MICHAEL VRABLE, STEFAN SAVAGE, AND GEOFFREY M. VOELKER

Cumulus: filesystem backup to the Cloud

Michael Vrable is pursuing a Ph.D. in computer science at the University of California, San Diego, and is advised by professors Stefan Savage and Geoffrey Voelker. He received an M.S. in computer science from UCSD (2007) and a B.S. in mathematics and computer science from Harvey Mudd College (2004).

mvrable@cs.ucsd.edu

Stefan Savage is an associate professor of computer science at the University of California, San Diego. He has a B.S. in history and reminds his colleagues of this fact anytime the technical issues get too complicated.

savage@cs.ucsd.edu

Geoffrey M. Voelker is an associate professor of computer science and engineering at the University of California, San Diego. He works in computer systems and networking.

voelker@cs.ucsd.edu

CUMULUS IS A SYSTEM FOR EFFICIENTLY implementing filesystem backups over the Internet, taking advantage of the growing availability of cheap storage options available online. Cloud service offerings such as Amazon's Simple Storage Service (S3), a part of Amazon Web Services, offer cheap storage at a fixed cost per gigabyte (no minimums or maximums) and are appealing for backup, since they provide an easy way to safely store data off-site.

There are pre-packaged online services specifically built for backup, such as Mozy and Carbonite. Cumulus explores the other end of the design space: building on top of a very generic cloud storage layer, an example of what we refer to as building on the "thin cloud." Using a generic, minimalist interface means that Cumulus is portable to virtually any online storage service—the client implements all application logic. Cumulus is not unique in this approach, but compared with existing backup tools targeting S3, Cumulus achieves lower costs, showing that this limited interface is not an impediment to achieving a very low and competitive cost for backup.

Related Tools

Unlike many traditional backup tools, Cumulus is not designed to stream backup data to tape. Cumulus instead takes advantage of the random access to files provided by online storage services—though it does still group writes together, since remote storage operations have a cost.

Unlike tools such as rsync, rdiff-backup, and boxbackup, no specialized code for Cumulus executes at the remote storage server. Cumulus cannot rely on a customized network protocol or run code at the server to manipulate snapshot data directly. However, like these systems, Cumulus does still attempt to be network-efficient, sending only changes to files over the network. If a user restores data, the client is responsible for reconstructing the snapshots from any deltas that were sent previously.

Other backup tools exist that target Amazon S3. Jungle Disk is a general-purpose network filesystem with S3 as the backing store; it can be used to store backups but has higher overhead, since it is optimized for random access to files. Brackup is quite similar to Cumulus, though Cumulus in-

cludes aggregation and cleaning mechanisms (described later) and can more efficiently represent incremental changes. Duplicity represents incremental backups very efficiently but cannot easily delete old snapshots. All of these systems, like Cumulus, can encrypt data before it is stored at the remote server.

Design

Cumulus stores backups on a remote server but, to be as portable as possible, imposes very few requirements on the server. Only four operations are required: put/get for storing and retrieving files, list for identifying data that is stored, and delete for reclaiming space. Cumulus does not depend upon the ability to read or write subsets of a file, nor does it need (or even use) support for reading and setting file attributes such as permissions and timestamps. The interface is simple enough to be implemented on top of any number of protocols: FTP, SFTP, WebDAV, Amazon's S3, or nearly any network file system.

Cumulus also adopts a *write-once storage model*: a file is never modified after it is first stored, except to be deleted 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, a failure will leave a partially written snapshot which can later be garbage-collected. Cumulus can keep snapshots at multiple points in time simply by not deleting the files that make up old snapshots.

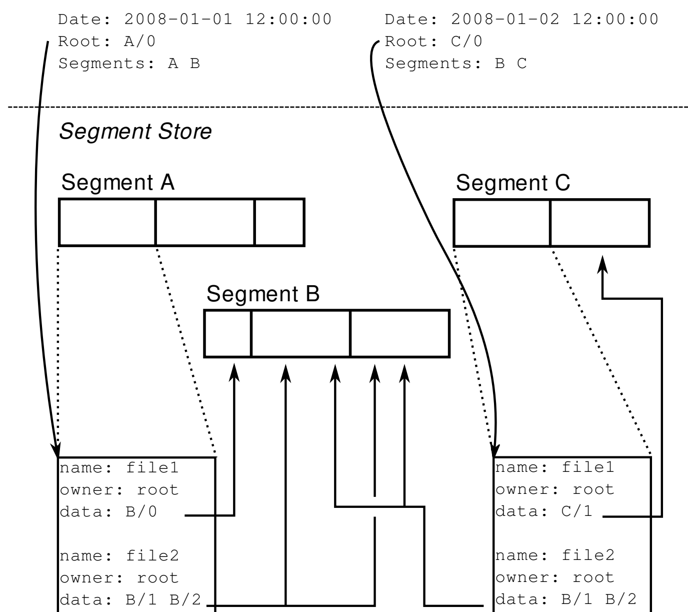


FIGURE 1: SIMPLIFIED SCHEMATIC OF THE BASIC FORMAT FOR STORING SNAPSHOTS ON A STORAGE SERVER. TWO SNAPSHOTS ARE SHOWN, TAKEN ON SUCCESSIVE DAYS. EACH SNAPSHOT CONTAINS TWO FILES. *FILE1* CHANGES BETWEEN THE TWO SNAPSHOTS, BUT THE DATA FOR *FILE2* IS SHARED BETWEEN THE SNAPSHOTS. FOR SIMPLICITY IN THIS FIGURE, SEGMENTS HAVE LETTERS AS NAMES INSTEAD OF THE 128-BIT UUIDS USED IN PRACTICE.

The basic Cumulus snapshot format is illustrated in Figure 1. A snapshot logically consists of two main parts. A *metadata log* lists all the files backed up as well as ownership, modification times, and similar information. Cumulus stores file *data* separately. Both data and metadata are broken apart

into smaller blocks, and a backup is structured as a tree (or sometimes a directed acyclic graph)—the block at the start of the metadata log contains pointers to other portions of the metadata log, which eventually contains pointers to data blocks for files. Cumulus stores metadata in textual, not binary, format. A *snapshot descriptor* points to the root of each backup snapshot.

Where duplicate data exists there may be multiple pointers to the same data blocks, making backups more space-efficient. Successive backup snapshots look something like the snapshots in a copy-on-write filesystem: multiple backup roots exist, but, when unchanged, data and metadata blocks are shared between the snapshots.

Aggregation and Cleaning

Backups in Cumulus would be straightforward if each backup were simply stored as a collection of blocks as described. However, these blocks will often be fairly small and, for many storage back ends, there is a penalty for storing large numbers of small files. For example, in addition to per-byte upload costs, Amazon S3 charges a small amount for each `put` operation.

To reduce costs, Cumulus aggregates blocks before sending them to a storage server. The example in Figure 1 illustrates this. We say that blocks are aggregated into *segments*, and Cumulus stores each segment as a separate file on the remote storage server. Each segment is internally structured as a tar file (so standard UNIX tools can unpack it), and segments may be filtered through a compression program (such as gzip) or encrypted (with gpg) before being sent over the network. Each segment has a unique name; we use a randomly generated 128-bit UUID so that segment names can be assigned without central coordination. Blocks are numbered sequentially within a segment.

Aggregation of data into segments can decrease costs but brings added complexity. When old snapshots are no longer needed, Cumulus reclaims space by garbage-collecting unused segments. It may be, however, that some segments only contain a small fraction of useful data. The remainder of these segments—data used only by deleted snapshots—is now wasted space. This problem is similar to the problem of reclaiming space in the Log-Structured File System (LFS) [1].

To reclaim space, Cumulus includes a *segment cleaner* that operates in two steps. First, it identifies segments which contain very little data and marks them as expired. Then, on the following backup run, Cumulus re-uploads (in new segments) any data that is still needed from the expired segments. Segment cleaning never requires downloading old segments. Cleaning cannot immediately delete expired segments when old snapshots still refer to them, but Cumulus can free them as the older snapshots are deleted.

Implementation

Our prototype Cumulus implementation is relatively compact: only slightly over 3,200 lines of C++ source code implementing the core backup functionality, along with another roughly 1,000 lines of Python for tasks such as restores, segment cleaning, and statistics gathering.

Each client stores on its local disk information about recent backups, primarily so that it can detect which files have changed and properly reuse blocks from previous snapshots. 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.

To simplify the implementation and keep Cumulus flexible, we implement several tasks as external scripts. Helper scripts filter data to perform compression and encryption. External scripts also handle file transfers—local storage and transfers to Amazon S3 are supported, but adding additional storage back-ends is straightforward.

Some files, such as log files or database files, may only be partly changed between backup runs. Our Cumulus implementation can efficiently represent these partial changes to files: the metadata log entry for a file can refer to a mixture of old and new blocks, or even parts of blocks. Cumulus computes these sub-file incrementals in a manner similar to that used in the Low-Bandwidth File System [2]: it divides data into variable-sized chunks of approximately 4KB, and it detects duplicate data between different versions of a file at a chunk granularity.

We implemented the restore functionality in Python. To reduce disk space requirements, the restore tool downloads segments as needed during the restore instead of all at once at the start. When restoring selected files from a snapshot, it downloads only the necessary segments. Cumulus also includes a FUSE 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.

Evaluation

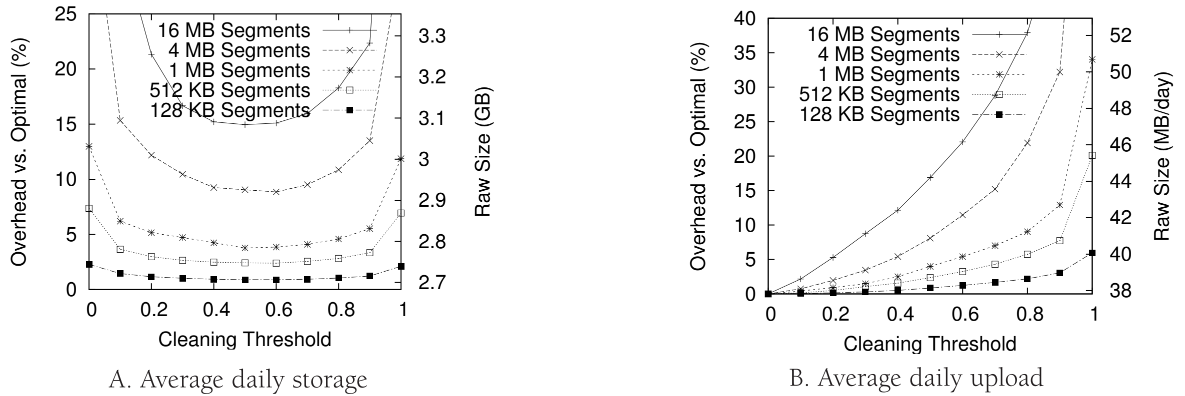
We use both trace-based simulation and a prototype implementation to evaluate the use of thin cloud services for remote backup. To drive our evaluation of Cumulus we replay a set of backups (taken with earlier versions of Cumulus) from a personal computer. These snapshots cover a period of over seven months and include an average of 2.4GB of data in each snapshot, with 40MB of data created or modified each day. In the FAST conference paper [3] we also consider traces taken from a research group file server. However, the end-user scenario is both more demanding (in terms of overhead within Cumulus) and likely more similar to expected uses for Cumulus.

Backup Simulations

Most of the overhead introduced by Cumulus is due to aggregation of data into segments and the associated cleaning costs. To better understand how this overhead depends on the details of aggregation and cleaning, we consider different scenarios in simulation using trace data, which allows us to explore the many possible parameter settings quickly.

The simulator tracks three overheads associated with performing backups, corresponding to the three quantities for which online services typically charge: daily storage requirements, network uploads, and an operation count (number of segments uploaded). The simulator makes several simplifications—it ignores file compression, sub-file incrementals, and file metadata overhead—but the prototype evaluation includes these.

In this simplified setting we compare Cumulus against an idealized *optimal backup* in which no space is wasted due to aggregation. In the optimal backup, each unique piece of data is transferred over the network and stored only once.



A. Average daily storage B. Average daily upload

FIGURE 2: OVERHEADS FOR BACKUPS IN THE USER TRACE

Figure 2 shows the simulated overheads for a variety of parameter settings. Storage overhead compares the storage required at the server for up to 12 recent backup snapshots, averaged over the several months of the backup trace, against the minimum required (optimal backup). Network overhead is similar, but compares the average daily upload size against the optimal. The x-axis of each graph shows the results of varying the segment cleaning aggressiveness: a cleaning threshold of 0.6 means that any segments less than 60% utilized will be expired and marked for cleaning. Cleaning thresholds near zero indicate very little cleaning, and those near one indicate very aggressive segment cleaning. In addition, we consider the effect of aggregation by grouping data into segments from as small as 128KB to as large as 16MB.

Storage and upload overheads improve with decreasing segment size: smaller segments result in less wasted space in segments and less cleaning needed. As expected, increasing the cleaning threshold increases the network upload overhead: frequently rewriting segments requires more data to be uploaded. For very low cleaning thresholds, storage overhead grows due to wasted space in segments. When cleaning very aggressively, however, storage overhead also grows: aggressive cleaning produces a high segment churn, which, when storing multiple snapshots, means there may be multiple copies of the same data. In between is a happy medium with relatively low storage overhead.

We can combine all these overheads into the single number that matters to an end user: monthly price. In this analysis, we use prices for Amazon S3 (values 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 100KB of data, i.e., when uploading 100KB files, half of the cost is for the bandwidth and half for the upload request itself. We would expect that segments somewhat larger than 100KB would achieve a minimum cost.

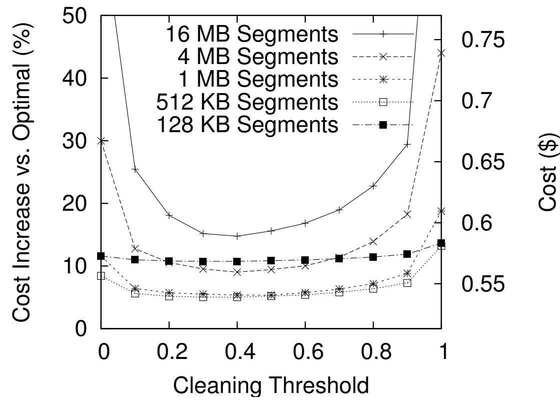


FIGURE 3: COSTS (US$) FOR BACKUPS FOR THE USER TRACE ASSUMING AMAZON S3 PRICES

Figure 3 shows the dollar costs from the Cumulus simulations. With per-segment costs included, a very small segment size becomes more expensive. At a segment size of 0.5–1 MB and a cleaning threshold near 0.5, Cumulus achieves costs competitive with the optimal: within about 5% of optimal and only slightly over $0.50 per month. The majority (over 75%) of the monthly cost pays for storage, with upload bandwidth a minor component. Importantly, the overhead is not overly sensitive to the system parameters, so Cumulus still provides good performance even if not tuned optimally.

Prototype Evaluation

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

TABLE 1: 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 FROM SIMULATION, IN PART BECAUSE SIMULATION IGNORED THE BENEFITS OF COMPRESSION AND SUB-FILE INCREMENTALS. VALUES ARE LISTED ON A PER-MONTH BASIS.

Finally, 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. We use the complete file contents from the user trace to accurately measure the behavior of our full Cumulus prototype and other real backup systems. We compute the average cost, per month, broken down into storage, upload bandwidth, and operation count (files created or modified). Each system keeps only the single most recent snapshot on each day.

Cumulus cleans segments at less than 60% utilization on a weekly basis. We evaluate Brackup with two different settings. The first uses the option of merge_files_under=1kB to only aggregate files if they are under 1KB in size

(this setting is recommended). Since this setting still results in many small files (many of the small files are still larger than 1KB), 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, Jungle Disk experiences overhead from a lack of de-duplication, sub-file incrementals, and compression.

Table 1 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 store metadata changes as incrementals.

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.

Conclusions

The market for Internet-hosted backup service continues to grow. 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) and the increasing competition due to the low barrier to switching providers, thus driving down prices.

Cumulus source code is available at http://sysnet.ucsd.edu/projects/cumulus/.

REFERENCES

[1] Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems* 10(1):26–52, 1992.

[2] Athicha Muthitacharoen, Benjie Chen, and David Mazières, "A Low-Bandwidth Network File System," *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (ACM, 2001), pp. 174–187.

[3] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker, "Cumulus: File-System Backup to the Cloud," *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)* (USENIX Association, 2009), pp. 225–238.