Cumulus: Filesystem Backup to the Cloud

MICHAEL VRABLE, STEFAN SAVAGE, and GEOFFREY M. VOELKER
University of California, San Diego

Cumulus is a system for efficiently implementing filesystem backups over the Internet, specifically designed under a *thin cloud* assumption—that the remote datacenter storing the backups does not provide any special backup services, but only a least-common-denominator storage interface. Cumulus aggregates data from small files for storage and uses LFS-inspired segment cleaning to maintain storage efficiency. While Cumulus can use virtually any storage service, we show its efficiency is comparable to integrated approaches.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management; E.5 [**Data**]: Files—*Backup/recovery*

General Terms: Design, Economics, Management, Measurement, Reliability

Additional Key Words and Phrases: Backup, cloud storage

ACM Reference Format:

Vrable, M., Savage, S., and Voelker, G. M. 2009. Cumulus: Filesystem backup to the cloud. ACM Trans. Storage 5, 4, Article 14 (December 2009), 28 pages.
DOI = 10.1145/1629080.1629084 http://doi.acm.org/10.1145/1629080.1629084

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

This article is an expanded version of the paper "Cumulus: Filesystem Backup to the Cloud" which appeared in the *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*.

This work was supported in part by the National Science Foundation grant CNS-0433668 and the UCSD Center for Networked Systems. M. Vrable was further supported in part by a National Science Foundation Graduate Research Fellowship.

Author's address: M. Vrable, University of California at San Diego, Department of Computer Science and Engineering, 9500 Gilman Drive, Mail Code 0404, La Jolla, CA 92093; e-mail: mvrable@cs.ucsd.edu.

DOI 10.1145/1629080.1629084 http://doi.acm.org/10.1145/1629080.1629084

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 article, 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 the cloud offers easy off-site storage, always a key concern for backup. 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 article, 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 interface (`put`, `get`, `delete`, `list`) that is trivially portable to virtually any online 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—competitive with integrated backup solutions, in spite of having no specific backup support in the underlying storage service. Second, we build a working prototype of this system using Amazon's Simple Storage Service (S3) and demonstrate 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 well as for a range of storage to network cost ratios.

In the remainder of this article, 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 [Preston 2006] utilities are common on Unix systems and will write a 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 faster to create, but mostly useless without the backups on which they are based.

Organizations may establish backup policies specifying at what granularity backups are made, and how long they are kept. These policies might then be implemented in various ways. For tape backups, 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 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. Cumulus supports flexible policies for backup retention: an administrator does not have to select at the start how long to keep backups, but rather can delete any snapshot at any point.

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. Many recent backup tools, including Cumulus, take advantage of this trend. Two approaches for comparing these systems are by the storage representation on disk, and by the interface between the client and the storage—while the disk could be directly attached to the client, often (especially with a desire to store backups remotely) communication will be over a network.

Rsync [Tridgell 1999] efficiently mirrors a filesystem across a network using a specialized network protocol to identify and transfer only those parts of files that have changed. Both the client and storage server must have rsync installed. Users typically want backups at multiple points in time, so rsnapshot and other wrappers around rsync exist 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.

The rdiff-backup [Escoto 2009] tool is similar to rsnapshot, but it changes the storage representation. The most recent snapshot is a mirror of the files, but the rsync algorithm creates compact deltas for reconstructing older versions—these reverse incrementals are more space efficient than full copies of files as in rsnapshot.

Another modification to the storage format at the server is to store snapshots in a content-addressable storage system. Venti [Quinlan and Dorward 2002] uses hashes of block contents to address data blocks, rather than a block number on disk. Identical data between snapshots (or even within a snapshot) is automatically coalesced into a single copy on disk—giving the space benefits of incremental backups automatically. Data Domain [Zhu et al. 2008] offers a similar but more recent and efficient product; in addition to performance improvements, it uses content-defined chunk boundaries so de-duplication can be performed even if data is offset by less than the block size.

A limitation of these tools is that backup data must be stored unencrypted at the server, so the server must be trusted. Box Backup [Summers and Wilson 2009] modifies the protocol and storage representation to allow the client to encrypt data before sending, while still supporting rsync-style efficient network transfers.

Most of the previous tools use a specialized protocol to communicate between the client and the storage server. An alternate approach is to target a more generic interface, such as a network file system or an FTP-like protocol. Amazon S3 [Amazon Web Services 2009] offers an HTTP-like interface to storage. The operations supported are similar enough between these different protocols—get/put/delete on files and list on directories—that a client can easily support multiple protocols. Cumulus tries to be network-friendly like rsync-based tools, while using only a generic storage interface.

Jungle Disk [jungledisk 2009] can perform backups to Amazon S3. However, the design is quite different from that of Cumulus. Jungle Disk is first a network filesystem with Amazon S3 as the backing store. Jungle Disk can also be used for backups, keeping copies of old versions of files instead of deleting them. But since it is optimized for random access it is less efficient than Cumulus for pure backup—features like aggregation in Cumulus can improve compression, but are at odds with efficient random access.

Duplicity [Escoto and Loafman 2009] aggregates files together before storage for better compression and to reduce per-file storage costs at the server. Incremental backups use space-efficient rsync-style deltas to represent changes. However, because each incremental backup depends on the previous, space cannot be reclaimed from old snapshots without another full backup, with its associated large upload cost. Cumulus was inspired by duplicity, but avoids this problem of long dependency chains of snapshots.

Brackup [Fitzpatrick 2009] has a design very similar to that of Cumulus. Both systems separate file data from metadata: each snapshot contains a separate copy of file metadata as of that snapshot, but file data is shared where possible. The split data/metadata design allows old snapshots to be easily deleted. Cumulus differs from Brackup primarily in that it places a greater emphasis on aggregating small files together for storage purposes, and adds a segment cleaning mechanism to manage the inefficiency introduced. Additionally, Cumulus tries to efficiently represent small changes to all types of large files and can share metadata where unchanged; both changes reduce the cost of incremental backups.

Peer-to-peer systems may be used for storing backups. Pastiche [Cox et al. 2002] is one such system, and focuses on the problem of identifying and sharing data between different users. Pastiche uses content-based addressing for de-duplication. But if sharing is not needed, Brackup and Cumulus could use

Table I.
Comparison of Features among Selected Tools That Back Up to Networked Storage

	Multiple snapshots	Simple server	Incremental forever	Sub-file delta storage	Encryption
rsync			✓	N/A	
rsnapshot	✓		✓		
rdiff-backup	✓		✓	✓	
Box Backup	✓		✓	✓	✓
Jungle Disk	✓	✓	✓		✓
duplicity	✓	✓		✓	✓
Brackup	✓	✓	✓		✓
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.

peer-to-peer systems as well, simply treating it as another storage interface offering get and put operations.

While other interfaces to storage may be available—Antiquity [Weatherspoon et al. 2007] for example provides a log append operation—a get/put interface likely still works best since it is simpler and a single put is cheaper than multiple appends to write the same data.

Table I summarizes differences between some of the tools we have discussed 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 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. The interface consists 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. Cumulus neither requires nor uses support for reading and setting file attributes such as permissions and timestamps. 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.

Since the only way to modify a file in this narrow interface is to upload it again in full, we adopt a *write-once storage model*, in which a file is never modified after it is first stored, except to delete it to recover space. The write-once model provides convenient failure guarantees: since files are never modified in place, a failed backup run cannot corrupt old snapshots. At worst, it will leave a partially-written snapshot which can garbage-collected. Because Cumulus does not modify files in place, we can keep snapshots at multiple points in time simply by 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. Filesystems typically contain many small files (both our traces described later and others, such as Agrawal et al. [2007], support this observation). Aggregation of data produces larger files for storage at the server, which can be beneficial to:

—*Avoid inefficiencies associated with many small files*. Storage servers may dislike storing many small files for various reasons—higher metadata costs, wasted space from rounding up to block boundaries, and more seeks when reading. This preference may be expressed in the cost model of the provider. Amazon S3, for example, has both a per-request and a per-byte cost when storing a file that encourages using files greater than 100 KB in size.

—*Avoid costs in network protocols*. Small files result in relatively larger protocol overhead, and may be slower over higher-latency connections. Pipelining (if supported) or parallel connections may help, but larger segments make these less necessary. We study one instance of this effect in more detail in Section 5.4.6.

—*Take advantage of inter-file redundancy with segment compression*. Compression can be more effective when small files are grouped together. We examine this effect in Section 5.4.3.

—*Provide additional privacy when encryption is used*. Aggregation helps hide the size as well as contents of individual files.

Finally, as discussed in Sections 3.4 and 4.3, changes to small parts of larger files can be efficiently represented by effectively breaking those files into smaller pieces during backup. For the reasons listed above, reaggregating this data becomes even more important when subfile incremental backups are supported.

3.3 Snapshot Format

Figure 1 illustrates the basic format for backup snapshots. Cumulus snapshots logically consist of two parts: a *metadata log* which lists all the files backed up,

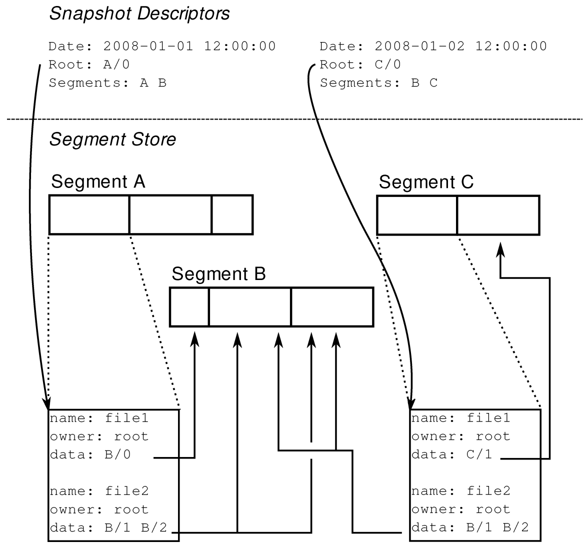


Fig. 1. Simplified schematic of the basic format for storing snapshots on a storage server. Two snapshots are shown, taken on successive days. Each snapshot contains two files. `file1` changes between the two snapshots, but the data for `file2` is shared between the snapshots. For simplicity in this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

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 without central coordination. 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 the tree (or, in the case of sharing, DAG) of objects. The root object in the tree is the start of the metadata log. The metadata log need not be stored as a flat object; it may contain pointers to objects containing other pieces of the metadata log. For example, if many files have not changed, then a single pointer to a portion of the metadata for an old snapshot may be written. The metadata objects eventually contain entries for individual files, with pointers to the file data as the leaves of 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 hash so that file integrity can be verified after a restore. Finally, it includes a list of pointers to objects containing the file data. Metadata is stored in a text, not binary, format to make it more transparent. Compression applied to the segments containing the metadata, however, makes the format space-efficient.

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

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, a garbage collection process deletes unreferenced segments when snapshots are removed. To simplify garbage-collection, each snapshot descriptor includes (though it is redundant) a summary of segments on which it depends.

Pointers within the metadata log include cryptographic hashes so that the integrity of all data can be validated starting from the snapshot descriptor, which can be digitally signed. Additionally, Cumulus writes a summary file with checksums for all segments so that it can quickly check snapshots for errors without a full restore.

3.4 Subfile Incrementals

If only a small portion of a large file changes between snapshots, only the changed portion of the file should be stored. The design of the Cumulus format supports this. The contents of each file is specified as a list of objects, so new snapshots can continue to point to old objects when data is unchanged. Additionally, pointers to objects can include byte ranges to allow portions of old objects to be reused even if some data has changed. We discuss how our implementation identifies data that is unchanged in Section 4.3.

3.5 Segment Cleaning

When old snapshots are no longer needed, space is reclaimed by deleting the root snapshot descriptors for those snapshots, then garbage collecting unreachable segments. 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) [Rosenblum and Ousterhout 1992].

There are two approaches that can be taken to segment cleaning given that multiple backup snapshots are involved. The first, *in-place cleaning*, is most like the cleaning in LFS. It identifies segments with wasted space and rewrites the segments to keep just the needed data.

This mode of operation has several disadvantages, however. It violates the write-once storage model, in that the data on which a snapshot depends is changed after the snapshot is written. It requires detailed bookkeeping to determine precisely which data must be retained. Finally, it requires downloading and decrypting old segments—normal backups only require an encryption key, but cleaning needs the decryption key as well.

The alternative to in-place cleaning is to never modify segments in old snapshots. Instead, Cumulus avoids referring to data in inefficient old segments when creating a new snapshot, and writes new copies of that data if needed. This approach avoids the disadvantages listed earlier, but is less space-efficient. Dead space is not reclaimed until snapshots depending on the old segments are deleted. Additionally, until then data is stored redundantly since old and new snapshots refer to different copies of the same data.

We analyzed both approaches to cleaning in simulation. We 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.

The simplest policy for selecting segments to clean is to set a minimum segment utilization threshold, α, that triggers cleaning of a segment. We define utilization as the fraction of bytes within the segment which are referenced by a current snapshot. For example, $\alpha = 0.8$ will ensure that at least 80% of the bytes in segments are useful. Setting $\alpha = 0$ disables segment cleaning altogether. Cleaning thresholds closer to 1 will decrease storage overhead for a single snapshot, but this more aggressive cleaning requires transferring more data.

More complex policies are possible as well, such as a cost-benefit evaluation that favors repacking long-lived segments. Cleaning may be informed by snapshot retention policies: cleaning is more beneficial immediately before creating a long-term snapshot, and cleaning can also consider which other snapshots currently reference a segment. Finally, segment cleaning may reorganize data, such as by age, when segments are repacked.

Though not currently implemented, Cumulus could use heuristics to group data by expected lifetime when a backup is first written in an attempt to optimize segment data for later cleaning (as in systems such as WOLF [Wang and Hu 2002]).

3.6 Restoring from Backup

Restoring data from previous backups may take several forms. A *complete restore* extracts all files as they were on a given date. A *partial restore* recovers one or a small number of files, as in recovering from an accidental deletion. As an enhancement to a partial restore, all available versions of a file or set of files can be listed.

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 downloaded should be only slightly larger than the size of the data to restore.

For partial restores, Cumulus downloads those segments that contain metadata for the snapshot to locate the files requested, then locates each of the segments containing file data. This approach might require fetching many segments—for example, if restoring a directory whose files were added incrementally over many days—but will usually be quick.

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. However, a client could keep a database of this information to allow more efficient queries.

3.7 Limitations

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

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. However, a specialized backup server could provide features such as automatically repacking backup data when deleting old snapshots, eliminating the overhead of client-side segment cleaning.

Cumulus, as designed, does not offer coordination between multiple backup clients, and so does not offer features such as deduplication between backups from different clients. While Cumulus could use convergent encryption [Douceur et al. 2002] to allow deduplication even when data is first encrypted at the client, several issues prevent us from doing so. Convergent encryption would not work well with the aggregation in Cumulus. Additionally, server-side deduplication is vulnerable to dictionary attacks to determine what data clients are storing, and storage accounting for billing purposes is more difficult.

Finally, 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. If it is known that the storage server and network protocol can efficiently deal with small files, however, 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 3200 lines of C++ source code (as measured by SLOCCount [Wheeler 2009]) implementing the core backup functionality, along with another roughly 1000 lines of Python for tasks such as restores, segment cleaning, and statistics gathering.

4.1 Local Client State

Each client stores on its 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. We do not need this information to recover data from a backup so its loss is not catastrophic, but 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 [Garnaat 2009] containing all other needed information.

Cumulus uses the local copy of the previous metadata log to quickly detect and skip over unchanged files based on modification time. Cumulus also uses it to delta-encode the metadata log for new snapshots.

An SQLite database keeps a record of recent snapshots and all segments and objects stored in them. The table of objects includes an index by content hash to support data deduplication. Enabling deduplication leaves Cumulus vulnerable to corruption from a hash collision [Henson 2003; Henson 2007], but, as with other systems, we judge the risk to be small. The hash algorithm (currently SHA-1) can be upgraded as weaknesses are found. In the event that client data must be recovered from backup, the content indices can be rebuilt from segment data as it is downloaded during the restore.

Note that the Cumulus backup format does not specify the format of this information stored locally. 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.

4.2 Segment Cleaning

The Cumulus backup program, written in C++, does not directly implement segment cleaning heuristics. Instead, a separate Cumulus utility program, implemented in Python, controls cleaning.

When writing a snapshot, Cumulus 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 marks the selected segments as “expired” in the local database. It also considers which snapshots refer to the segments, and how long those snapshots are likely to be kept, during cleaning. On subsequent backups, the Cumulus backup program re-uploads any data that is needed from expired segments. Since the database contains information about the age of all data blocks, segment data can be grouped by age when it is cleaned.

If local client state is lost, this age information will be lost. When the local client state is rebuilt all data will appear to have the same age, so cleaning may not be optimal, but can still be done.

4.3 Subfile 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 [Muthitacharoen et al. 2001].

When a file is first backed up, Cumulus divides it 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 storing each megabyte block, Cumulus computes a set of chunk signatures: it divides the data block into non-overlapping chunks and computes a (20-byte SHA-1 signature, 2-byte length) tuple for each chunk. The list of chunk signatures for each object is 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 limit the search for unmodified data to the chunks in the previous version of the file. Cumulus computes chunk signatures for the new file data, and matches with old chunks are written as a reference to the old data. New chunks are written out to a new object. However, Cumulus could be extended to perform global data deduplication while maintaining backup format compatibility.

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` can provide encryption.

Similarly, network protocols are implemented by calling out to external scripts. Cumulus first writes segments to a temporary directory, then calls an upload script to transfer them in the background while the main backup process continues. Slow uploads will eventually throttle the backup process so that the required temporary storage space is bounded. Upload scripts may be quite simple; a script for uploading to Amazon S3 is merely 12 lines long in Python using the boto [boto 2009] library.

4.5 Snapshot Restores

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

To reduce disk space requirements, the restore tool downloads segments as needed instead of all at once at the start, and can delete downloaded segments as it goes along. The restore tool downloads the snapshot descriptor first, followed by the metadata. The backup tool segregates data and metadata into separate segments, so this phase does not download any file data. Then, file contents are restored—based on the metadata, as each segment is downloaded data from that segment is restored. For partial restores, only the necessary segments are downloaded.

Currently, in the restore tool it is possible that a segment may be downloaded multiple times if blocks for some files are spread across many segments. However, this situation is merely an implementation issue and can be fixed by restoring data for these files non-sequentially as it is downloaded.

Finally, Cumulus includes a FUSE [fuse 2009] interface that allows a collection of backup snapshots to be mounted as a virtual filesystem on Linux, thereby providing random access with standard filesystem tools. This interface relies on the fact that file metadata is stored in sorted order by filename, so a binary search can quickly locate any specified file within the metadata log.

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, subfile incrementals) 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 to the experiments.

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 taken 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 complete backups of all file data, and enables evaluation of the effects of compression and sub-file incrementals. Table II summarizes the key statistics of each trace.

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. We use a simulator, rather than our prototype, because a parameter sweep of the space of cleaning

Table II.
Key Statistics of the Two Traces Used in the Evaluations. File Counts and Sizes are for the Last Day in the Trace. “Entries” is Files Plus Directories, Symlinks, etc

	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

parameters on datasets as large as our traces is not feasible in a reasonable amount of time.

The simulator tracks three overheads associated with performing backups. It tracks storage overhead, or 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 predetermined 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. This schedule emulates the backup policy an enterprise might employ.

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 subfile 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 approaches. To

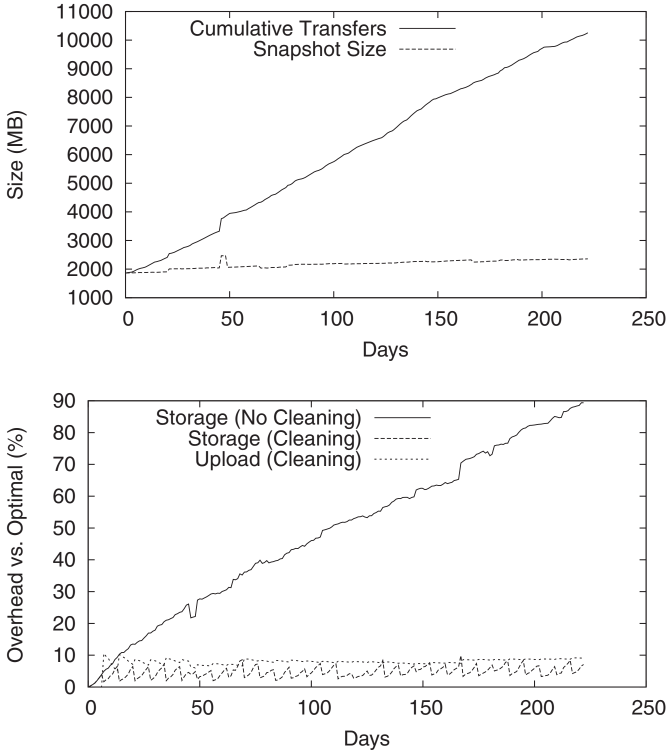
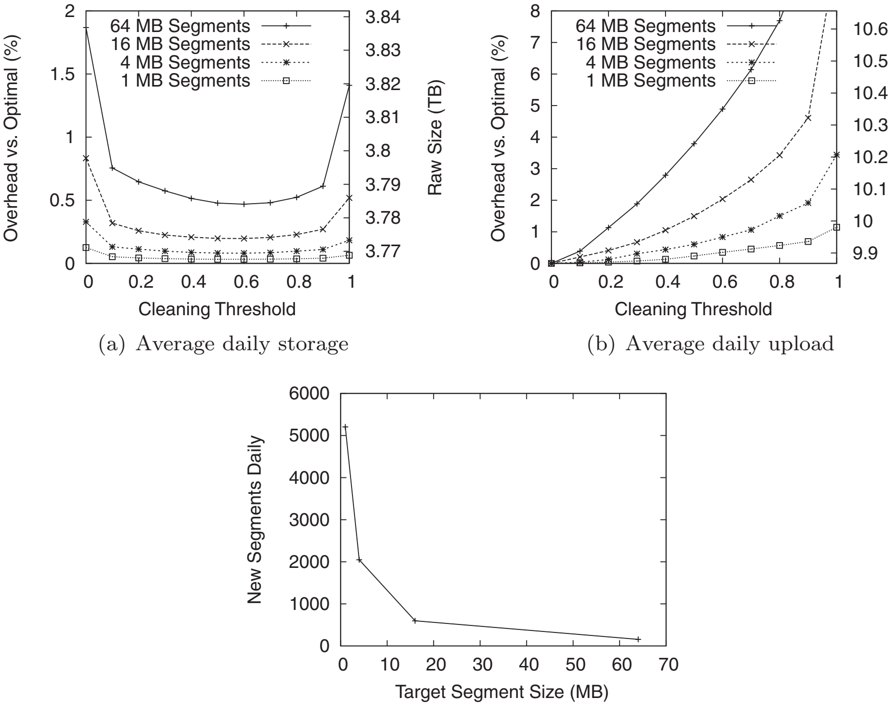


Fig. 2. (a) Storage and network overhead for an optimal backup of the files from the user trace. (b) Overheads with and without cleaning; segments are cleaned at 60% utilization. Only storage overheads are shown for the no-cleaning case since there is no network transfer overhead without cleaning.

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. 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 larger than the average file.

As an example of these costs and how we measure them, Figure 2(a) 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.



(a) Average daily storage

(b) Average daily upload

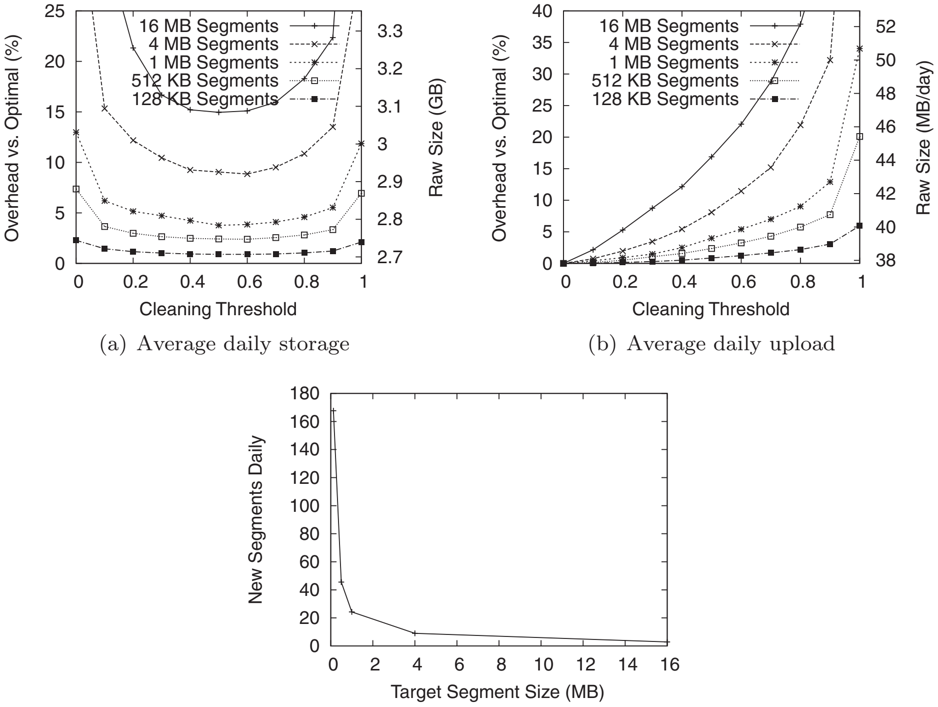
(c) Average segments per day

Fig. 3. Overheads for backups in the fileserver trace.

Figure 2(b) 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 nearly 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). In our next set of experiments, we explore the parameter space to quantify the impact of these two metrics on backup performance.

Figures 3 and 4 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



(a) Average daily storage

(b) Average daily upload

(c) Average segments per day

Fig. 4. Overheads for backups in the user trace.

(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 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 value 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; we exclude the first day 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 this behavior further below.

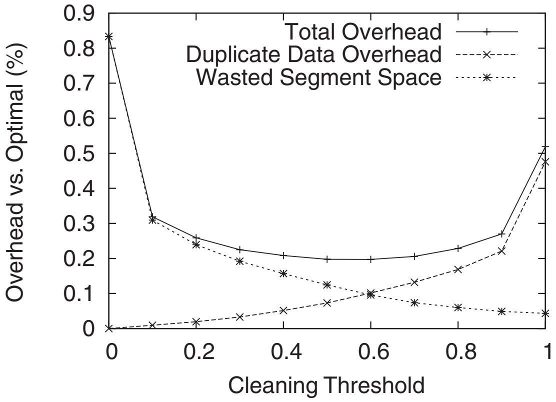


Fig. 5. Detailed breakdown of storage overhead when using a 16 MB segment size for the fileserver workload.

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 smaller average files and more churn in the file data, but overall the overhead penalties remain low.

Figures 3(a) and 4(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 5 explains why storage overhead increases again at higher cleaning thresholds. It shows three curves, the 16 MB segment size curve from Figure 3(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.

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. While similar, there are differences between this problem and the typical evaluation of cleaning policies for a typical log-structured file system: instead of a fixed disk size and a goal to minimize I/O, we have no fixed limits but want to minimize monetary cost.

Table III.
Costs for Backups in US Dollars, if Performed Optimally, for the Fileserver and User Traces Using Current Prices for Amazon S3

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

We use the prices for Amazon S3 as an initial point in the pricing space. As of August 2009, these prices are (in US dollars):

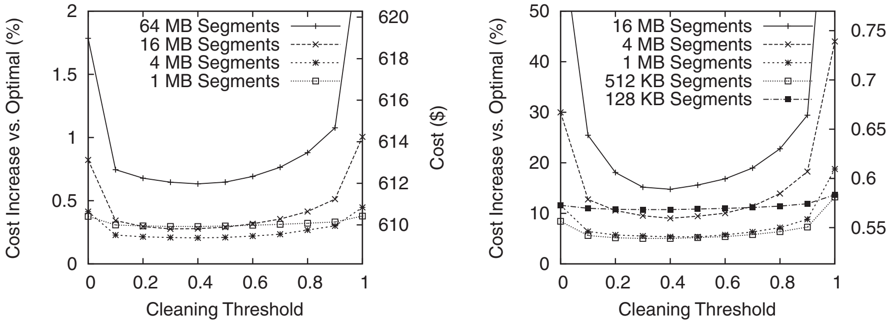
Storage:	$0.15 per GB · month,
Upload:	$0.10 per GB,
Segment:	$0.01 per 1000 files uploaded.

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, that is, 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 III 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 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 than our user workload, yearly ongoing costs are roughly US$50.

Whereas Figure 3 explored the parameter space of cleaning thresholds and segment sizes in terms of resource overhead, Figure 6 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, 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, though, 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). However, since overhead is not overly sensitive to the cleaning threshold, Cumulus still provides good performance even if the cleaning threshold is not tuned optimally. Furthermore, in contrast to resource usage, decreasing segment size does not



(a) Fileserver (b) User

Fig. 6. Costs in US dollars for backups in the fileserver assuming Amazon S3 prices.

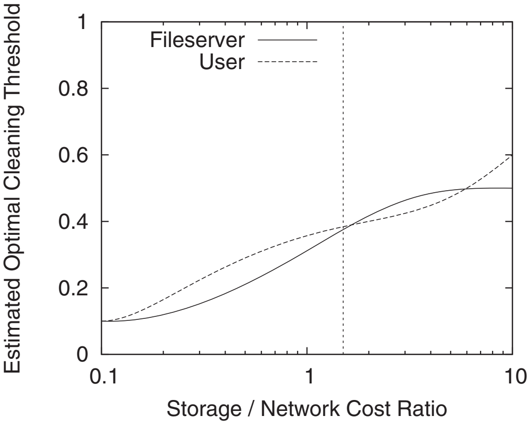


Fig. 7. Sensitivity of the optimal cleaning threshold to the relative cost of storage vs. network. Amazon S3 currently has a storage/network cost ratio of 1.5, shown by a vertical line.

always decrease overall cost. At some point—in this case between 1 and 4 MB—decreasing segment size increases overall cost due to the per-file pricing. We do not evaluate segment sizes less than 1 MB for the fileserver trace since, by 1 MB, smaller segments are already a loss. The results for the user workload, although not shown, are qualitatively similar, with a segment size of 0.5 MB to 1 MB best.

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 7 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.

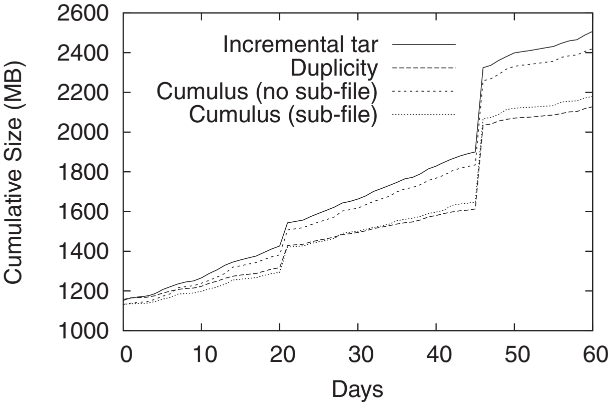


Fig. 8. Cumulative storage costs for actual runs of multiple backup systems.

5.4 Prototype Evaluation

In our final set of experiments, we evaluate the performance of the complete Cumulus prototype. We compare the overhead of the Cumulus prototype implementation with other backup systems, both more traditional and those targeting cloud storage. We also evaluate the sensitivity of compression on segment size, the overhead of metadata in the implementation, the performance of subfile incrementals and restores, and the time it takes to upload data to a remote service like Amazon S3.

5.4.1 *System Comparisons: Noncloud Systems.* First we provide results from running our Cumulus prototype and compare with two existing backup tools which, though not designed for cloud storage, provide a baseline for comparison. We use the complete file contents included in the user trace to accurately measure the behavior 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, with and without subfile incrementals.

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

Neither Duplicity nor tar can reclaim space from older backups without making another full backup snapshot, while Cumulus can. But since none of these systems upload extra data unnecessarily, they provide a good baseline for the optimal network uploads.

Figure 8 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. This cumulative size is thus equivalent to total network uploads. 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 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 subfile incrementals. Finally, when subfile incremental backups are enabled in Cumulus, its daily upload rate falls so that it almost exactly matches that of duplicity.

Initial full backups for all tools are comparable: around 1150 MB for all. Cumulus saves about 20 MB due to data de-duplication not performed by the other systems. More interesting are the sizes of the incremental backups. Over a two month period, tar uploads 1355 MB in incrementals and duplicity uploads 971 MB. For comparison, Cumulus uploads 1287 MB without subfile incrementals, and 1048 MB with subfile incrementals. Using duplicity as a proxy for near-optimal backups with subfile incrementals, the network upload overhead of Cumulus is under 8%. And, for this slight additional cost, Cumulus can reclaim storage space used by old snapshots.

5.4.2 *System Comparisons: Cloud Systems.* Next, we provide some results from running our Cumulus prototype and compare with two existing backup tools that also target Amazon S3: Jungle Disk and Brackup. Unlike in the previous comparison, here all systems can easily delete old snapshots to reclaim space.

As before, we use the complete file contents included in the user trace, this time over a three month period. We compute the average cost, per month, broken down into storage, upload bandwidth, and operation count (files created or modified).

We configured Cumulus to clean segments with less then 60% utilization on a weekly basis. We evaluate Brackup with two different settings. The first uses the `merge_files_under=1kB` option to only aggregate files if those files are under 1 KB in size (this setting is recommended). Since this setting still results in many small files (many of the small files are still larger than 1 KB), a “high aggregation” run sets `merge_files_under=16kB` to capture most of the small files and further reduce the operation count. Brackup includes the digest database in the files backed up, which serves a role similar to the database Cumulus stores locally. For fairness in the comparison, we subtract the size of the digest database from the sizes reported for Brackup.

Both Brackup and Cumulus use `gpg` to encrypt data in the test; `gpg` compresses the data with `gzip` prior to encryption. Encryption is enabled in Jungle Disk, but no compression is available.

In principle, we would expect backups with Jungle Disk to be near optimal in terms of storage and upload since no space is wasted due to aggregation. But, as a tradeoff, Jungle Disk will have a much higher operation count. In practice,

Table IV.
Cost Comparison for Backups Based on Replaying Actual File Changes in the User Trace over a Three Month Period. Costs for Cumulus are Lower than those Shown in Table III Since that Evaluation Ignored the Possible Benefits of Compression and Subfile Incrementals, Which are Captured Here. Values are Listed on a Per-Month Basis

System	Storage	Upload	Operations
Jungle Disk	≈ 2 GB	1.26 GB	30000
	$0.30	$0.126	$0.30
Brackup	1.340 GB	0.760 GB	9027
(default)	$0.201	$0.076	$0.090
Brackup	1.353 GB	0.713 GB	1403
(aggregated)	$0.203	$0.071	$0.014
Cumulus	1.264 GB	0.465 GB	419
	$0.190	$0.047	$0.004

Jungle Disk will also suffer from a lack of de-duplication, subfile incrementals, and compression.

Table IV compares the estimated backup costs for Cumulus with Jungle Disk and Brackup. Several key points stand out in the comparison.

- Storage and upload requirements for Jungle Disk are larger, owing primarily to the lack of compression.
- Except in the high aggregation case, both Brackup and Jungle Disk incur a large cost due to the many small files stored to S3. The per-file cost for uploads is larger than the per-byte cost, and for Jungle Disk significantly so.
- Brackup stores a complete copy of all file metadata with each snapshot, which in total accounts for 150–200 MB/month of the upload cost. The cost in Cumulus is lower since Cumulus can reuse metadata.

The Cumulus prototype thus shows that a service with a simple storage interface can achieve low overhead, and that Cumulus can achieve a lower total cost than other existing backup tools targeting S3.

While perhaps none of the systems are yet optimized for speed, initial full backups in Brackup and Jungle Disk were both notably slow. In the tests, the initial Jungle Disk backup took over six hours, Brackup (to local disk, not S3) took slightly over two hours, and Cumulus (to S3) approximately 15 minutes. For comparison, simply archiving all files with `tar` to local disk took approximately 10 minutes.

For incremental backups, elapsed times for the tools were much more comparable. Jungle Disk averaged 248 seconds per run archiving to S3. Brackup averaged 115 seconds per run and Cumulus 167 seconds, but in these tests each were storing snapshots to local disk rather than to Amazon S3.

5.4.3 *Segment Compression.* Next we 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,

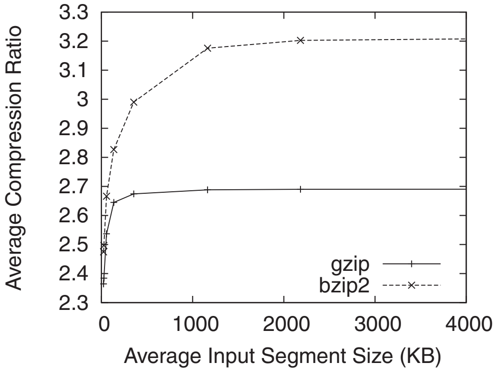


Fig. 9. Measured compression ratio for packing small files together based on a sample of files from the user trace.

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.

Figure 9 shows the observed compression ratios from packing together groups of smaller files in the user trace. These observed compression ratios are higher than those observed for the entire trace since the sample of files taken excludes very large files (often not very compressible). It shows the dependence of compression on segment size: larger segments give compression algorithms more data across which to find commonality.

When aggregating data together into segments, larger input segment sizes yielded better compression, up to about 300 KB when using `gzip` and 1–2 MB for `bzip2` where compression ratios leveled off.

5.4.4 *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. On days where only differences were written out, though, the average metadata log delta was under 2% of the size of a full metadata log. Overall, across all the snapshots taken, the data written out for file metadata was approximately 5% of the total size of the file data itself.

5.4.5 *Subfile Incrementals.* The effectiveness of subfile incrementals depends greatly on the workload. In the fileserver trace, data in new files dominates changes to existing files, so the possible gains from accurate subfile incrementals are small. In the user trace, more data is modified in-place and so the possible additional gains from subfile incrementals are greater, as Section 5.4.1 showed.

Table V.
Comparison of Cumulus Subfile Incrementals with an Idealized System Based on Rdiff, Evaluated on Two Sample Files from the User Trace

	File A	**File B**
File size	4.860 MB	5.890 MB
Compressed size	1.547 MB	2.396 MB
Cumulus size	5.190 MB	3.081 MB
Size overhead	235%	29%
rdiff delta	1.421 MB	122 KB
Cumulus delta	1.527 MB	181 KB
Delta overhead	7%	48%

To further understand the subfile incrementals in Cumulus, we use a microbenchmark. We identify and extract several files from the user trace that are frequently modified in place. The files are taken from a 30-day period at the start of the trace. File A is a frequently-updated Bayesian spam filtering database, about 90% of which changes daily. File B records the state for a file-synchronization tool (unison), of which an average of 5% changes each day—however, unchanged content may still shift to different byte offsets within the file. While these samples do not capture all behavior, they do represent two distinct and notable classes of subfile updates.

To provide a point of comparison, we use `rdiff`[1] to generate an rsync-style delta between consecutive file versions. Table V summarizes the results.

The *size overhead* measures the storage cost of subfile incrementals in Cumulus. To reconstruct the latest version of a file, Cumulus might need to read data from many past versions, though cleaning will try to keep this bounded. This overhead compares the average size of a daily snapshot ("Cumulus size") against the average compressed size of the file backed up. As file churn increases overhead tends to increase.

The *delta overhead* compares the data that must be uploaded daily by Cumulus ("Cumulus delta") against the average size of patches generated by rdiff ("rdiff delta"). When only a small portion of the file changes each day (File B), rdiff is more efficient than Cumulus in representing the changes. However, subfile incrementals are still a large win for Cumulus, as the size of the incrementals is still much smaller than a full copy of the file. When large parts of the file change daily (File A), the efficiency of Cumulus approaches that of rdiff.

5.4.6 *Upload Time.* Next, 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 Python library to interface with S3.

As our measurements are from one experiment from a single computer (on a university campus network) and at a single point in time (in early 2008)

[1] http://librsync.sourceforge.net/.

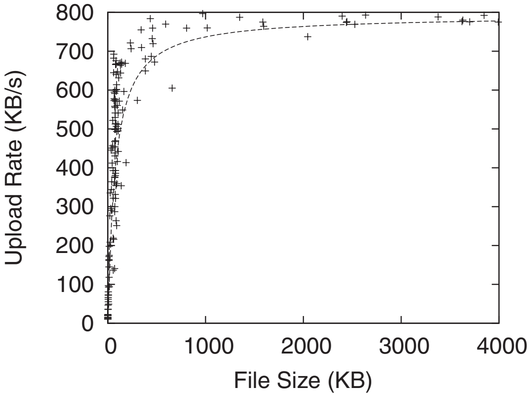


Fig. 10. Measured upload rates for Amazon S3 as a function of file size. The measurements are fit to a curve where each upload consists of a fixed delay (for connection establishment and receiving a status code after the transfer) along with a fixed-rate upload.

they should not be taken as a good measure of the overall performance of S3. Figure 10 shows the measured upload rate as a function of the size of the file uploaded. The time for an upload is measured from the start of the HTTP request until receipt of the final response. For large files—a megabyte or larger—uploads proceed at a maximum rate of about 800 KB/s. According to our results there is an overhead equivalent to a latency of roughly 100 ms per upload, and for small files this dominates the actual time for data transfer. More recent tests indicate that rates may have improved; however, the basic lesson—that file size matters—is still valid.

The S3 protocol, based on HTTP, does not support pipelining multiple upload requests. Multiple uploads in parallel could reduce overhead somewhat. Still, it remains beneficial to perform uploads in larger units.

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.

5.4.7 *Restore Time.* To completely restore all data from one of the user snapshots takes approximately 11 minutes, comparable to but slighly faster than the time required for an initial full backup.

When restoring individual files from the user dataset, almost all time is spent extracting and parsing metadata—there is a fixed cost of approximately 24 seconds to parse the metadata to locate requested files. Extracting requested files is relatively quick, under a second for small files.

Both restore tests were done from local disk; restoring from S3 will be slower by the time needed to download the data.

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 (such as 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 $0.24 per month. (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 strategy 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 article has focused on an admittedly simple application, we believe it identifies a key research agenda influencing the future of "cloud computing": can one build a competitive product economy around a cloud of abstract commodity resources, or do underlying technical reasons ultimately favor an integrated service-oriented infrastructure?

ACKNOWLEDGMENTS

The authors would like to thank Chris X. Edwards and Brian Kantor for assistance in collecting fileserver traces and other computing support. We would also like to thank our shepherd Niraj Tolia and the anonymous reviewers for their time and insightful comments regarding Cumulus and this article.

REFERENCES

AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. 2007. A five-year study of file-system metadata. *ACM Trans. Stor. 3*, 3, 9.

AMAZON WEB SERVICES. 2009. Amazon Simple Storage Service. http://aws.amazon.com/s3/.

boto 2009. boto: Python interface to Amazon Web Services. http://code.google.com/p/boto/.

COX, L. P., MURRAY, C. D., AND NOBLE, B. D. 2002. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 285–298.

DOUCEUR, J. R., ADYA, A., BOLOSKY, W. J., SIMON, D., AND THEIMER, M. 2002. Reclaiming space from duplicate files in a serverless distributed file system. Tech. rep. MSR-TR-2002-30.

ESCOTO, B. 2009. rdiff-backup. http://www.nongnu.org/rdiff-backup/.

ESCOTO, B. AND LOAFMAN, K. 2009. Duplicity. http://duplicity.nongnu.org/.

FITZPATRICK, B. 2009. Brackup. http://code.google.com/p/brackup/, http://brad.livejournal.com/tag/brackup.

fuse 2009. FUSE: Filesystem in userspace. http://fuse.sourceforge.net/.

Garnaat, M. 2009. Sqlite. http://www.sqlite.org/.

Henson, V. 2003. An analysis of compare-by-hash. *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*.

Henson, V. 2007. The code monkey's guide to cryptographic hashes for content-based addressing. http://www.linuxworld.com/news/2007/111207-hash.html.

jungledisk 2009. Jungle disk. http://www.jungledisk.com/.

librsync 2009. librsync. http://librsync.sourceforge.net/.

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

Preston, W. C. 2006. *Backup & Recovery*. O'Reilly.

Quinlan, S. and Dorward, S. 2002. Venti: a new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association.

Rosenblum, M. and Ousterhout, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1, 26–52.

Summers, B. and Wilson, C. 2009. Box backup. http://www.boxbackup.org/.

Tridgell, A. 1999. Efficient algorithms for sorting and synchronization. Ph.D. thesis, Australian National University.

Wang, J. and Hu, Y. 2002. WOLF–A novel reordering write buffer to boost the performance of log-structured file systems. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association.

Weatherspoon, H., Eaton, P., Chun, B.-G., and Kubiatowicz, J. 2007. Antiquity: Exploiting a secure log for wide-area distributed storage. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*. ACM, New York, NY, 371–384.

Wheeler, D. A. 2009. SLOCCount. http://www.dwheeler.com/sloccount/.

Zhu, B., Li, K., and Patterson, H. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 269–282.

Received August 2009; accepted August 2009