process the metadata logs for all snapshots.

One important issue in restoring from backups is that the order data appears in the metadata log (filesystem traversal order) is often different from the order data appears segments (roughly chronological by modification time). Since reading data from segments may be expensive (possibly involving downloading and decompression/decryption), an initial scan of the metadata log is made to read all file metadata, then file data is restored in segment order.

3.7 Limitations

Cumulus is not designed to replace all existing backup systems; there are situations in which other systems will do a better job.

First and perhaps most obviously, Cumulus is not designed for environments where backup is done to tape, since it assumes random access to the segments within a backup snapshot.

The approach embodied by Cumulus is for the client making a backup to do most of the work, and leave the backup itself almost entirely opaque to the server. This approach makes Cumulus portable to nearly any type of storage server, but also means that it cannot take advantage of any special capabilities of the server. In cases where there are resources to create a specialized backup server — for example, centralized backup for a department — the specialized server may perform better. Certainly, a server that understands the backup format can avoid the difficulties that Cumulus faces with segment cleaning, since the server could simply reclaim any wasted space without any help needed from the client. Another area in which a smarter server can provide benefits is in restoring a small collection of files (rather than an entire snapshot), or fetching information about changes to that set of files.

Cumulus does not offer coordination between multiple backup clients. An intelligent backup server could perform tasks such as data de-duplication across all backup clients. Cumulus, as designed, cannot naturally support this.

The design of Cumulus is predicated on the fact that backing up each file on the client to a separate file on the server may introduce too much overhead, and so Cumulus groups data together into segments. However, if it is known that the storage server and network protocol

can efficiently deal with small files, then grouping data into segments adds unnecessary complexity and overhead. Other disk-to-disk backup programs may be a better match in this case.

4 Implementation

We discuss details of the implementation of the Cumulus prototype in this section. Our implementation is relatively compact: only slightly over 3000 lines of C++ source code (as measured by SLOCCount) implementing the core backup functionality, along with another roughly 700 lines of Python for tasks such as restores, segment cleaning, and statistics gathering.

4.1 Local Client State

Each client stores on local disk information about recent backups, primarily so that it can detect which files have changed and properly reuse data from previous snapshots. This information could be kept on the storage server, however storing it locally reduces network bandwidth and improves access times. None of this information is needed to recover data from a backup, so its loss is not catastrophic. However, this local state does enable various performance optimizations during backups.

The client's local state is divided into two parts: a local copy of the metadata log, and an SQLite database [12] containing all other needed information.

Note that the information stored locally and its format are not specified by the Cumulus backup format. It is entirely possible to create a new and very different implementation which nonetheless produces backups conforming to the structure described in Section 3.3 and readable by our Cumulus prototype.

4.1.1 Local Metadata Cache (Statcache)

The client saves a local copy of the metadata log from the most recent snapshot. This data is also referred to as the statcache file, since in large part it caches the results of the `stat` system call on each file from the previous backup. The statcache file serves multiple purposes. First, it greatly speeds up the backup process. If file metadata (size, modification time, inode number) for the current version of a file matches the metadata in the statcache file, then the file is unchanged and it is not necessary to read the full contents. The file metadata from the statcache file (including pointers to the file data from the previous backup) can simply be re-used.

Second, the statcache file is instrumental in delta-encoding metadata logs between snapshots. Along with file metadata, the statcache file records the location of the metadata for each file within the last backup. If a file does not change, this information can be used to emit a reference to the old metadata log entry.

Since Cumulus will iterate through the entire contents of the statcache file on a backup anyway, the statcache file is simply saved as a flat text file.

4.1.2 Local Client Database

All client state stored locally that is not part of the statcache is kept in an SQLite database file. SQLite was chosen since the database engine is embedded directly into the Cumulus executable, so there are no external dependencies when running Cumulus. But it still provides the convenience of a real database — it allows for efficient random access to information, automatically maintains indices over the data, and allowed for more rapid prototyping of Cumulus features. Additionally, SQLite is transactional so an interrupted backup will not corrupt the local database.

Cumulus keeps a record of recent snapshots, and all segments and objects stored in them, in the local database. The table of objects includes an index by the content hash so that duplicate data (at the level of an object) can be identified, and only one copy written out.

4.2 Segment Cleaning

Segment cleaning heuristics are not implemented directly as part of the Cumulus backup program (in C++). Instead, cleaning is done by the Cumulus utility program, implemented in Python. Communication between these two programs is mediated by the local database.

When each snapshot is written, the Cumulus backup program records in the local database a summary of all segments used by that snapshot and the fraction of the data in each segment that is actually referenced. The Cumulus utility program uses these summaries to identify segments which are poorly-utilized, and then marks the selected segments as "expired" in the local database. On subsequent backups, the Cumulus backup program will avoid referencing objects in expired segments. If such data would have been used, the data is instead written out as a new object.

The Cumulus utility program may additionally write out a hint with each object in an expired segment to direct how objects should be grouped together when they are rewritten. The Cumulus utility may sort expired objects based on age (the date at which an object was first written is recorded in the local database), group them in to buckets, and labels into the local database. When the Cumulus backup program runs, objects with different labels that must be rewritten are placed in separate segments; this gives a mechanism for grouping data by age when repacking segment data.

4.3 Sub-File Incrementals

As discussed in Section 3.4, the Cumulus backup format supports efficiently encoding differences between file

versions. Our implementation detects changes by dividing files into small *chunks* in a content-sensitive manner (using Rabin fingerprints) and identifying chunks that are common, as in the Low-Bandwidth File System [8].

When a file is first backed up, it is divided into blocks of about a megabyte in size which are stored individually in objects. In contrast, the chunks used for sub-file incrementals are quite a bit smaller: the target size is 4 kB (though variable, with a 2 kB minimum and 64 kB maximum). Before each megabyte block is stored, a set of chunk signatures are computed for it: the data is divided into non-overlapping chunks that cover the entire block (the last chunk may be short), and a (20-byte SHA-1 signature, 2-byte length) tuple is computed for each chunk. The list of chunk signatures for each object are stored in the local database. These signatures consume 22 bytes for every roughly 4 kB of original data, so the signatures are about 0.5% of the size of the data to back up.

Unlike LBFS, we do not create a global index of chunk hashes — to limit overhead, we do not attempt to find common data between different files. When a file changes, we instead limit the search for unmodified data to the chunks in the previous version of the file. Specifically, Cumulus reads from the statcache file the previous list of objects in the file, and loads the chunk signatures for each of those objects into memory. Finally, chunk signatures for the new file data are computed. Any chunks which match an old chunk are written as a reference to the old data; new chunks are written to new objects (and chunk signatures for those objects stored in the local database too).

In the common case where a consecutive sequence of chunks in the new file matches a consecutive sequence in the old, a single combined reference is written to the metadata log instead of listing each chunk separately.

4.4 Segment Filtering and Storage

The core Cumulus backup implementation is only capable of writing segments as uncompressed TAR files to local disk. Additional functionality is implemented by calling out to external scripts.

When performing a backup, all segment data may be filtered through a specified command before writing it. Specifying a program such as `gzip` can provide compression, or `gpg` for encryption.

Similarly, network protocols are implemented by calling out to external scripts. During a backup, segments are first written to a temporary directory locally. Once a segment is completely written locally, an external script is called to transfer the segment to the remote storage server. One upload is permitted at a time, but uploads are allowed to proceed in parallel with the main backup process for better throughput. A limit is placed on the number of completed segments not yet uploaded — progress in backing up is throttled so an unbounded amount of temporary space is not needed. Upload scripts may be quite simple; a script for uploading to Amazon S3 is merely 12 lines long in Python using the boto [3] library.

4.5 Snapshot Restores

The Cumulus utility tool written in Python implements complete restore functionality. This tool is able to automatically decompress and extract objects from segments, and can efficiently extract just a subset of files from a snapshot.

5 Evaluation

We use both trace-based simulation and a prototype implementation to evaluate the use of thin cloud services for remote backup. Our goal is to answer three high-level sets of questions:

- What is the penalty of using a thin cloud service with a very simple storage interface compared to a more sophisticated service?
- What are the monetary costs for using remote backup for two typical usage scenarios? How should remote backup strategies adapt to minimize monetary costs as the ratio of network and storage prices varies?
- How does our prototype implementation compare with other backup systems? What are the additional benefits (e.g., compression) and overheads (e.g., metadata) of an implementation not captured in simulation? What is the performance of using an online service like Amazon S3 for backup?

The following evaluation sections answer these questions, beginning with a description of the trace workloads we use as inputs.

5.1 Trace Workloads

We use two traces as workloads to drive our evaluations. A **fileserver** trace tracks all files stored on our research group fileserver, and models the use of a cloud service for remote backup in an enterprise setting. A **user** trace is extracted from the Cumulus backups of the home directory of one of the author's personal computers, and models the use of remote backup in a home setting. The traces contain a daily record of the metadata of all files in each setting, including a hash of the file contents. The user trace further includes all data necessary to construct the contents of all files at any point in time in the trace, necessary for evaluating the effects of compression and sub-file incrementals in an implementation. Table 2 summarizes the key statistics of each trace.

	Fileserver	User
Duration (days)	157	223
Entries	26673083	122007
Files	24344167	116426
File Sizes		
Median	0.996 kB	4.4 kB
Average	153 kB	21.4 kB
Maximum	54.1 GB	169 MB
Total	3.47 TB	2.37 GB
Update Rates		
New data/day	9.50 GB	10.3 MB
Changed data/day	805 MB	29.9 MB
Total data/day	10.3 GB	40.2 MB

Table 2: Key statistics of the two traces used in evaluations. File counts and file size statistics are for the last day in the trace. "Entries" counts the number of files, directories, symlinks, etc. "Files" counts the subset of entries that refer to regular files.

5.2 Remote Backup to a Thin Cloud

First we explore the overhead of using remote backup to a thin cloud service that has only a simple storage interface. We compare this thin service model to an "optimal" model representing more sophisticated backup systems.

We use simulation for these experiments, and start by describing our simulator. We then define our optimal baseline model and evaluate the overhead of using a simple interface relative to a more sophisticated system.

5.2.1 Cumulus Simulator

The Cumulus simulator models the process of backing up collections of files to a remote backup service. It uses traces of daily records of file metadata to perform backups by determining which files have changed, aggregating changed file data into segments for storage on a remote service, and cleaning expired data as described in Section 3.

The simulator tracks three overheads associated with performing backups. It tracks storage overhead, the total number of bytes to store a set of snapshots, computed as the sum of the size of each segment needed. Storage overhead includes both actual file data as well as wasted space within segments. It tracks network overhead, the total data that must be transferred over the network to accomplish a backup. On graphs, we show this overhead as a cumulative value: the total data transferred from the beginning of the simulation until the given day. Since remote backup services have per-file charges, the simulator also tracks segment overhead as the number of segments created during the process of making backups.

The simulator also models two snapshot scenarios. In the *single snapshot* scenario, the simulator maintains only one snapshot remotely and it deletes all previous snapshots. In the *multiple snapshot* scenario, the simulator retains snapshots according to a pre-determined backup schedule. In our experiments, we keep the most recent seven daily snapshots, with additional weekly snapshots retained going back farther in time so that a total of 12 snapshots are kept.

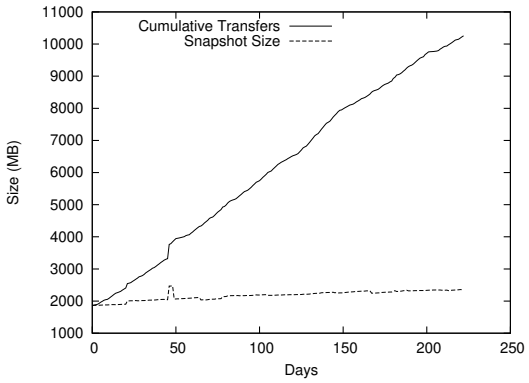


Figure 2: Storage and network overhead for an optimal backup of the files from the user trace.

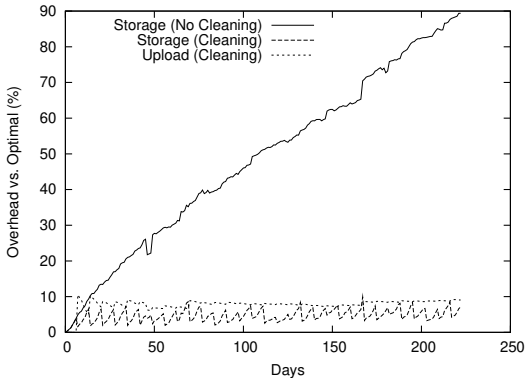
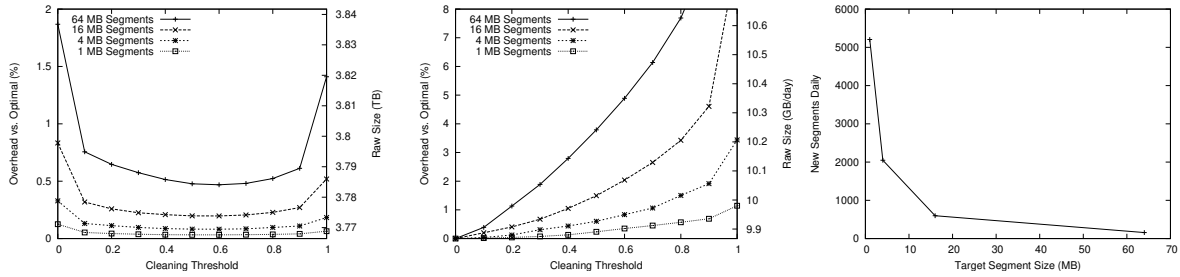


Figure 3: Overheads with and without cleaning; segments are cleaned at 60% utilization. Only storage overheads are shown for the no cleaning case since there is no network transfer overhead without cleaning.

The simulator makes some simplifying assumptions that we explore later when evaluating our implementation. The simulator detects changes to files in the traces using a per-file hash. Thus, the simulator cannot detect changes to only a portion of a file, and assumes that the entire file is changed. The simulator also does not model compression or metadata. We account for sub-file changes, compression, and metadata overhead when evaluating the prototype in Section 5.4.

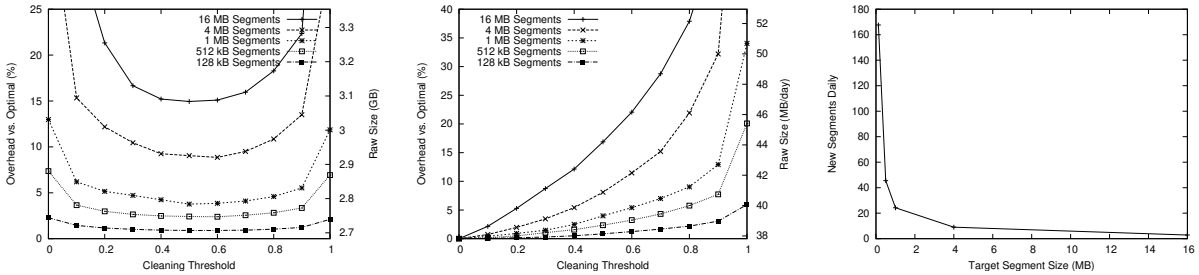
5.2.2 Optimal Baseline

A simple storage interface for remote backup can incur an overhead penalty relative to more sophisticated ap-



(a) Average daily storage (b) Average daily upload (c) Average segments per day

Figure 4: Overheads for backups in the fileserver trace.



(a) Average daily storage (b) Average daily upload (c) Average segments per day

Figure 5: Overheads for backups in the user trace.

proaches. To quantify the overhead of this approach, we use an idealized *optimal backup* as a basis of comparison.

For our simulations, the optimal backup is one in which no more data is stored or transferred over the network than is needed. Since simulation is done at a file granularity, the optimal backup will transfer the entire contents of a file if any part changes (results from the prototype implementation in Section 5.4.1 indicates that this simplifying assumption is reasonable). Optimal backup will, however, perform data de-duplication at a file level, storing only one copy if multiple files have the same hash value. In the optimal backup, no space is lost to fragmentation when deleting old snapshots. Cumulus could achieve this optimal performance in this simulation by storing each file in a separate segment—that is, to never bundle files together into larger segments. As discussed in Section 3.2 and as our simulation results show, though, there are good reasons to use segments with sizes much larger than the average file.

As an example of these costs and how we measure them, Figure 2 shows the optimal storage and upload overheads for daily backups of the 223 days of the user trace. In this simulation, only a single snapshot is retained each day. Storage grows slowly in proportion to the amount of data in a snapshot, and the cumulative network transfer grows linearly over time.

Figure 3 shows the results of two simulations of Cumulus backing up the same data. The graph shows the overheads relative to optimal backup; a backup as good as optimal would have 0% relative overhead. These results clearly demonstrate the need for cleaning when using a simple storage interface for backup. When segments are not cleaned (only deleting segments that by chance happen to be entirely no longer needed), wasted storage space grows quickly with time — by the end of the simulation at day 223, the size of a snapshot is more than double the required size. In contrast, when segments are marked for cleaning at the 60% utilization threshold, storage overhead quickly stabilizes below 10%. The overhead in extra network transfers is similarly modest.

5.2.3 Cleaning Policies

Cleaning is clearly necessary for efficient backup, but it is also parameterized by two metrics: the size of the segments used for aggregation, transfer, and storage (Section 3.2), and the threshold at which segments will be cleaned (Section 3.5.1). In our next set of experiments, we explore the parameter space to quantify the impact of these two metrics on backup performance.

Figures 4 and 5 show the simulated overheads of backup with Cumulus using the fileserver and user traces, respectively. The figures show both relative overheads to optimal backup (left y-axis) as well as the absolute overheads (right y-axis). We use the backup policy of multiple daily and weekly snapshots as described in Section 5.2.1. The figures show cleaning overhead for

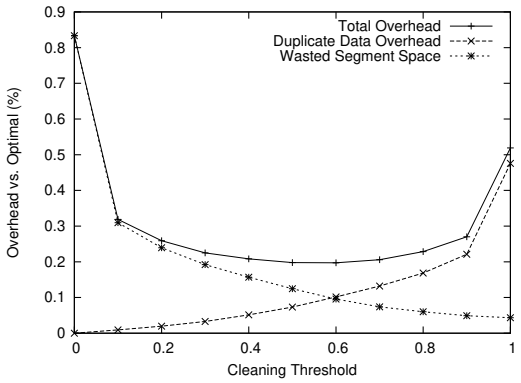


Figure 6: Detailed breakdown of storage overhead when using a 16 MB segment size for the fileserver workload.

a range of cleaning thresholds and segment sizes. Each figure has three graphs corresponding to the three overheads of remote backup to an online service. *Average daily storage* shows the average storage requirements per day over the duration of the simulation; this is the total storage needed for tracking multiple backup snapshots, not just the size of a single snapshot. Similarly, *average daily upload* is the average of the data transferred each day of the simulation, excluding the first. The first day is excluded since any backup approach must transfer the entire initial filesystem. Finally, *average segments per day* tracks the number of new segments uploaded each day to account for per-file upload and storage costs.

Storage and upload overheads improve with decreasing segment size, but at small segment sizes ($<$ 1 MB) backups require very large numbers of segments and limit the benefits of aggregating file data (Section 3.2). As expected, increasing the cleaning threshold increases the network upload overhead. Storage overhead with multiple snapshots, however, has an optimum cleaning threshold value. Increasing the threshold initially decreases storage overhead, but high thresholds increase it again; we explore the this behavior further below.

Both the fileserver and user workloads exhibit similar sensitivities to cleaning thresholds and segment sizes. The user workload has higher overheads relative to optimal due to more potential benefit from sub-file incrementals, but overall the overhead penalties remain low.

Figures 4(a) and 5(a) show that there is a cleaning threshold that minimizes storage overheads. Increasing the cleaning threshold intuitively reduces storage overhead relative to optimal since the more aggressive cleaning at higher thresholds will delete wasted space in segments and thereby reduce storage requirements. Figure 6 explains why storage overhead increases again at higher cleaning thresholds.

It shows three curves, the 16 MB segment size curve from Figure 4(a) and two curves that decompose the storage overhead into individual components (Section 3.5). One is overhead due to duplicate copies of data stored over time in the cleaning process; cleaning at lower thresholds reduces this component. The other is due to wasted space in segments which have not been cleaned; cleaning at higher thresholds reduces this component. A cleaning threshold near the middle, however, minimizes the sum of both of these overheads.

Fileserver	Amount	Cost
Initial upload	3563 GB	$356.30
Upload	303 GB/month	$30.30/month
Storage	3858 GB	$578.70/month
User	Amount	Cost
Initial upload	1.82 GB	$0.27
Upload	1.11 GB/month	$0.11/month
Storage	2.68 GB	$0.40/month

Table 3: Costs for backups in US dollars, if performed optimally, for the fileserver and user traces using current prices for Amazon S3.

5.3 Paying for Remote Backup

The evaluation in the previous section measured the overhead of Cumulus in terms of storage, network, and segment resource usage. Remote backup as a service, however, comes at a price. In this section, we calculate monetary costs for our two workload models, evaluate cleaning threshold and segment size in terms of costs instead of resource usage, and explore how cleaning should adapt to minimize costs as the ratio of network and storage prices varies.

We use the prices for the Amazon S3 service as an initial point in the pricing space. As of May 2008, these prices are:

Storage: $0.15 per GB $\cdot$ month
Upload: $0.10 per GB
Segment: $0.01 per 1000 files uploaded

With this pricing model, the *segment* cost for uploading an empty file is equivalent to the *upload* cost for uploading approximately 100 kB of data, i.e., when uploading 100 kB files, half of the cost is for the bandwidth and half for the upload request itself. As the file size increases, the per-request component becomes an increasingly smaller part of the total cost.

Neglecting for the moment the segment upload costs, Table 3 shows the monthly storage and upload costs for each of the two traces. Storage costs dominate ongoing costs. They account for about 95% and 78% of the monthly costs for the fileserver and user traces, respectively. Thus, changes to the storage efficiency will have a more substantial effect on total cost than changes in

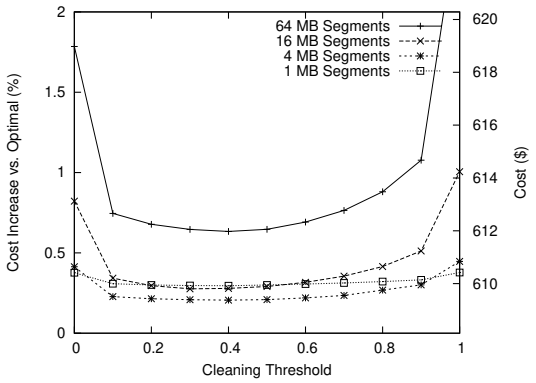


Figure 7: Costs in US dollars for backups in the fileserver assuming Amazon S3 prices. Costs for the user trace differ in absolute values but are qualitatively similar.

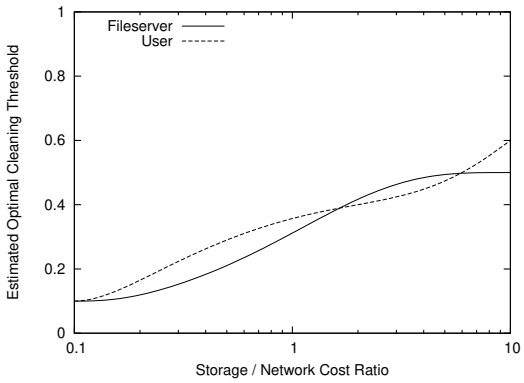


Figure 8: How the optimal threshold for cleaning changes as the relative cost of storage vs. network varies.

bandwidth efficiency. We also note that the absolute costs for the home backup scenario are very low, indicating that Amazon's pricing model is potentially quite reasonable for consumers: even for home users with an order of magnitude more data to backup, yearly ongoing costs are roughly US$50.

Whereas Figure 4 explored the parameter space of cleaning thresholds and segment sizes in terms of resource overhead, Figure 7 shows results in terms of overall cost for backing up the fileserver trace. These results show that using a simple storage interface for remote backup also incurs very low additional monetary cost than optimal backup, from 0.5–2% for the fileserver trace depending on the parameters, and as low as about 5% in the user trace. When evaluated in terms of monetary costs, the choices of cleaning parameters change compared to the parameters in terms of resource usage. The cleaning threshold providing the minimum cost is smaller and less aggressive (threshold $= 0.4$) than in terms of resource usage (threshold $= 0.6$). Furthermore, in contrast to resource usage, decreasing segment size does not always decrease overall cost. At some point — in this case between 1–4 MB — decreasing segment size increases overall cost due to the per-file pricing. The results for the user workload, although not shown, are qualitatively similar.

The pricing model of Amazon S3 is only one point in the pricing space. As a final cost experiment, we explore how cleaning should adapt to changes in the relative price of storage versus network. Figure 8 shows the optimal cleaning threshold for the fileserver and user workloads as a function of the ratio of storage to network cost. The storage to network ratio measures the relative cost of storing a gigabyte of data for a month and uploading a gigabyte of data. Amazon S3 has a ratio of 1.5. In general, as the cost of storage increases, it becomes advantageous to clean more aggressively (the optimal cleaning threshold increases). The ideal threshold stabilizes around 0.5–0.6 when storage is at least ten times more expensive than network upload, since cleaning too aggressively will tend to increase storage costs.

5.4 Prototype Evaluation

In our final set of experiments, we compare the overhead of the Cumulus prototype implementation with other backup systems. We also evaluate the sensitivity of compression on segment size and the overhead of metadata in the implementation, and the time it takes to upload data to a remote service like Amazon S3.

5.4.1 System Comparisons

First we provide results from running our Cumulus prototype and compare with two existing backup tools. We use the complete file contents included in the user trace to accurately measure the beavhior of our full Cumulus prototype and other real backup systems. For each day in the first two months of the user trace, we extract a full snapshot of all files, then back up these files with several backup tools:

Incremental tar: Backups using GNU tar and the `--listed-incremental` option. This option produces incremental backups at a file-level granularity.

Duplicity: Backups using duplicity, with encryption disabled. These backups are much like the incremental tar backups, except that the rsync algorithm efficiently captures small changes to files.

Cumulus: Our full prototype implementation.

In all tests, data is compressed using `gzip` at the maximum compression level.

Figure 9 shows the storage costs for the three systems, taking a single full backup at the start and then incrementals each following day; in the case of Cumulus, all snapshots are retained. The initial cost of a full backup in all three systems is comparable, since all are effectively storing a copy of each file compressed with gzip. Cumulus

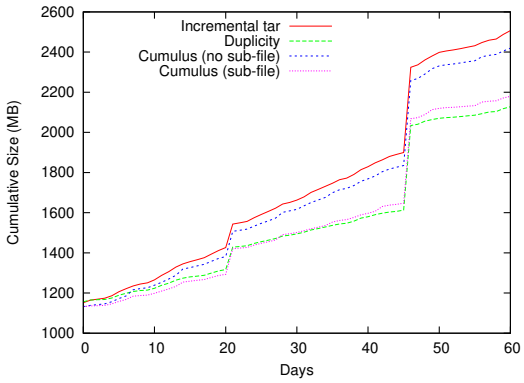


Figure 9: Cumulative storage costs for actual runs of multiple backup systems.

uses slightly less space since it performs deduplication at a coarse level (fixed 1-MB blocks), and identifies a small amount of duplicate data. Second, the rate of growth of Cumulus and incremental tar backups are comparable. This result is not too surprising since, in this mode of operation, on each successive day Cumulus backs up most data in all changed files. Duplicity performs better than the other two schemes by taking advantage of sub-file incrementals. Finally, when sub-file incremental backups are enabled in Cumulus, its daily upload rate falls so that it almost exactly matches that of duplicity.

Over the course of the two-month simulation, tar uploads 1355 MB in incrementals and duplicity uploads 971 MB. For comparison, Cumulus uploads 1287 MB without sub-file incrementals, and 1048 MB with sub-file incrementals. Since duplicity is assumed close to optimal in generating efficient incrementals, the network upload overhead of Cumulus is just under 8%.

Duplicity is unable to delete old snapshots as long as more recent incrementals depend on them. To allow daily incrementals to be deleted, we configured duplicity to create weekly incrementals relative to the previously weekly backup, and daily incrementals during the week. With this configuration, duplicity uploads 1105 MB, more than 5% more than Cumulus. Eventually, to recover the space from the first full backup, there must be another full backup—at which point the overhead for duplicity rises dramatically (another 1200 MB to transfer).

In summary, the Cumulus prototype further shows that using a thin cloud service with a simple storage interface for remote backup suffers little penalty relative to more sophisticated backup system configurations.

5.4.2 Segment Compression

To isolate the effectiveness of compression at reducing the size of the data to back up, particularly as a function of segment size and related settings, we used as a sample the full data contained in the first day of the user trace: the uncompressed size is 1916 MB, the compressed tar size is 1152 MB (factor of 1.66), and files individually compressed total 1219 MB (1.57×), 5.8% larger than whole-snapshot compression.

Varying the segment size used to aggregate data for backup, we calculated the average compression ratios using `gzip` and `bzip2`. Larger segments produce better compression ratios. `gzip` compression stabilizes at a 2.7 ratio at a segment size of 300 kB, and `bzip2` stabilizes at 3.2 at 1–2 MB.

5.4.3 Metadata

The Cumulus prototype stores metadata for each file in a backup snapshot in a text format, but after compression the format is still quite efficient. In the full tests on the user trace, the metadata for a full backup takes roughly 46 bytes per item backed up. Since most items include a 20-byte hash value which is unlikely to be compressible, the non-checksum components of the metadata average under 30 bytes per file.

Metadata logs can be stored incrementally: new snapshots can reference the portions of old metadata logs that are not modified. In the full user trace replay, a full metadata log was written to a snapshot weekly, but on days where only differences were written out, the average metadata log delta was under 2% of the size of a full metadata log. Overall, across all the snapshots taken, the (compressed) data written out for file metadata was approximately 5% of the total size of the (compressed) file data itself.

5.4.4 Upload Time

As a final experiment, we consider the time to upload to a remote storage service. Our Cumulus prototype is capable of uploading snapshot data directly to Amazon S3. To simplify matters, we evaluate upload time in isolation, rather than as part of a full backup, to provide a more controlled environment. Cumulus uses the boto [3] Python library to interface with S3.

Figure 10 shows the results of our measurements of upload performance to Amazon S3. As these results are from one experiment from a single computer (from a university campus network), they should not be taken as a good measure of the overall performance of S3. However, they do illustrate a few general features relevant to Cumulus. Upload rates for large (about a megabyte or larger) files approaches a speed of about 800 kB/s, but the upload rates for small files are significantly smaller. This behavior is expected: uploads are accomplished using an HTTP PUT request; since these requests cannot be pipelined, there must be at least one network round-trip for each file uploaded. The upload rates were consistent with a latency of around 100 ms per upload.

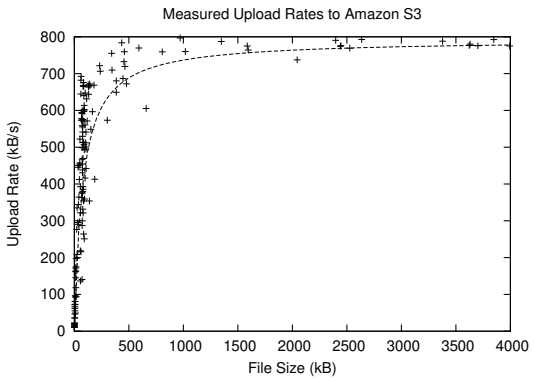


Figure 10: Measured upload rates for Amazon S3 as a function of file size. The measurements are fit to a curve where each upload consists of a fixed delay (for connection establishment and receiving a status code after the transfer) along with a fixed-rate upload.

With these rates, files one megabyte or larger are uploaded at effectively maximum speed. Other protocols might support pipelining upload requests, and thus support better uploads of many small files. However, by using moderately-sized segments, it is possible to get good performance out of any protocol, pipelined or not.

For perspective, assuming the maximum transfer rates above, ongoing backups for the fileserver and user workloads will take on average 3.75 hours and under a minute, respectively. Overheads from cleaning will increase these times, but since network overheads from cleaning are generally small, these upload times will not change by much. For these two workloads, backup times are very reasonable for daily snapshots.

6 Conclusions

It is fairly clear that the market for Internet-hosted backup service is growing. However, it remains unclear what form of this service will dominate. On one hand, it is in the natural interest of service providers to package backup as an integrated service since that will both create a "stickier" relationship with the customer and allow higher fees to be charged as a result. On the other hand, given our results, the customer's interest may be maximized via an open market for commodity storage services (ala S3), increasing competition due to the low barrier to switching providers, and thus driving down prices. Indeed, even today integrated backup providers charge between $5 and $10 per month per user while the S3 charges for backing up our test user using the Cumulus system was only $0.51 per month.[2]

[2]For example, Symantec's Protection Network charges 9.99 per month for 10GB of storage and EMC's MozyPro service costs 3.95+0.50/GB per month per desktop.

Moreover, a thin-cloud approach to backup allows one to easily hedge against provider failures by backing up to multiple providers. This may be particularly critical for guarding against business risk — a lesson that has been learned the hard way by customers whose hosting companies have gone out of business. Providing the same hedge using the integrated approach would require running multiple backup systems in parallel on each desktop or server, incurring redundant overheads (e.g., scanning, compression, etc.) that will only increase as disk capacities grow.

Finally, while this paper has focused on an admittedly simple application, we believe it identifies a key research agenda influencing the future of "cloud computing". The fundamental question is whether one can build a competitive product economy around a cloud of abstract commodity resources or if there underlying technical reasons that ultimately favor an integrated service-oriented infrastructure.

References

[1] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. *ACM Trans. Storage 3*, 3 (2007), 9.

[2] The advanced maryland automatic network disk archiver. `http://www.amanda.org/`.

[3] boto: Python interface to amazon web services. `http://code.google.com/p/boto/`.

[4] ESCOTO, B. rdiff-backup. `http://www.nongnu.org/rdiff-backup/`.

[5] ESCOTO, B., AND LOAFMAN, K. Duplicity. `http://duplicity.nongnu.org/`.

[6] FITZPATRICK, B. Brackup. `http://code.google.com/p/brackup/`, `http://brad.livejournal.com/tag/brackup`.

[7] IBM. Tivoli Storage Manager. `http://www.ibm.com/software/tivoli/products/storage-mgr/`.

[8] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev. 35*, 5 (2001), 174–187.

[9] PRESTON, W. C. *Backup & Recovery*. O'Reilly, 2006.

[10] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (1992), 26–52.

[11] rsnapshot. `http://www.rsnapshot.org/`.

[12] Sqlite. `http://www.sqlite.org/`.

[13] SUMMERS, B., AND WILSON, C. Box backup. `http://www.boxbackup.org/`.

[14] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Feb. 1999.