UNIVERSITY OF CALIFORNIA, SAN DIEGO

Migrating Enterprise Storage Applications to the Cloud

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Michael Daniel Vrable

Committee in charge:

Professor Stefan Savage, Co-Chair
Professor Geoffrey M. Voelker, Co-Chair
Professor Bill Lin
Professor Paul Siegel
Professor Amin Vahdat

2011

The dissertation of Michael Daniel Vrable is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2011

DEDICATION

To my family, for all the support I’ve received.

EPIGRAPH

If I have seen further it is only by
standing on the shoulders of giants.
—Sir Isaac Newton

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Many people have helped me over the years to reach this point, for which I am grateful.

I would like to thank first my advisors, Professors Geoff Voelker and Stefan Savage, with whom I have had the privilege of working for the past seven years. They have been the source of countless good research ideas, and at the same time have been extremely supportive when I've wanted to head off in a new direction. Their feedback has improved my research, papers, and talks, and without them I wouldn't be where I am today. On top of this, working with them has been enjoyable and about as low-stress as grad school can be; it's difficult to imagine a better set of advisors.

I'd like to thank the rest of my committee as well—Bill Lin, Paul Siegel, and Amin Vahdat—for spending the time to work with me and give me feedback.

The Systems and Networking group forms a great community. Thanks to all the faculty—Geoff, Stefan, Alex, Amin, George, Joe, Keith, YY—and the other students. My officemates over the years have provided a fun and enjoyable environment in which to work: David Moore, Colleen Shannon, Diane Hu, Barath Raghavan, Justin Ma, Patrick Verkaik, Alvin AuYoung, Ryan Braud, Andreas Pitsillidis, Damon McCoy, Peng Huang, Yang Liu, and David Wang.

Even though the work in this dissertation was mostly undertaken independently, during my time in grad school I've worked on many projects collaboratively. I've had a chance, at one point or another, to work with most of the SysNet faculty and a number of student co-authors: Jay Chen, Diwaker Gupta, Chip Killian, Sangmin Lee, Justin Ma, John McCullough, David Moore, Nabil Schear, Erik Vandekieft, and Qing Zhang.

The system administrators for the Systems and Networking group have helped to keep all the computer systems running, and importantly have also helped me out with numerous special requests—from troubleshooting under a deadline, to setting up special software and hardware configurations, and to helping to gather usage data about our systems for my research. A big thanks to Marvin McNett, Chris X. Edwards, Brian Kantor, and Cindy Moore. The administrative staff have been a great help as well, in particular Michelle Panik, Kim Graves, Paul Terry, and Jennifer Folkestad.

The ACM International Collegiate Programming Contest has been a big part of my grad school experience, first as a contestant and later as a student coach. I'm grateful for the faculty members who helped make that possible: Brad Calder at the start, Michael Taylor for taking over later, and Geoff Voelker for extra help all along the way.

Chez Bob has been fun (even if it did sometimes consume too much time), and thanks to everyone else who has helped to run that, but especially: Justin Ma, John McCullough, Mikhail Afanasyev, Nathan Bales, Tristan Halvorson, Neha Chachra, and Keaton Mowery.

I've enjoyed hiking over and exploring San Diego county with many people from the CSE department, including Kirill Levchenko, Ming Kawaguchi, Qing Zhang, and many others. My advisor Geoff Voelker has also been very supportive of getting outdoors (going so far as to lead hikes for the rest of the research group.)

There are far too many people from Harvey Mudd College, both faculty and students, to list here. I would, however, like to acknolwedge in particular Professor Geoff Kuenning—for, among many other things, contributing to my interest in storage research and for first introducing me to my (then future) grad school advisors. Thanks as well to all the friends who have kept in touch.

I'm undoubtedly forgetting to include many other people who have helped out, so apologies to those left off and thank you still.

And finally, but certainly not least, I'd like to thank my family. They have been immensely supportive, both up to and through grad school.

Chapters 2 and 3, in part, are a reprint of material as it appears in the article "Cumulus: Filesystem Backup to the Cloud" by Michael Vrable, Stefan Savage, and Geoffrey M. Voelker which appears in *ACM Transactions on Storage*, Volume 5, Issue 4 (December 2009). The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, has been submitted for publication as "BlueSky: A Cloud-Backed Filesystem for the Enterprise" by Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. The dissertation author was the primary investigator and author of this material.

VITA

2004	Bachelor of Science in Mathematics and Computer Science Harvey Mudd College
2007	Master of Science in Computer Science University of California, San Diego
2011	Doctor of Philosophy in Computer Science University of California, San Diego

PUBLICATIONS

Chip Killian, Michael Vrable, Alex C. Snoeren, Amin Vahdat, and Joseph Pasquale. “Brief Announcement: The Overlay Network Content Distribution Problem”. *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. “Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm”. *Proceedings of the 20th ACM Symposium on Operating System Principles*, Brighton, UK, October 2005.

Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. “XFI: Software Guards for System Address Spaces”. *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. “Difference Engine: Harnessing Memory Redundancy in Virtual Machines”. *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. “Cumulus: Filesystem Backup to the Cloud”. *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, San Francisco, CA, February 2009.

Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. “Difference Engine”. *USENIX ;login:*, 34(2):24–31, April 2009.

Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. “Cumulus: Filesystem Backup to the Cloud”. *USENIX ;login:*, 34(4):7–13, August 2009.

Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. “Cumulus: Filesystem Backup to the Cloud”. *ACM Transactions on Storage*, Volume 5, Issue 4 (December 2009).

Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. “Neon: System Support for Derived Data Management”. *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Pittsburgh, PA, March 2010.

Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. “Difference Engine: Harnessing Memory Redundancy in Virtual Machines”. *Communications of the ACM*, 53(10):85–93, October 2010.

ABSTRACT OF THE DISSERTATION

Migrating Enterprise Storage Applications to the Cloud

by

Michael Daniel Vrable

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor Stefan Savage, Co-Chair
Professor Geoffrey M. Voelker, Co-Chair

Cloud computing has emerged as a model for hosting computing infrastructure and outsourcing management of that infrastructure. It offers the promise of simplified provisioning and management, lower costs, and access to resources that scale up and down with demand. Cloud computing has seen growing use for Web site hosting, large batch processing jobs, and similar tasks. Despite potential advantages, however, cloud computing is not much used for enterprise applications such as backup, shared file systems, and other internal systems. Many challenges need to be overcome to make cloud suitable for these applications, among them cost, performance, security, and interface mismatch.

In this dissertation, I investigate how cloud infrastructure can be used for internal services in an organization, with a focus on storage applications. I show how to design systems to address the challenges of using the cloud by building two example systems. The first, Cumulus, implements file system backup to a remote cloud storage provider. With Cumulus I consider the constraints imposed by the interface to cloud storage, and how to work within those constraints to minimize the cost. The second system, BlueSky, is a shared network file server which is backed by cloud storage. BlueSky builds on ideas from Cumulus to reduce system cost. It relies on cloud storage for data durability, but provides good performance by caching data locally. I additionally study how file system maintenance tasks can be offloaded to the cloud while protecting the confidentiality and integrity of file system data. Together, these two systems demonstrate that, despite the challenges, we can bring the benefits of the cloud to enterprise storage applications.

Chapter 1

Introduction

Cloud computing has recently emerged as a new model for hosting computing resources such as servers and storage. Cloud computing can be seen as one means for outsourcing computer hardware, making it easier to build many new applications on top of the outsourced resources.

Hardware outsourcing has been around for some time: since at least the rise of the Web, Web hosting providers have offered servers for rent in managed data centers. One feature distinguishing cloud computing from earlier hosting, however, is the granularity at which resources are available and the responsiveness in acquiring resources. In the traditional hosting model, a customer might sign a contract to rent a server or servers for a month at a time. In the cloud computing model, servers are often billed by the hour and can be requested and released in a matter of minutes—thus, customers can quickly adjust the number of servers reserved to match demand.

To users, cloud computing offers a number of compelling advantages over building out and managing their own infrastructure. For one, it transforms large up-front purchases of hardware into more steady ongoing costs to rent needed resources. It also frees users from the concerns of managing hardware, including power, networking, cooling, and dealing with hardware failures.

With traditional provisioning, the user must either provision for peak load on the system—which wastes resources at less-busy times—or face periods when the system is overloaded. The elastic nature of cloud resources—the ability to quickly increase or decrease the storage and computational resources—means that, building on the cloud, the user can allocate and pay for exactly the resources required at any point in time. This makes the overall system potentially both cheaper and more scalable.

Cloud computing offers advantages for the provider as well. Providers can rent spare capacity in their already-existing data centers to earn additional profit. Due to economies of scale, large providers can also add additional capacity to their cloud operations; the result can be a net win for both the provider and the smaller customers (who cannot build and operate the resources more cheaply than the large provider).

The term “Utility Computing” is sometimes used to refer to cloud computing, as an analogy with traditional utilities such as electric companies. In both cases, the utility makes a resource (electricity, computing) available to customers. Customers consume resources as needed, and are billed based on the quantity consumed.

Cloud computing offerings fall along a spectrum defined by the level of abstraction at which resources are offered. At one extreme is Infrastructure as a Service (IaaS). This is closest to the traditional hosting model: with IaaS users rent resources that are very close to the underlying hardware. For example, Amazon’s Elastic Compute Cloud (EC2) sells virtual machines (VMs) by the hour, and the Amazon Elastic Block Store (EBS) offers virtual hard drives for these VMs. On top of these resources, customers manage the VMs and install whatever software they require.

Platform as a Service (PaaS) operates at a slightly higher level of abstraction. More management tasks may be handled by the provider—such as deployment of code written by the customer, or management of the number of machines needed. Google’s AppEngine [18] falls into this category; customers write Python code to handle requests but Google manages all deployment. This simplifies the customer’s work but forces the customer to write software using a specific framework, which limits flexibility.

Finally, Software as a Service (SaaS) operates at the highest level of abstraction. Here, the provider sells pre-packaged applications rather than computational building blocks. This maximizes ease-of-use: the provider handles all the details and management of both the hardware and the software. Access to the application is often (though not necessarily) through a Web browser so that client machines need no modifications to access the software.

Offerings at higher levels of abstraction (SaaS) provide simpler deployment and management for users. However, IaaS and PaaS offerings have numerous benefits as well. They have far more flexibility, so customers can build whatever applications they need rather than being limited to what the provider offers. Furthermore, there is much less vendor lock-in at the lower levels: all IaaS providers offer more or less the same features, so it is much easier to move from one provider to another. A user of SaaS products may have much less choice.

In my work, I focus on applications built on top of Infrastructure- and Platform-as-a-Service offerings, both because those offer additional flexibility and because portability between providers (and the resulting competition) should result in lower costs for IaaS/PaaS offerings.

Cloud infrastructure can also be categorized separately as public, private, or hybrid [4]. Public clouds are the model described earlier, where a large provider (Amazon, Microsoft, etc.) makes the infrastructure available to the general public. With a private cloud, a company uses its own internal systems to build a cloud; the company does not get the benefits of hardware outsourcing but makes the same cloud APIs available to internal applications, so that internal applications can scale up and down on the company's infrastructure (though some people prefer not to use the term "cloud" to refer to these smaller-size datacenters). A hybrid cloud combines aspects of public and private clouds: a company may build a compute cloud own its own infrastructure, but applications can allocate resources either internally or from a public provider, as needed—for example, using internal resources most of the time but using a public provider to handle peaks in demand.

1.1 Cloud Computing Applications

Many companies have built successful applications on top of cloud computing infrastructure. Numerous Web startups have built on top of public cloud infrastructure instead of managing all hardware themselves. Early examples include Animoto (video rendering) and SmugMug (photo storage). Companies have also used cloud compute infrastructure for running large batch computations, especially when those computations are one-off jobs that do not justify investment in a large cluster that would see repeat use. One early example was the New York Times, which used Amazon EC2 to process scanned images from their newspaper archives. These examples have the characteristic that the workloads are either very unpredictable (based on the popularity of a startup's product) or bursty (batch processing workloads that require a large amount of processing power, but only for a short period of time).

The cloud has seen less use for other applications. One area where clouds are not much used is what I refer to as "enterprise storage applications". These are applications primarily used internally in an organization, and not by the organization's customers. Examples of these are file system backup tools, shared network file systems, e-mail systems, and internal business applications. Software-as-a-Service providers have made some progress with e-mail and business applications, but for many others organizations still mostly manage their own infrastructure.

The cloud has the potential to provide benefits for these storage applications. Building on the cloud greatly reduces the need for ahead-of-time capacity planning: the cloud can provide as much storage as is needed, when it is needed. The cloud provides durability for data: data written to the cloud is replicated by a cloud provider for reliability, so the chance of data loss can be much lower than with storage managed in-house. Finally, eliminating the need to manage reliable and scalable hardware locally can reduce hardware costs and ongoing management costs.

Yet the cloud has not seen much use, despite these benefits. There are many possible reasons, including performance, cost, security concerns, and the need to support legacy systems within the organization.

Performance can be an issue when migrating to the cloud. Services running in an organization can be accessed over a fast local-area network. Access to data and services in the cloud requires communication across the Internet which increases latency. Depending on network connectivity, bandwidth may be limited as well. Achieving good performance in spite of network conditions can require changes to the application.

When migrating applications to the cloud, cost is an important concern. The cloud can transform large up-front capital costs into ongoing operating expenses, and potentially leads to savings—but applications must be written with the cloud provider's cost schedule in mind or these savings can be lost. The cost optimizations needed for an application running on the cloud might not be the same as the optimizations that would be made for an application running locally.

Organizations are concerned about data security in the cloud. Data stored in internal applications is often quite sensitive. Cloud providers implement layers of security to protect data from unauthorized access, and have an incentive to do so correctly to protect their reputations. But for very sensitive data, an organization may want stronger assurances than simply the provider's word that data is kept safe from unauthorized access and tampering.

Finally, organizations are often constrained by legacy computer systems that are not easy to replace. When possible, migrating services to the cloud should not require extensive reconfiguration of these older systems.

1.2 Contributions

In this dissertation I investigate how cloud infrastructure can be used for internal services in an organization, with a focus on storage applications. I show how to design systems to address the issues of cost, performance, security, and legacy systems. I look at two particular instances of the more general problem of storage applications: file system backup and shared network file systems.

Cumulus implements file system backup by storing a backup copy of file data with a cloud storage provider. Because backup can run in the background, performance is less of a concern but the issues of cost and security are critical. With Cumulus, I show how to construct cost-effective backup to the cloud; the design used in Cumulus differs from most other backup systems due to the unique constraints of the cloud. In particular, I show that Cumulus can achieve a cost within a few percent of what could be achieved even given complete freedom to redesign the cloud provider's interface.

BlueSky, the second system, is a network file system backed by cloud storage. My design for BlueSky builds on ideas from Cumulus, but a network file system brings new challenges. The BlueSky design allows storage for a network file system to be migrated to the cloud with a minimum of reconfiguration on legacy clients. Access patterns for a file system are different from backup, and clients reading and writing data in BlueSky care about file system performance. I show that BlueSky can achieve performance competitive with local file servers in many cases, depending upon how effectively data can be cached locally. As with Cumulus, I design BlueSky to reduce costs charged by the cloud provider. BlueSky also allows file system maintenance tasks to be run in the cloud, while protecting data security guarantees.

Taken together, these two systems demonstrate how enterprise storage applications can be migrated to the cloud in a way that maintains security, achieves good performance, and minimizes cost.

1.3 Organization

The remainder of this dissertation is organized as follows:

Chapter 2 covers background material useful for understanding the remainder of the dissertation and discusses related work.

Chapters 3 and 4 present the two systems which form the core of this work. Chapter 3 describes Cumulus, which implements file system backup to a cloud storage provider. After briefly describing the problem, I explain the design used in Cumulus to solve it. Then, I describe the implementation of Cumulus in more detail and evaluate the performance of Cumulus using simulations and the actual prototype. In Chapter 4 I describe BlueSky, which extends the ideas in Cumulus to build a complete network file server. I first lay out the high-level architecture of BlueSky, which includes a local proxy to cache file system data. Then, I explain the data layout in the cloud and the design and implementation of the proxy in more detail. Following this, I measure and present an evaluation of the performance of BlueSky before concluding.

Finally, Chapter 5 summarizes my work and discusses directions for future research.

Chapter 2

Background

In this chapter I set the context for the remainder of the dissertation by providing more detailed background information. First, I describe the functionality offered by cloud service providers and discuss the two features on which my work builds—storage and compute nodes—in more detail. Then, I discuss enterprise storage applications and explain the functionality needed in backup and shared network file systems.

2.1 Cloud Providers

A number of infrastructure-as-a-service providers exist. Several of the largest are Amazon Web Services [2], Microsoft Windows Azure [30], and Rackspace Cloud [37]. There are some minor differences between the interfaces and services provided by each, but broadly speaking all offer the same basic services.

The basic services offered by all infrastructure providers include storage (structured as a key-value store) and compute nodes of some kind. Other common services include database-like services such as Amazon's SimpleDB (suitable for storing data and running simple queries, but not a full SQL database), job queueing services (for coordinating jobs to execute in a failure-tolerant way), network load balancers, and monitoring tools.

In my work, I focus almost entirely on cloud storage and raw computation services, described in further detail below. These are sufficient for building the applications I need, and most of the other services are useful primarily for scaling computation up, which is not as necessary for simpler enterprise storage applications.

2.1.1 Cloud Storage

The cloud providers mentioned all provide a simple key/value storage interface for storing arbitrary amounts of data. This storage interface does *not* behave like a standard file system. The namespace is not hierarchical: names used for storage can include a slash which appears to act like a directory separator, but the storage provider does not in fact treat the slash specially. Only a small set of operations is supported, and consistency guarantees are weaker than most normal file systems.

The base set of operations supported by cloud storage providers includes:

- **put:** Store a piece of data under a given key (name). The key can be any short string. The value is any sequence of bytes, from only a few bytes up to several gigabytes in size. Some providers allow for even larger files, but for maximum portability users should assume a maximum file size of 2–4 GB. The put operation atomically writes the data to the specified key (meaning a client never see a partially-written item); if an old value exists it is overwritten. Partial updates to an object are not allowed: to change an item the value must be entirely rewritten. Usually, additional metadata can be attached (key/value pairs that will be returned with the object headers when fetching an object).

- **get:** Retrieve the data stored under a given name. A client can request just a specific (contiguous) range of bytes rather than the entire object. Returns any metadata stored with the object as well.

- **list:** Return a list of keys stored on the server. The client can request only keys starting with a given prefix. If there are many matches, the provider will return a subset of the matches and the client can request additional items in subsequent requests. The client can ask for a specific character to be treated as a delimiter; this allows the client to get a listing hierarchically as if keys were stored as files grouped in directories.

- **delete:** Delete the specified object from the storage provider.

Some providers offer extensions to this basic interface. Windows Azure Blob Storage, for example, allows objects to be uploaded in pieces which allows interrupted uploads to be restarted and permits multiple parallel uploads of a single object. However, in my work I only rely on the above operations as doing so allows for maximum portability between providers.

Most commonly, clients use a RESTful [15] interface—based on HTTP—to interact with the storage provider. Read operations are simply HTTP GET operations; one side effect of this interface is that Web browsers can transparently access publicly-readable objects, so the cloud provider can be used to host static content. The other operations map to HTTP operations of the same name.

Operations sent to the storage provider are authenticated with an additional HTTP header containing a hash-based message authentication code based on a secret shared between the client and provider. Authenticating a request as coming from a client's account allows for access control and appropriate billing. If the client wishes to protect the privacy of data sent across the network, HTTPS can be used.

A simple example of interaction with a cloud provider (Amazon S3) is shown in Figure 2.1. The first two lines of each request specify the key being acted on and the account/container being accessed. Authentication for the message is included in the "Authorization: AWS" header; this header includes both an account ID and a signature computed over the other fields in the headers. A date header protects against replay attacks, where an attacker could resubmit an operation at a later time. The "Content-Type" header in the PUT request is stored as metadata with the object, and returned with later GET requests. Amazon S3 overloads the meaning of the "ETag" header to store an MD5 message digest of the object contents.

While cloud providers offer good data durability guarantees they offer only weak data consistency guarantees. Amazon provides *eventual consistency* [51]: the storage system guarantees that if no new updates are made to the object eventually all accesses will return the last updated value. Eventual consistency allows a number of visible anomalies: if an object is overwritten, then for some period of time future reads might return either the old data or the new data. For example, a write might be sent to one data center while some reads might fetch the old value from a different data center.

```
PUT /key HTTP/1.1
Host: mvrable-bluesky.s3.amazonaws.com
Authorization: AWS AKIAJ4RG3GWBMGEBRRXQ:WIfle+hyiHYpAR7JzseGVIEri58=
content-length: 6
content-type: binary/octet-stream
x-amz-date: Mon, 02 May 2011 03:58:12 +0000

value

HTTP/1.1 200 OK
Date: Mon, 02 May 2011 03:58:13 GMT
ETag: "756c412732c9e787f483d35d939b8ef2"
Content-Length: 0
Server: AmazonS3

GET /key HTTP/1.1
Host: mvrable-bluesky.s3.amazonaws.com
Authorization: AWS AKIAJ4RG3GWBMGEBRRXQ:FftZU5r33I+pvB5Z9qzMfhHC61I=
x-amz-date: Mon, 02 May 2011 03:58:19 +0000

HTTP/1.1 200 OK
Date: Mon, 02 May 2011 03:58:20 GMT
Last-Modified: Mon, 02 May 2011 03:58:13 GMT
ETag: "756c412732c9e787f483d35d939b8ef2"
Accept-Ranges: bytes
Content-Type: binary/octet-stream
Content-Length: 6
Server: AmazonS3

value

DELETE /key HTTP/1.1
Host: mvrable-bluesky.s3.amazonaws.com
content-length: 0
Authorization: AWS AKIAJ4RG3GWBMGEBRRXQ:K/0AhThbis+9NLLICw2eeOwPPzc=
x-amz-date: Mon, 02 May 2011 03:58:29 +0000

HTTP/1.1 204 No Content
Date: Mon, 02 May 2011 03:58:30 GMT
Server: AmazonS3
```

Figure 2.1: Example of communication with Amazon's S3 storage service using the RESTful API. A slightly simplified transcript of communication is shown. Requests from the client are shown in normal text, with responses from the provider in bold.

Table 2.1: Prices for cloud storage providers as of April 2011

	Amazon	Azure	Rackspace
Storage ($/GB/month)	0.14	0.15	0.15
Bandwidth In ($/GB)	0.10	0.10	0.08
Bandwidth Out ($/GB)	0.15	0.15	0.18
Put Request ($/req)	10^{-5}	10^{-6}	0*
Get Request ($/req)	10^{-6}	10^{-6}	0*

* Rackspace Cloud previously charged for requests over 250 KB in size, but in February 2011 eliminated all per-request charges.

Eventually the write will propagate to all data centers and reads will be consistent, but this takes an indeterminate amount of time. In contrast, a client writing to disk or a file system does not see these types of inconsistencies: once data has been synchronized to disk, subsequent reads will not see stale data.

The simple interface offered by cloud storage providers and the weak consistency guarantees let providers more easily build scalable storage systems. However, these same properties make the cloud storage interface unsuitable for direct use as a file system. Projects such as s3fs [38] do map cloud storage directly to a mountable file system but cannot provide all file system semantics that clients expect. Operations such as atomic renames of directory trees simply cannot be implemented, and eventual consistency is very different from the stronger consistency of local file systems.

Pricing

Cloud providers all have well-defined cost structures for data storage. Table 2.1 shows the costs for three different storage providers (Amazon S3, Widows Azure Blob Storage, and Rackspace Cloud Files). Costs are broken down into three main categories: storage, bandwidth, and per-operation charges.

Storage costs are measured per gigabyte of storage consumed on a monthly basis and are nearly identical across all providers. Amazon offers multiple pricing tiers based on the quantity of storage used; the table only shows the costs for the first tier. For Amazon S3, prices drop as low as $0.055/GB/month when storing 5 PB of data or more. In my work, however, I use prices for the lowest usage tiers since these represent worst-case prices.

Network bandwidth charges cover the cost of transferring data to or from the cloud provider. Different charges apply for inbound and outbound transfers. Transferring data to the cloud provider is, at least at present, cheaper than transferring data out. (This might reflect the fact that, if the cloud provider is used for any type of hosting there will be more traffic out of the cloud provider than in, and so the inbound bandwidth will be less utilized and thus cheaper.) Unlike storage costs which are quite uniform across providers, network transfer costs vary more widely.

Additionally, the customer's Internet service provider may charge for bandwidth used, though usually in a less direct fashion. If a customer is making heavy use of cloud storage and transferring large amounts of data, the customer may need to pay more to the ISP.

Finally, most cloud providers have some type of per-request charge for operations performed. In the case of Amazon S3 requests are broken down into two categories: cheap operations (GET requests) and expensive operations (PUT and LIST). These per-operation charges create a lower bound on the cost of an operation, helping to cover fixed costs at the provider (for example, the cost of a synchronous disk write or wide-area replication costs needed for a write operation). Dividing the per-request charge by the bandwidth charge gives a break-even point: the size at which equal parts of the cost of an operation are for the per-operation cost and the network transfer cost. For example, with Amazon S3 the PUT break-even point is around 100 KB. Writes smaller than this are not very economical since the per-request charges add up quickly. In contrast, costs are dominated by network transfers charges (and the per-operation charges are negligible) when writing values larger than a few times the break-even size. For reads, the break-even point is at 7 KB, so small reads are more cost-effective than small writes.

2.1.2 Cloud Computation

Different cloud providers offer different mechanisms for offloading computation to the cloud, at different levels of abstraction.

Several providers, including Amazon (with their Elastic Compute Cloud offering, or “EC2”) and Rackspace Cloud (with Cloud Servers) build their cloud computation offerings on top of Xen [5]. A customer receives a virtual machine that looks much like renting a private server. Virtual machines are available in a variety of configurations: differing amounts of memory, disk space, and available CPU cycles. Providers offer something very much like unmanaged hosting: a variety of pre-built software configurations are available, but all systems administration after the original setup is the responsibility of the customer. This allows for the simplest migration to the cloud, since the customer has full system administration privileges to configure the system. Most providers offer both Linux and Windows as options (with Windows costing slightly more).

Windows Azure offers managed operating system instances for specific roles: a “Web role” for responding to HTTP requests and a “Worker role” for background processing tasks. These roles do allow additional software to be installed and permit some customization. Some tasks, like operating system patching, are handled automatically. Windows Azure also offers a VM role which is more similar to what Amazon and Rackspace offer and allows for more customization if needed.

As with storage, providers have a published set of costs for cloud computation. Providers charge for each hour that a virtual machine runs, with a rate based on the hardware configuration. Network data transfer charges in and out of the data center are typically at the same rate as for storage transfer costs. Computation running in the cloud has the advantage that access to data in the cloud storage system does not have a data transfer charge, since access is within the data center and not over the wide area.

2.2 Enterprise Storage Applications

There are many enterprise applications that could potentially be moved to the cloud. These applications include e-mail and other communication tools, accounting, various types of business analytics, databases and applications built on top of them, and others. In this dissertation I focus on those applications that are storage-centric: those involving the management or sharing of data and not primarily the processing of that data. To explore this space, I design and build systems for file system backup and shared network file systems. Before describing the systems I have built, it is useful to look at those types of systems more broadly.

2.2.1 File System Backup

The goal of file system backup is to store a backup copy of all data stored on a computer system. These backups have multiple purposes. Backups are used to recover from a catastrophic data loss in which all data on the primary computer system is lost. A backup copy of data can also be used to recover old versions of selected files, for example if a user accidentally deletes or corrupts an important file and wants to recover the original data. The data loss may not be noticed immediately, so good backup systems will store multiple versions of files at different points in time. Also, because catastrophic data loss might be due to disasters that affect an entire data center (fire, natural disaster, etc.), good backups require storing a copy of the data somewhere completely off-site for safety.

Many traditional backup tools are designed to work well for tape backups. The dump, cpio, and tar [35] utilities are common on Unix systems and will write a file system backup as a single stream of data to tape. These backup utilities may create a full backup of a file system, 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. The backup tapes can be rotated to off-site storage locations, though recovery can be complicated by the need to locate the correct tapes.

Organizations 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.

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 and makes restores much more convenient. Many recent backup tools take advantage of this trend. Even more recently, increasing network capacity has meant that the disk or disks storing the backup data can be located at a remote site and accessed over a network. Cumulus, the work presented in this dissertation, falls into this category: it is a backup system that stores data over a network to disks located within a cloud storage provider.

2.2.2 Shared Network File Systems

Many businesses use network file systems to store data that should be made available to multiple computers and to give users a place to store data that is centrally-managed and backed up.

NFS [42] and CIFS [19] are the two protocols most widely used for implementing shared network file systems. NFS is widely used on Unix systems, while CIFS is the protocol that underlies Windows file sharing.

NFS

NFS, initially developed by Sun, comes in multiple versions. NFS version 2 is the earliest version that found widespread use; it is implemented on top of a remote procedure call (RPC) layer, where clients make procedure calls to the server to perform file system operations. By design, the server can be made almost entirely stateless. It need not track per-client state or information about open files—in fact, the server has no notion of files being "open" for access. When a client program opens and writes to a file, the NFS client will issue lookup operations to obtain a filehandle on which operations are made—but the filehandle merely identifies the file itself and tracks no per-client information. There is no notion of "closing" a filehandle after use. Filehandles are persistent, so that with a properly written NFS server, the server can crash and reboot during operation and the client will simply re-issue any lost requests and continue to function without any recovery code needed. For data durability even in the face of crashes, the NFS specification mandates that the server not reply to any client write operation until the updated data has been safely committed to disk.

The stateless design of the NFS server makes recovery from a server reboot simple, but has other implications. For one, since there is no per-client state the server cannot manage cache consistency for the clients. If client A makes file system updates, it must quickly flush those updates to the server. Otherwise, there is no way for client B to learn of the updates. Similarly, client B must repeatedly poll the server for changes to files. In practice, NFS clients will often cache file data but will revalidate this cached data by querying the server for the files' modification times. Clients usually implement close-to-open consistency, which guarantees applications that if client A writes to file, closes it, and client B then opens the file, then the changes will be visible: the first client must flush changes to the server before the close operation succeeds, and the second client must re-validate any cached data at each open request. Two clients that open a file concurrently and issue reads and writes may see inconsistent data.

These behaviors have several key performance implications:

- The mandate that data mutation operations commit synchronously to disk means that write-heavy workloads over NFS will often suffer from poor performance.
- Clients have frequent interactions with the server to maintain their (still limited) cache coherence. Each open operation requires a call to the server to check metadata, and each close requires flushing all updates to the server. Limited caching is possible across multiple accesses to the same file.

The NFS protocol version 3 makes a number of minor improvements and one more significant one. All operations that update file metadata remain synchronous, but the write operation is extended to allow uncommitted writes. In this case, a client can issue a number of file write operations in sequence, not waiting for the data to reach disk, and at a later point—though often when the file is closed—issue a commit operation to ensure the updates are safely on disk. The client must keep all updates buffered during this time, since if the server crashes before the commit the client must detect the crash and reissue the write operations to guarantee data durability.

NFS version 4 [44] is a major rewrite of the NFS protocol that actually makes it much more similar to CIFS. With NFSv4, the server does track file opens and closes, and mechanisms known as leases and delegations allow the client to more effectively cache data. NFSv4 has only recently been standardized, however, and is not very widely adopted. NFSv3 and NFSv2 remain much more common.

In my work on BlueSky described in this dissertation, I implement the NFS protocol but only version 3, not version 4.

CIFS

CIFS [19], the Common Internet File System, is the current basis for Windows filesharing. In contrast to NFS which (except for NFSv4) is a relatively simple protocol, Windows file sharing is extremely complex. Windows file sharing has evolved over time, with the result that there are multiple protocols and many dialects of each protocol that can be used.

In CIFS file open and close operations are sent to the server and the server is involved with locking and managing concurrent access to files. The need for the server to track per-client state increases server complexity. However, it also allows clients to achieve stronger file consistency guarantees.

Samba [41] is an open source implementation of CIFS (as well as other Microsoft networking protocols). Samba implements both client and server components and can interoperate with Microsoft products. As such, it is often used by Linux and MacOS computers to access Windows file shares, and as a way of setting up a Windows-compatible file server without having to run a Windows server. Samba implements a flexible architecture which makes it relatively easy to add a new storage backend for a Samba server.

As with NFS, the protocol as implemented is chatty, with frequent requests sent between client and server. While protocol chattiness is not a large issue in a local-area network, when operating over a wide-area network the need for many round-trip communications can cause large performance degradation. Thus, moving storage to the cloud should use, at the very least, some protocol besides NFS or CIFS for wide-area communication.

Chapter 2, in part, is a reprint of material as it appears in the article “Cumulus: Filesystem Backup to the Cloud” by Michael Vrable, Stefan Savage, and Geoffrey M. Voelker which appears in *ACM Transactions on Storage*, Volume 5, Issue 4 (December 2009). The dissertation author was the primary investigator and author of this paper.

Chapter 3

Cumulus

As we have seen, there are a range of architectures that all fall under the umbrella of the term “cloud computing”, ranging along a spectrum from highly integrated and focused (*e.g.*, Software-as-a-Service offerings such as Salesforce.com) to decomposed and abstract (*e.g.*, Infrastructure-as-a-Service offerings such as Amazon’s EC2/S3). Towards the former end of the spectrum, complex logic is bundled together with abstract resources at a datacenter to provide a highly specific service—potentially offering greater performance and efficiency through integration, but also reducing flexibility and increasing the cost to switch providers. At the other end of the spectrum, datacenter-based infrastructure providers offer minimal interfaces to very abstract resources (*e.g.*, “store file”), making portability and provider switching easy, but potentially incurring additional overheads from the lack of server-side application integration.

In this chapter, I begin to explore this trade-off between IaaS and SaaS (which I also refer to as *thin-cloud* vs. *thick-cloud*) in the context of a very simple application: file system 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 chapter I explore the other end of the design space—the "thin cloud". I describe a cloud-based backup system, called Cumulus, designed around a minimal interface (get, put, list, delete) that is trivially portable to virtually any on-line storage service. Thus, I assume that *any* application logic is implemented solely by the client. In designing and evaluating this system I make several contributions. First, I 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, I build a working prototype of this system using Amazon's Simple Storage Service (S3) and demonstrate its effectiveness on real end-user traces. Finally, I 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 chapter, I first describe prior work in backup, followed by a design overview of Cumulus and an in-depth description of its implementation. I then provide both simulation and experimental results of Cumulus performance, overhead, and cost in trace-driven scenarios. I conclude with a discussion of the implications of this work and how this research agenda might be further explored.

3.1 Related Work

As discussed in Chapter 2, traditional backup tools are designed to write data to magnetic tapes which can then be stored in an off-site location. Given the falling cost of disk, however, many more recent backup tools write backup data to disk.

Two approaches for comparing these more recent 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 and a variety of network protocols can be used.

Rsync [49] efficiently mirrors a file system 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 [13] 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 [36] 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 [56] 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 [48] 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 that used by cloud storage provider. Cumulus tries to be network-friendly like rsync-based tools, while using only a generic get/put storage interface.

Jungle Disk [24] can perform backups to Amazon S3. However, the design is quite different from that of Cumulus. Jungle Disk is first a network file system 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 [14] 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 [16] 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 by aggregation. 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 [10] 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 peer-to-peer systems as well, simply treating it as another storage interface offering get and put operations.

Table 3.1: Comparison of features among selected tools that back up to networked storage. Features considered are *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.

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

While other interfaces to storage may be available—Antiquity [53] 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 3.1 summarizes differences between some of the tools discussed above for backup to networked storage. In relation to existing systems, Cumulus is most similar to duplicity (without the need to occasionally re-upload a new full backup), and Brackup (with an improved scheme for incremental backups including rsync-style deltas, and improved reclamation of storage space).

3.2 Design

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

3.2.1 Storage Server Interface

We assume only a very narrow interface between a client generating a backup and a server responsible for storing the backup. Cumulus only relies on the standard set of operations supported by all cloud storage providers, as described in Section 2.1.1:

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.

In Cumulus we only require operations that work 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 in addition to all the cloud storage providers, it can be implemented on top of any number of other protocols: FTP, SFTP, WebDAV, 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.

3.2.2 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. File systems typically contain many small files (both our traces described later and others, such as [1], 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 3.4.4.

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 3.4.4.

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

Finally, as discussed in Sections 3.2.4 and 3.3.3, 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, re-aggregating this data becomes even more important when sub-file incremental backups are supported.

3.2.3 Snapshot Format

Figure 3.1 illustrates the basic format for backup snapshots. Cumulus snapshots logically consist of two parts: a *metadata log* which lists all the files backed up, 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

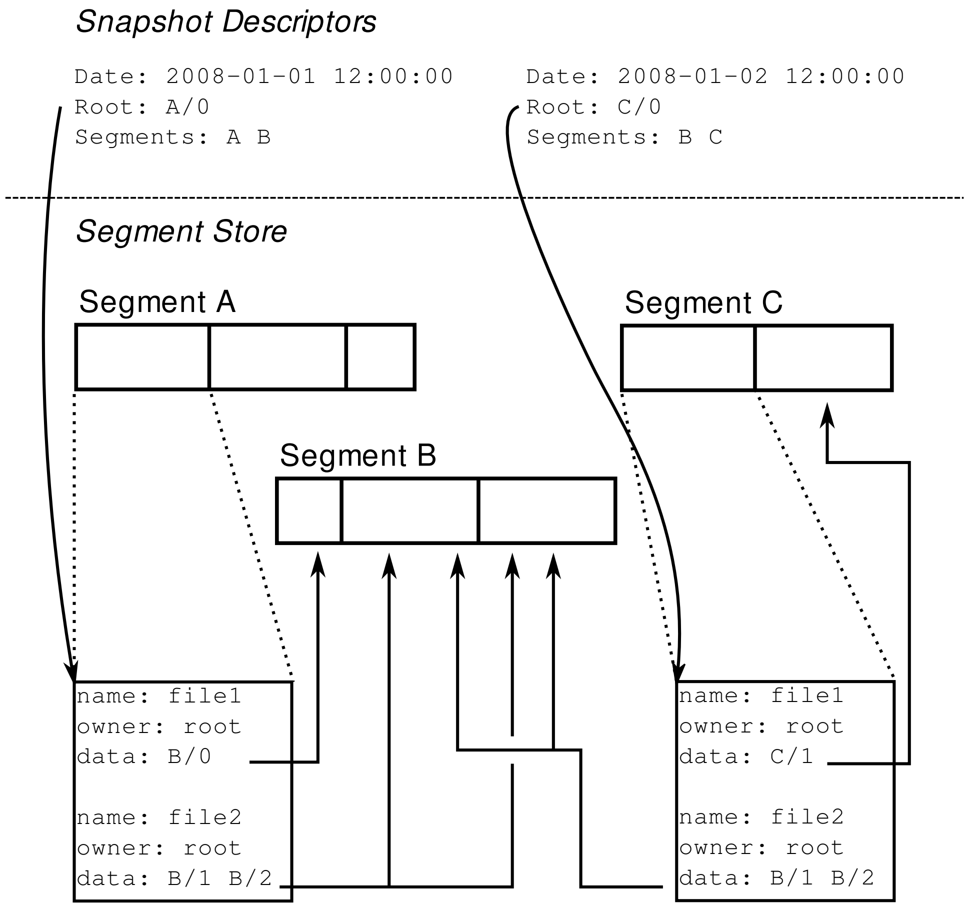


Figure 3.1: Simplified schematic of the basic format for storing Cumulus snapshots on a storage server. Two snapshots are shown, taken on successive days. Each snapshot contains two files. `file1` changes between the two snapshots, but the data for `file2` is shared between the snapshots. For simplicity in this figure, segments are given letters as names instead of the 128-bit UUIDs used in practice.

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.

3.2.4 Sub-File Incrementals

If only a small portion of a large file changes between snapshots, only the changed portion of the file should be stored. The 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 3.3.3.

3.2.5 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) [39].

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 [52]).

3.2.6 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.

3.2.7 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 de-duplication between backups from different clients. While Cumulus could use convergent encryption [11] to allow de-duplication 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 de-duplication 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.

3.3 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 [54]) implementing the core backup functionality, along with another roughly 1000 lines of Python for tasks such as restores, segment cleaning, and statistics gathering.

3.3.1 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 [45] 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 de-duplication. Enabling de-duplication leaves Cumulus vulnerable to corruption from a hash collision [20], [21], 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 3.2.3 and readable by our Cumulus prototype.

3.3.2 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.

3.3.3 Sub-File Incrementals

As discussed in Section 3.2.4, 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 [31].

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 de-duplication while maintaining backup format compatibility.

3.3.4 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 [7] library.

3.3.5 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 only 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 [17] interface that allows a collection of backup snapshots to be mounted as a virtual file system on Linux, thereby providing random access with standard file system 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.

3.4 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, sub-file 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.

3.4.1 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 3.2 summarizes the key statistics of each trace.

3.4.2 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.

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

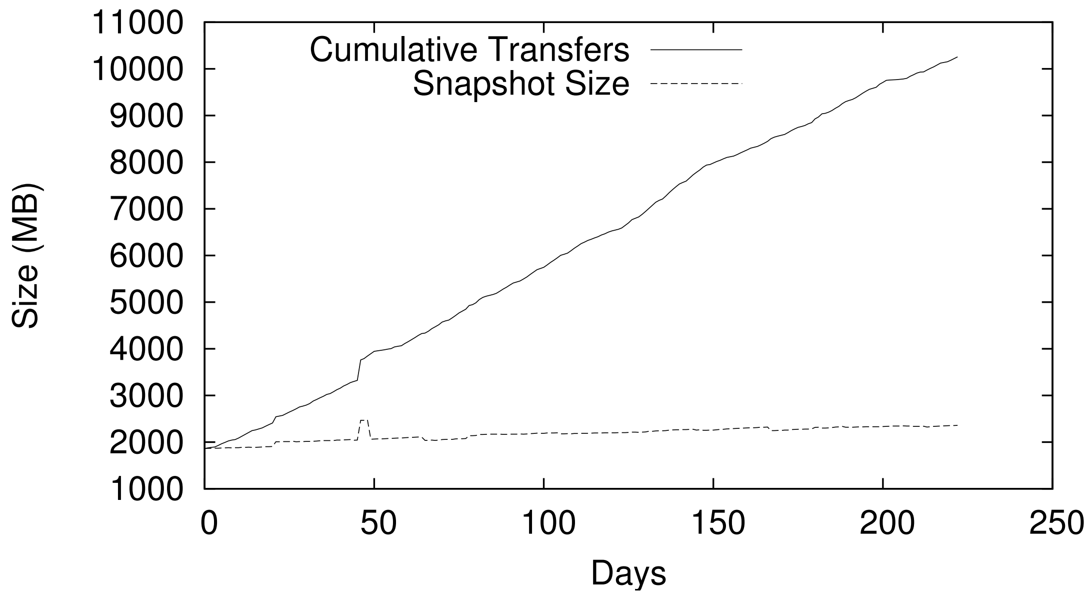
Table 3.2: Key statistics of the two traces used in the Cumulus evaluations. File counts and sizes are for the last day in the trace. "Entries" is files plus directories, symlinks, etc.

	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

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



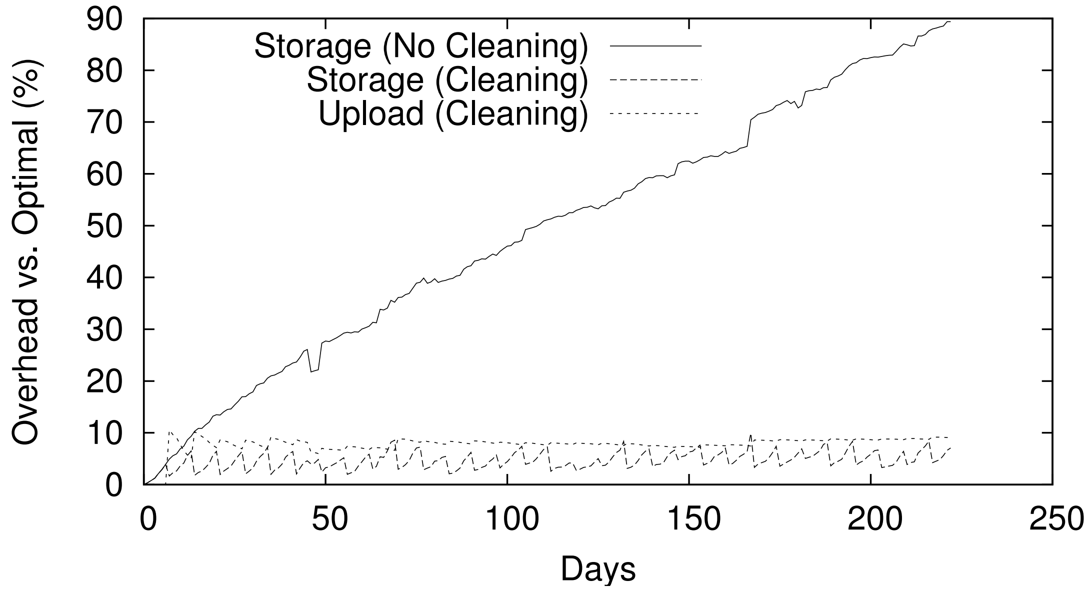


Figure 3.2: (a) Storage and network overhead for an optimal backup of the files from the user trace. (b) Overheads with and without cleaning; segments are cleaned at 60% utilization. Only storage overheads are shown for the no-cleaning case since there is no network transfer overhead without cleaning.

ing to a pre-determined backup schedule. In our experiments, we keep the most recent seven daily snapshots, with additional weekly snapshots retained going back farther in time so that a total of 12 snapshots are kept. 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 sub-file changes, compression, and metadata overhead when evaluating the prototype in Section 3.4.4.

Optimal Baseline

A simple storage interface for remote backup can incur an overhead penalty relative to more sophisticated approaches. To quantify the overhead of this approach, we use an idealized *optimal backup* as a basis of comparison.

For our simulations, the optimal backup is one in which no more data is stored or transferred over the network than is needed. Since simulation is done at a file granularity, the optimal backup will transfer the entire contents of a file if any part changes. 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 3.2.2 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 3.2(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.

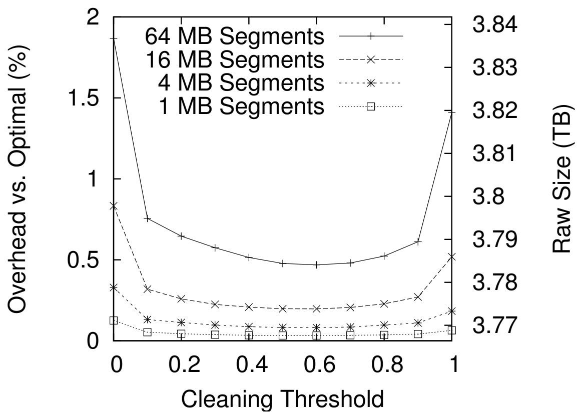
Figure 3.2(b) shows the results of two simulations of Cumulus backing up the same data. The graph shows the overheads relative to optimal backup; a backup as good as optimal would have 0% relative overhead. These results clearly demonstrate the need for cleaning when using a simple storage interface 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.

Cleaning Policies

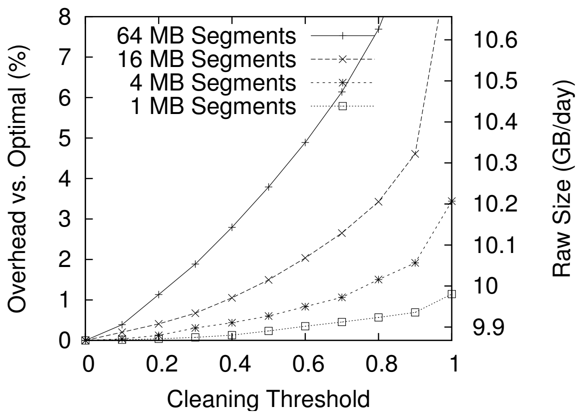
Cleaning is clearly necessary for efficient backup, but it is also parameterized by two metrics: the size of the segments used for aggregation, transfer, and storage (Section 3.2.2), and the threshold at which segments will be cleaned (Section 3.2.5). In our next set of experiments, we explore the parameter space to quantify the impact of these two metrics on backup performance.

Figures 3.3 and 3.4 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 (right y-axis). We use the backup policy of multiple daily and weekly snapshots as described in Section 3.4.2. 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 file system. Finally, *average segments per day* tracks the number of new segments uploaded each day to account for per-file upload and storage costs.

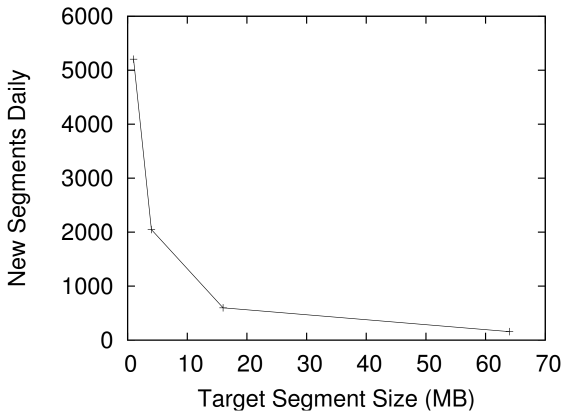
Storage and upload overheads improve with decreasing segment size, but at small segment sizes ($<$ 1 MB) backups require very large numbers of segments and limit the benefits of aggregating file data (Section 3.2.2). As expected, increasing the clean-



(a) Average daily storage

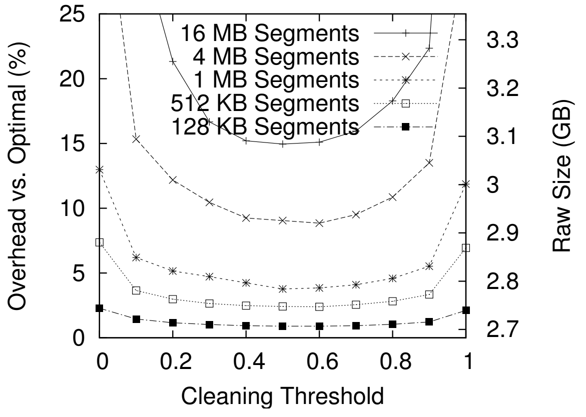


(b) Average daily upload

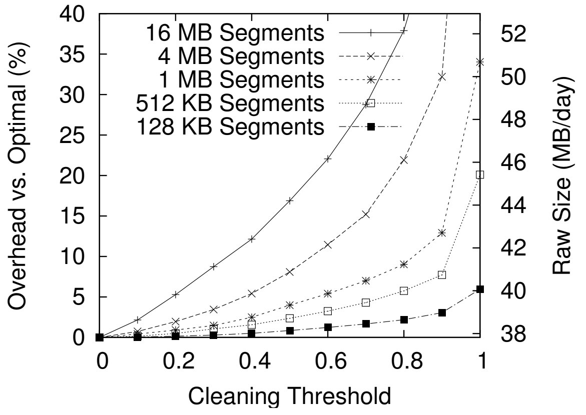


(c) Average segments per day

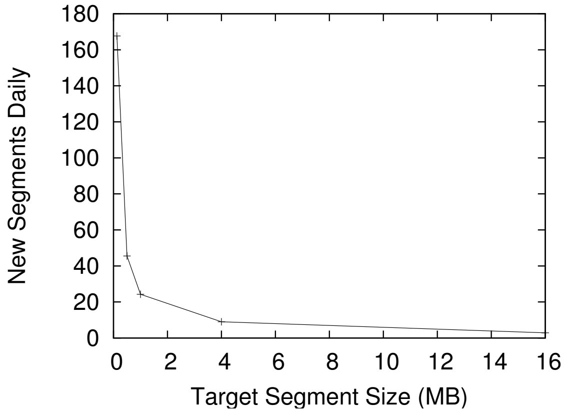
Figure 3.3: Overheads for backups in the fileserver trace



(a) Average daily storage



(b) Average daily upload



(c) Average segments per day

Figure 3.4: Overheads for backups in the user trace

ing 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.

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 3.3(a) and 3.4(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 3.5 explains why storage overhead increases again at higher cleaning thresholds. It shows three curves, the 16 MB segment size curve from Figure 3.3(a) and two curves that decompose the storage overhead into individual components (Section 3.2.5). 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.

3.4.3 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.

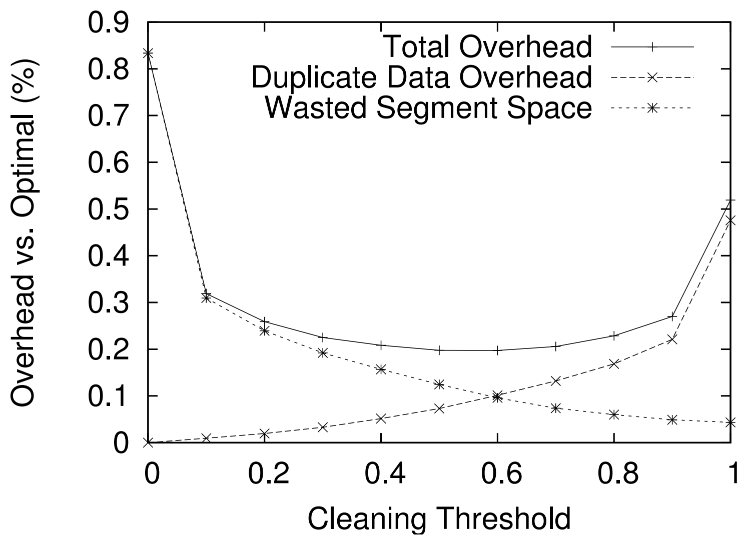


Figure 3.5: Detailed breakdown of storage overhead when using a 16 MB segment size for the fileserver workload

We use the prices for Amazon S3 as an initial point in the pricing space. In this evaluation we use the prices as of August 2009, which were (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, i.e., when uploading 100 KB files, half of the cost is for the bandwidth and half for the upload request itself. As the file size increases, the per-request component becomes an increasingly smaller part of the total cost.

Table 3.3 shows the monthly storage and upload costs for each of the two traces, neglecting for the moment the segment upload costs. 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

Table 3.3: Costs for backups in US dollars, if performed optimally, for the fileserver and user traces using current prices for Amazon S3

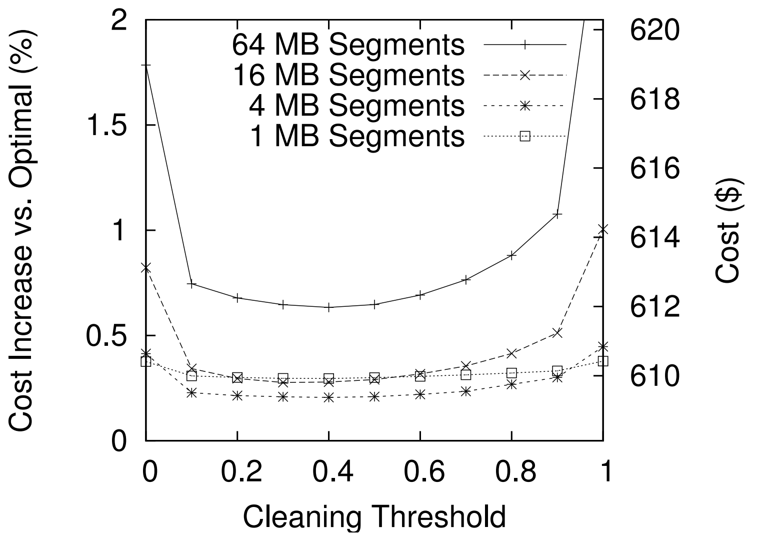
Fileserver	Amount	Cost
Initial upload	3563 GB	\$356.30
Upload	303 GB/month	\$30.30/month
Storage	3858 GB	\$578.70/month

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

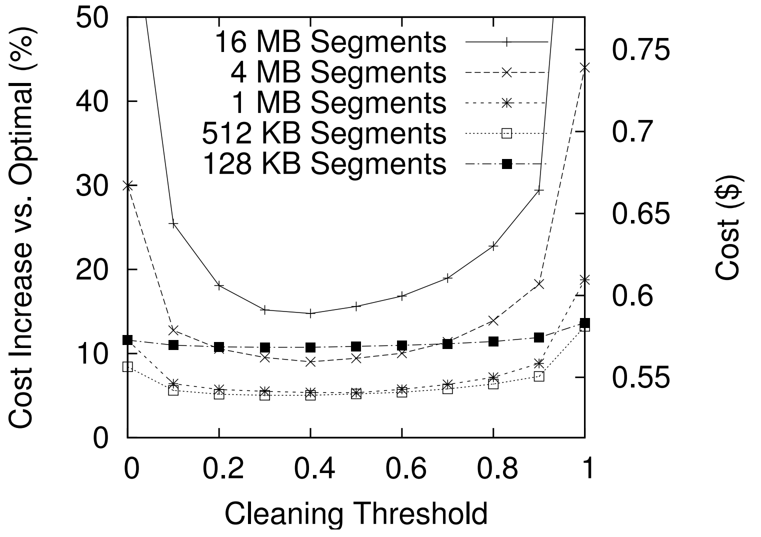
are roughly \$50.

Whereas Figure 3.3 explored the parameter space of cleaning thresholds and segment sizes in terms of resource overhead, Figure 3.6 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, from 0.5–2% for the fileserver trace depending on the parameters, and as low as about 5% in the user trace.

When evaluated in terms of monetary costs, 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 always decrease overall cost. At some point—in this case between 1–4 MB—decreasing segment size increases overall cost due to the per-file pricing. 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.



(a) Fileserver



(b) User

Figure 3.6: Costs in US dollars for backups in the fileserver assuming Amazon S3 prices

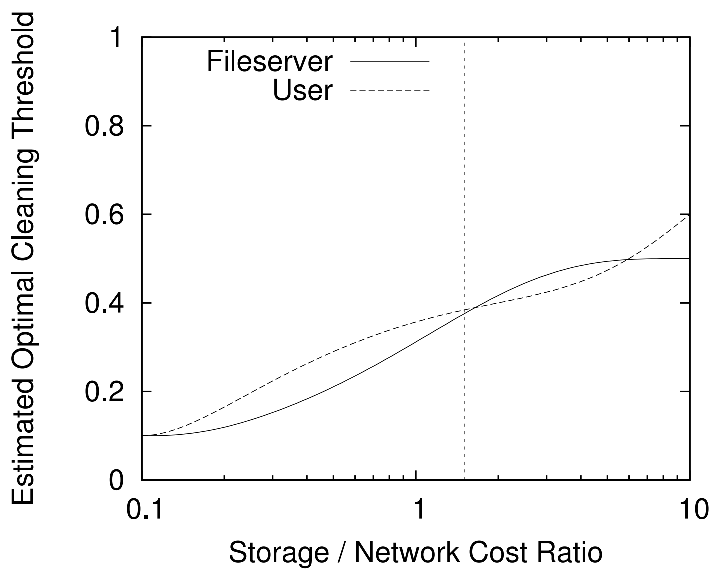


Figure 3.7: Sensitivity of the optimal cleaning threshold to the relative cost of storage vs. network. Amazon S3 currently has a storage/network cost ratio of 1.5, shown by a vertical line.

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 3.7 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.

3.4.4 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 sub-file incrementals and restores, and the time it takes to upload data to a remote service like Amazon S3.

System Comparisons: Non-Cloud 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 sub-file 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 3.8 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

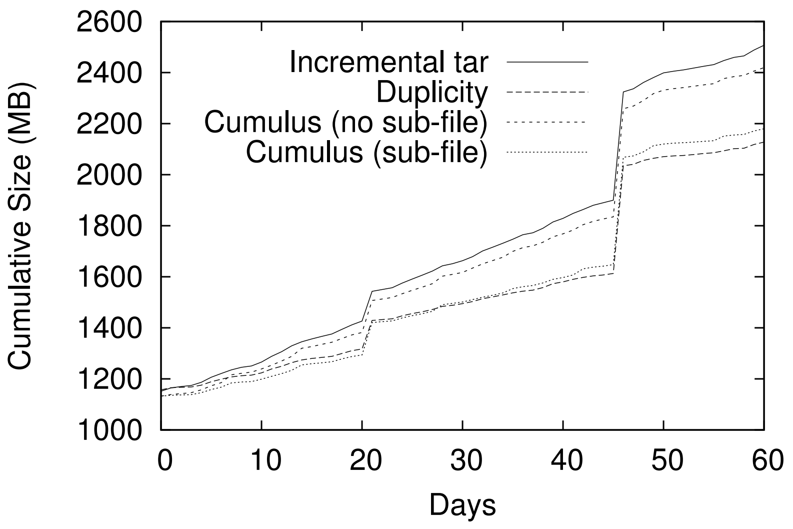


Figure 3.8: Cumulative storage costs for actual runs of multiple backup systems

mode of operation, on each successive day Cumulus backs up most data in all changed files. Duplicity performs better than the other two schemes by taking advantage of sub-file incrementals. Finally, when sub-file incremental backups are enabled in Cumulus, its daily upload rate falls so that it almost exactly matches that of duplicity.

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 sub-file incrementals, and 1048 MB with sub-file incrementals. Using duplicity as a proxy for near-optimal backups with sub-file 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.

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.

Table 3.4: 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 3.3 since that evaluation ignored the possible benefits of compression and sub-file incrementals, which are captured here. Values are listed on a per-month basis.

System	**Storage**	**Upload**	**Operations**
Jungle Disk	$\approx$ 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

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

Table 3.4 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 re-use metadata.

Comparing storage requirements of Cumulus with the average size of a full backup with the venerable `tar` utility, both are within 1%: storage overhead in Cumulus is roughly balanced out by gains achieved from de-duplication. Using duplicity as a proxy for near-optimal incremental backups, in a test with two months from the user trace Cumulus uploads only about 8% more data than is needed. Without sub-file incrementals in Cumulus, the figure is closer to 33%.

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.

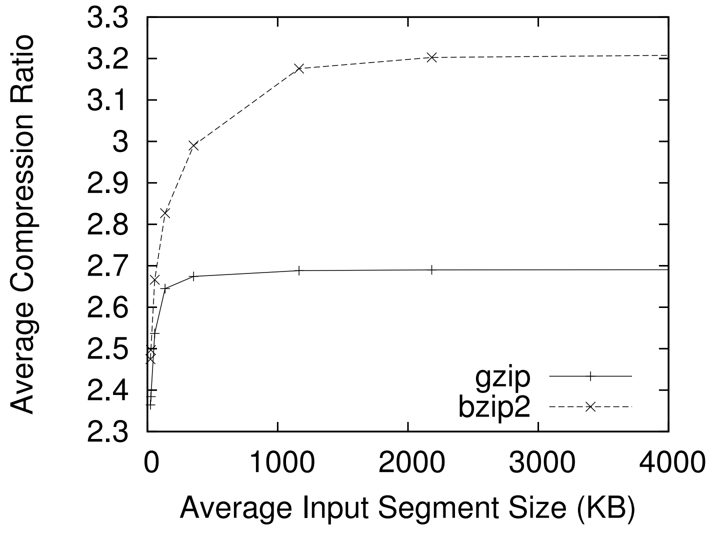


Figure 3.9: Measured compression ratio for packing small files together based on a sample of files from the user trace

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, 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 3.9 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 yield better compression, up to about 300 KB when using `gzip` and 1–2 MB for `bzip2` where compression ratios level off.

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.

Sub-File Incrementals

The effectiveness of sub-file 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 sub-file incrementals are small. In the user trace, more data is modified in-place and so the possible additional gains from sub-file incrementals are greater, as Section 3.4.4 showed.

To better understand the sub-file 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 sub-file updates.

To provide a point of comparison, we use `rdiff` [27] to generate an rsync-style delta between consecutive file versions. Table 3.5 summarizes the results.

The *size overhead* measures the storage cost of sub-file incrementals in Cumulus. To reconstruct the latest version of a file, Cumulus might need to read data from many

Table 3.5: Comparison of Cumulus sub-file 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%

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, sub-file 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.

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, they should not be taken as a good measure of the overall performance of S3 (in fact, our own measurements later show that performance has improved since then). Figure 3.10 shows

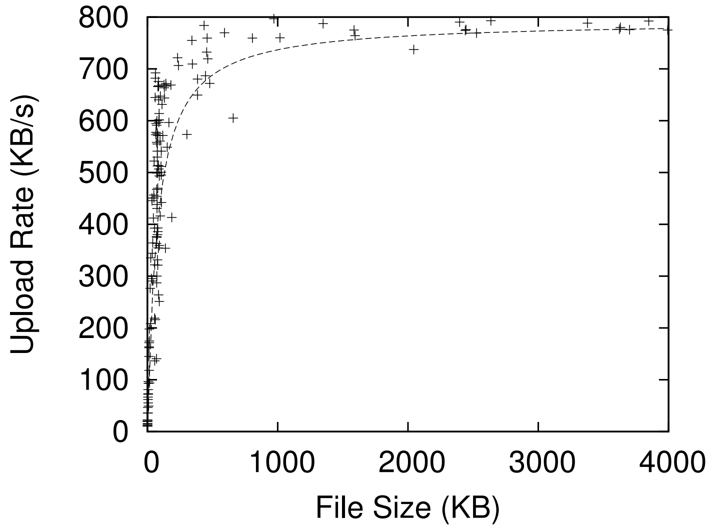


Figure 3.10: Measured upload rates for Amazon S3 as a function of file size. The measurements are fit to a curve where each upload consists of a fixed delay (for connection establishment and receiving a status code after the transfer) along with a fixed-rate upload.

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.

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.

3.5 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.

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 chapter has focused on an admittedly simple application, I 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? In this chapter I have shown that, at least for backup, a basic interface to commodity resources is sufficient. In the next chapter, I consider an even richer application: a full-fledged network file system.

Chapter 3, in part, is a reprint of material as it appears in the article “Cumulus: Filesystem Backup to the Cloud” by Michael Vrable, Stefan Savage, and Geoffrey M. Voelker which appears in *ACM Transactions on Storage*, Volume 5, Issue 4 (December 2009). The dissertation author was the primary investigator and author of this paper.

Chapter 4

BlueSky

In the previous chapter, I showed how a very simple interface to cloud storage can be used to build efficient file system backup to the cloud. In this chapter, I advance that work to an application beyond backup.

The promise of third-party "cloud computing" services is a trifecta of reduced cost, dynamic scalability, and high availability. While there remains debate about the precise nature and limit of these properties, it is difficult to deny that cloud services offer real utility—evident by the large numbers of production systems now being cloud-hosted via services such as Amazon's AWS and Microsoft's Azure. However, thus far, services hosted in the cloud have largely fallen into two categories: consumer-facing Web applications (*e.g.*, Netflix customer Web site and streaming control) and large-scale data crunching (*e.g.*, Netflix media encoding pipeline).

Little of this activity, however, has driven widespread outsourcing of enterprise computing and storage applications. The reasons for this are many and varied, but they largely reflect the substantial inertia of existing client-server deployments. Enterprises have large capital and operational investments in client software and depend on the familiar performance, availability and security characteristics of traditional server platforms. In essence, cloud computing it is not currently a transparent "drop in" replacement for existing services.

There are also substantive technical challenges to overcome, as the design points for traditional client-server applications (e.g., file systems, databases, directory servers, etc.) frequently do not mesh well with the services offered by cloud providers. In particular, many such applications are designed to be bandwidth hungry and latency sensitive (a reasonable design in a LAN environment), while the remote nature of cloud service offerings naturally increases latency and the cost of bandwidth. Moreover, while cloud services typically export simple interfaces to abstract resources (e.g., “put file” for Amazon’s S3), traditional server protocols can encapsulate significantly more functionality. Thus, until such applications are redesigned, much of the latent potential for outsourcing computing and storage services remains untapped. Indeed, at $115B/year, small and medium business (SMB) expenditures for servers and storage represent an enormous market should these issues be resolved [23].

In this chapter, we explore an approach for bridging these domains for one particular application: network file service. In particular, we are concerned with the extent to which traditional network file service can be replaced with commodity cloud services. However, our design is purposely constrained by the tremendous investment (both in capital and training) in established file system client software; we take as a given that end-system software will be unchanged. Consequently, we focus on a proxy-based solution, one in which a dedicated proxy server provides the *illusion* of a traditional file server, while translating these requests into appropriate cloud storage API calls over the Internet.

We explore this approach through a prototype system, called BlueSky, that supports both NFS and CIFS network file system protocols and includes drivers for both the Amazon EC2/S3 environment and Microsoft’s Azure. The engineering of such a system faces a number of design challenges, the most obvious of which revolve around performance (i.e., caching, hiding latency and maximizing the use of Internet bandwidth), but less intuitively also interact strongly with cost. In particular, as we will show, the interaction between the storage interfaces and fee schedule provided by current cloud service providers conspire to favor large segment-based layout designs (as well as cloud-based file system cleaners). We demonstrate that ignoring these issues can dramatically inflate costs (as much as $30\times$ in our benchmarks) without significantly improving performance. Finally across a series of benchmarks we demonstrate that, when using such a design, commodity cloud-based storage services can provide performance competitive with large dedicated file servers for the capacity and working sets demanded by enterprise workloads, while still accruing the scalability and cost benefits offered by third-party cloud infrastructures.

4.1 Related Work

Network storage systems have engendered a vast literature, much of it focused on the design and performance of traditional client server systems such as NFS, AFS, CIFS [19, 22, 42]. Recently, a range of efforts have considered other structures, including those based on peer-to-peer storage [32] among distributed sets of untrusted servers [26, 28] which have indirectly informed subsequent cloud-based designs.

Cloud storage is a newer topic, driven by the availability of commodity services from Amazon's S3. Perhaps the closest academic work to our own is SafeStore [25], which stripes erasure-coded data objects across multiple storage providers, ultimately exploring access via an NFS interface. However, SafeStore is focused clearly on availability, rather than performance or cost and thus its design decisions are quite different. A similar, albeit more complex system, is DepSky [6], which also focuses strongly on availability, proposing a "cloud of clouds" model to replicate across providers.

At a more abstract level, Chen and Sion create an economic framework for evaluating cloud storage costs and conclude that the computational costs of the cryptographic operations needed to ensure privacy can overwhelm other economic benefits [8]. However, this paper was written before the introduction of Intel's AES-NI architecture extension (present since the Westmere and Sandy Bridge processor generations) which significantly accelerates data encryption operations.

There have also been a range of non-academic attempts to provide traditional file system interfaces for the key-value storage systems offered by services like Amazon's S3. Most of these install new per-client file system drivers. Exemplars include s3fs [38], which tries to map the file system directly on to S3's storage model (which both changes file system semantics, but also can dramatically increase costs) and ElasticDrive [12], which exports a block-level interface (potentially surrendering optimizations that use file-level knowledge such as prefetching).

However, the systems closest to our own are "cloud storage gateways", a new class of storage server that has emerged in the last year (contemporaneous with our effort). These systems, exemplified by companies such as Nasuni, Cirtas, TwinStrata, StorSimple and Panzura, provide caching network file system proxies (or "gateways") that are, at least on the surface, very similar to our design. Pricing schedules for these systems generally reflect a $2\times$ premium over raw cloud storage costs). While few details of these systems are public, in general they validate the design point we have chosen.

Of commercial cloud storage gateways, Nasuni [33] is perhaps most similar to BlueSky. Nasuni provides a "virtual NAS appliance" (or "filer"), software packaged as a virtual machine which the customer runs on their own hardware—this is very much like the BlueSky proxy software that we build. The Nasuni filer acts as a cache and writes data durably to the cloud. Because Nasuni does not publish implementation details it is not possible to know precisely how similar Nasuni is to BlueSky, though there are some differences. In terms of cost, Nasuni charges a price based simply on the quantity of disk space consumed (around $0.30/GB/month, depending on the cloud provider)—and not at all a function of data transferred or operations performed. Presumably, Nasuni optimizes their system to reduce the network and per-operation overheads—otherwise those would eat into their profits—but the details of how they do so are unclear, other than by employing caching.

Cirtas [9] builds a cloud gateway as well but sells it in appliance form: Cirtas's Bluejet is a rack-mounted computer which integrates software to cache file system data with storage hardware in a single package. Cirtas thus has a higher up-front cost than Nasuni's product, but is easier to deploy. Panzura [34] provides yet another CIFS/NFS gateway to cloud storage. Unlike the others, Panzura allows multiple customer sites to each run a cloud gateway. Each of these gateways accesses the same underlying file system, so Panzura is particularly appropriate for teams sharing data over a wide area. But again, implementation details are not provided.

TwinStrata [50] offers a cloud gateway that presents an iSCSI [43] interface to storage—thus, TwinStrata appears like a disk array instead of a network file system to clients. This provides flexibility: any type of file server can be used in front of Twin-Strata's appliance, and it will work well for hosting virtual machine disk images as well. However, by operating at the block level instead of the file system level, TwinStrata's appliance misses semantic knowledge of the data being accessed which could lead to poorer caching. TwinStrata offers both physical appliances (like Cirtas) and virtual appliances (like Nasuni). StorSimple [47] is yet another company, offering a product similar to a TwinStrata appliance—iSCSI-accessible block storage backed by the cloud.

4.2 Architecture

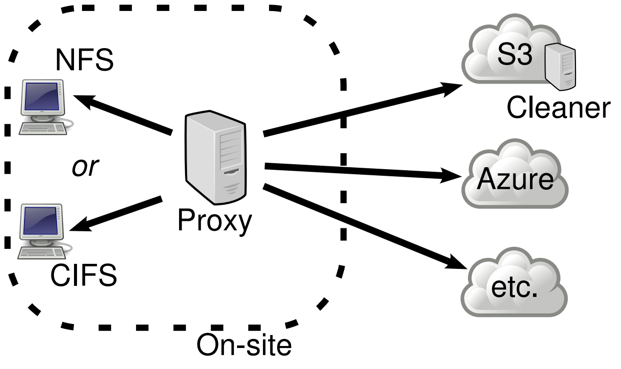
BlueSky provides service to clients in an enterprise using a transparent proxy-based architecture that stores data persistently on cloud storage providers (Figure 4.1). This section discusses the role of the proxy and cloud provider components, as well as the security model supported by BlueSky. Sections 4.3 and 4.4 then describe the layout and operation of the BlueSky file system and the BlueSky proxy, respectively.

4.2.1 Local Proxy

The central component of BlueSky is a proxy situated between clients and cloud providers. The proxy communicates with clients in an enterprise using a standard network file system protocol, and communicates with cloud providers using a cloud storage protocol. Our prototype supports both the NFS (version 3) and CIFS protocols for clients, and the RESTful protocols for the Amazon S3 and Windows Azure cloud services. Ideally, the proxy runs in the same enterprise network as the clients to minimize latency to them. The proxy caches data locally and manages sharing of data among clients without requiring an expensive round-trip to the cloud.

Clients do not require modification since they continue to use standard file-sharing protocols. They mount BlueSky file systems exported by the proxy just as if they were exported from an NFS or CIFS server. Further, the same BlueSky file system can be mounted by any type of client with shared semantics equivalent to Samba.

As described in more detail later, BlueSky lowers cost and improves performance by adopting a log-structured data layout for the file system stored on the cloud provider. File system objects are packed together into larger log segments which are then written to cloud storage. A cleaner reclaims storage space by garbage-collecting old log segments which do not contain any live objects, and processing almost-empty



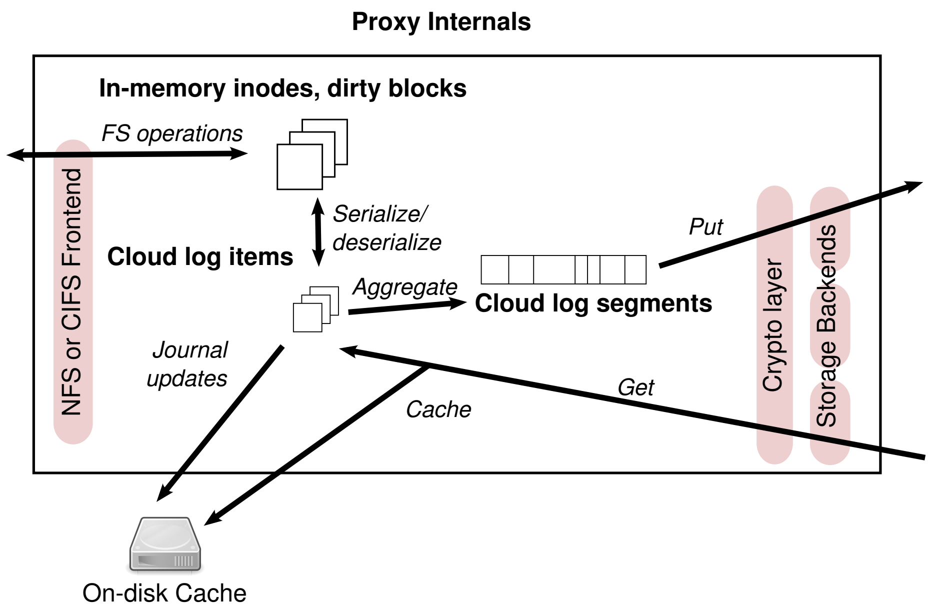


Figure 4.1: BlueSky architecture. A proxy running on-site mediates all access to the file system and provides local caching. Clients communicate with the proxy via NFS or CIFS. The proxy stores all data to a cloud provider eventually, using any of a number of possible backends. A log cleaner may execute in the cloud. The bottom half of the figure shows the internal organization of the proxy. File system operations are translated by a frontend and passed to the BlueSky core. Operations are initially performed on in-memory representations of inodes, but updates are quickly serialized into cloud log items. These log items are journaled to local disk for crash-recovery, and are aggregated together into log segments to be stored in the cloud. Log segments pass through a cryptographic layer to encrypt and add authentication data, then to a storage backend driver which communicates with the cloud. Cloud log items are fetched back from the cloud later if needed, and fetched items are cached on disk at the proxy.

segments by copying live data out of old segments into new segments.

As a write-back cache, the BlueSky proxy can fully satisfy client write requests with local network file system performance by writing to its local disk—as long as its cache capacity can absorb periods of write bursts as constrained by the bandwidth the proxy has to the cloud provider (Section 4.5.5). For read requests, the proxy can provide local performance to the extent that the proxy can cache the working set of the client read workload (Section 4.5.4). In both cases we argue that a proxy architecture like BlueSky can achieve both goals effectively for enterprise workloads.

4.2.2 Cloud Provider

So that BlueSky can potentially use any cloud provider for persistent storage service, it makes minimal assumptions of the provider; in our experiments, we use both Amazon S3 and the Windows Azure blob service as cloud storage providers. BlueSky assumes a basic interface supporting `get`, `put`, `list`, and `delete` operations. Using just that interface, BlueSky constructs and maintains a log-structured file system on the provider. If the cloud provider also supports a hosting service, BlueSky can co-locate the file system cleaner at the provider to reduce cost and improve cleaning performance.

4.2.3 Security

Security becomes a key concern with outsourcing critical functionality such as data storage. In designing BlueSky, our goal is to provide high assurances of data confidentiality and integrity. The proxy encrypts all client data before sending it over the network, so the provider cannot read private data. Data stored at the provider also includes integrity checks so that any tampering by the storage provider can be detected.

However, some trust in the cloud provider is unavoidable, particularly for data availability. The provider could always delete or corrupt stored data, rendering it unavailable. These actions could be intentional—*e.g.*, if the provider is malicious—or accidental, for instance due to insufficient redundancy in the face of correlated hardware failures from disasters. Ultimately, the best guard against such problems is through auditing and the use of multiple independent providers [25, 6]. BlueSky could readily incorporate such functionality, such as maintaining redundant versions of a file system on multiple cloud providers, but doing so remains outside the scope of our current work.

A buggy or malicious storage provider could also serve stale data—instead of returning the most recent data, it could return an old copy of a data object that nonetheless has a valid signature (because it was written by the client at an earlier time). By authenticating pointers between objects starting at the root, however, BlueSky prevents a provider from selectively rolling back file data. A provider can only roll back the entire file system to an earlier state, which is likely to be detected.

BlueSky can also take advantage of computation in the cloud for running the file system cleaner. As with storage, we do not want to completely trust the computational service, yet doing so provides a tension in the design. To maintain confidentiality, data encryption keys should not be available on cloud compute nodes. Yet, if cloud compute nodes are used for file system maintenance tasks, the compute nodes must be able to read and manipulate file system data structures. For BlueSky, we make the tradeoff of encrypting file data while leaving the metadata necessary for cleaning the file system unencrypted. As a result, storage providers can understand the layout of the file system and potentially corrupt data, but the data still remains confidential and the proxy can still validate its integrity.

In summary, BlueSky attempts to provide strong confidentiality guarantees and slightly weaker integrity guarantees (some data rollback attacks might be possible but are largely prevented), but must rely on the provider for availability.

4.3 BlueSky File System

This section describes the BlueSky file system layout. We present the object data structures maintained in the file system and their organization in a log-structured format. We also describe how BlueSky cleans the logs comprising the file system, and how the design conveniently lends itself to providing versioned backups of the data stored in the file system. Subsequently in Section 4.4 we describe how the BlueSky proxy provides file system service to clients.

4.3.1 Object Types

BlueSky uses four types of objects for representing data and metadata in its log-structured file system [39] format: data blocks, inodes, inode maps, and checkpoints. These objects are aggregated into log segments for storage. Figure 4.2 illustrates their relationship in the layout of the file system.

Data blocks store file data. Files are broken apart into fixed-size blocks (except for the last block in a file which may be short). BlueSky uses 32 KB blocks instead of typical disk file system sizes like 4 KB to reduce overhead: block pointers as well as extra header information impose a higher per-block overhead in BlueSky than in an on-disk file system. We briefly discuss the effects of varying the size later. Nothing fundamental prevents BlueSky from using variable-size blocks optimized for the access patterns of each file, but we have not implemented this.

Inodes for all file types include basic metadata (ownership and access control, timestamps, etc.). For regular files, the inode includes a list of pointers to data blocks with the file contents. Directory entries are stored within the directory inode itself to reduce the overhead of making path traversals. BlueSky currently does not use indirect blocks for locating file data—inodes directly contain pointers to all data blocks (easy to do since inodes are not fixed-size).

Inode maps list the locations in the log of the most recent version of each inode. Since inodes are not stored at fixed locations, inode maps provide the necessary level of indirection for locating inodes.

A checkpoint object determines the root of a file system snapshot. A checkpoint contains pointers to the locations of the current inode map objects. When first reading a file system, the proxy locates the most recent checkpoint simply by scanning backwards in the log, since the checkpoint is always one of the last objects written. Checkpoints are useful for maintaining file system integrity in the face of proxy failures, for decoupling cleaning and file service (Section 4.3.3), and for providing versioned backup (Section 4.3.4).

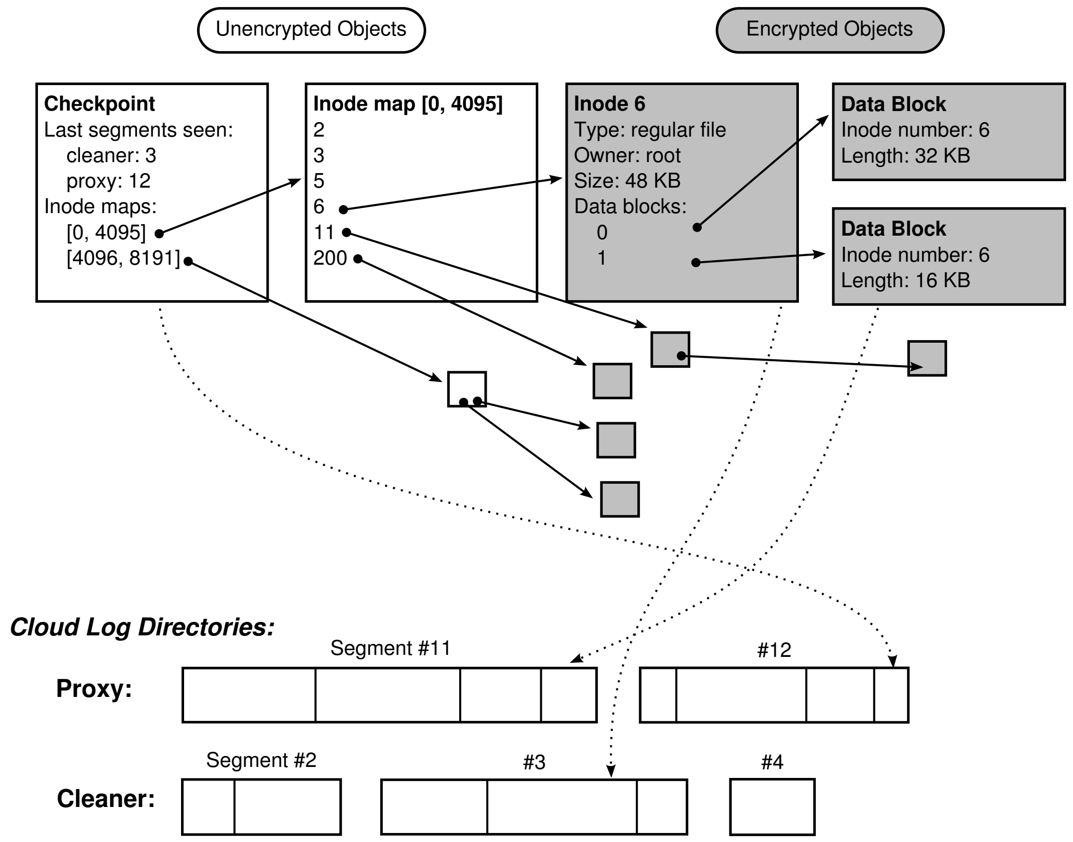


Figure 4.2: BlueSky filesystem layout. The top portion shows the logical organization, with checkpoints pointing to inode maps pointing to inodes pointing to data blocks. Object pointers are shown with solid arrows, and include both the unique identifier of the object and the physical location. Shaded objects are stored in encrypted form in the cloud for privacy; checkpoints and inode maps are unencrypted so that an in-cloud cleaner can read them. Object pointers are always unencrypted. The bottom half of the figure shows the physical layout, where log items are packed into segments and stored in the cloud. Dashed lines indicate the physical placement of objects. Note that pointers between log directories are permitted: for example, an inode in one directory pointing at a data block in another. Also note that the checkpoint object falls at the very end of the log, allowing the proxy to find it on startup.

4.3.2 Cloud Log

For each file system stored at a cloud service, BlueSky maintains a separate log for each writer to the file system. Typically, each file system has two writers: the proxy managing the file system on behalf of a set of clients and a cleaner that garbage collects overwritten data. Each writer stores its log segments to a separate directory, and as a result each writer can make updates to a file system independently and without coordination.

File system logs consist of a logical list of log segments. Log segments aggregate objects into large, approximately fixed-size containers for storage and transfer. In the current implementation segments are up to about four megabytes in size, large enough to avoid the overhead of dealing with many small objects. Writing large segments instead of many small objects substantially reduces the per-operation cost and makes better use of the network bandwidth between the proxy and the cloud service. Though the storage interface requires that each log segment be written in a single operation, typically cloud providers allow partial reads of objects (using HTTP byte-range requests). As a result, data can be read at the granularity at which it is requested, which is independent of the size of a segment. Section 4.5.6 quantifies the performance benefits of grouping data into segments and of selective reads, and Section 4.5.7 quantifies their cost benefits.

A monotonically-increasing *sequence number* identifies each log segment within a directory. Since each log segment may contain many objects, a byte *offset* identifies the specific object in the segment. Additionally, while not required a location pointer will usually include the size of the object as a hint for how much data to fetch to read the object. Together, the triple $(\textit{directory}, \textit{sequence number}, \textit{offset})$ describes the physical location of each object.

BlueSky generates a unique identifier (UID) for each object when that object is written into a segment in the log. This unique identifier remains unchanged if the object is simply relocated (without change) during the cleaning process.

An object can contain pointers to other objects (for example, an inode pointing to data blocks), and the pointer lists both the UID and the physical location of the object. The UID is integrity-protected but the location is not. Thus, a cleaner can reorganize data items and update the location pointers without needing any secret keys, but the

$$\textbf{Authenticated:}\quad \begin{cases} \text{Object type} \\ \text{Unique identifier (UID)} \\ \text{Inode number} \\ \textbf{Encrypted: } \left\{ \text{Object payload} \right. \\ \text{Object pointers: UIDs} \end{cases}$$

$$\textbf{Unauthenticated:}\quad \text{Object pointers: Physical locations}$$

Figure 4.3: Data fields included in most objects stored in the cloud

proxy can still verify the identity of the relocated object using the UID.

Data blocks list the inode number of the inode to which the data belongs. While not needed in normal operation, this backpointer can be used by a log cleaner to locate the possible owner of the data block and determine whether the data is still in fact in use.

In support of BlueSky's security goals (Section 4.2.3), file system objects are individually encrypted (with AES) and protected with a keyed message authentication code (HMAC-SHA-256) by the proxy before uploading to the cloud service. Each object can contain data with a mix of protections: some data is encrypted and authenticated, some data is authenticated plain-text, and some data is unauthenticated. The keys for encryption and authentication are not shared with the cloud, though we assume that a safe backup of these keys (which do not change) is kept for disaster recovery. Figure 4.3 summarizes the fields included in most objects.

4.3.3 Cleaner

As with any log-structured file system, BlueSky requires a file system cleaner to garbage collect data that has been overwritten. Unlike traditional disk-based systems, though, the completely elastic nature of cloud storage means that the file system will not run out of space, and so it is not necessary to run the cleaner to make progress writing out new data. Running the cleaner periodically will reclaim space to reduce ongoing storage costs, and can defragment data for more efficient access later.

We designed the BlueSky cleaner so that it can run either remotely at the proxy, or locally on a compute instance within the cloud provider where it has faster, cheaper access to the storage. For example, when running the cleaner in Amazon EC2 and accessing storage in Amazon S3, Amazon does not charge for data transfers (though it still charges for operations). We have also specifically designed the BlueSky file system so that a cleaner running in the cloud does not need to be fully trusted—it will need permission to read and write cloud storage, but does not require the file system encryption and authentication keys.

The cleaner runs online with no synchronous interactions with the proxy: clients can continue to access and modify the file system through the proxy even while the cleaner is running. The proxy and cleaner write to separate log directories in the cloud so that neither overwrites the log data of the other.

There may be concurrent updates to the same object—for example, the proxy may write to an inode while the cleaner relocates its data. In cases like these, the divergent inode versions in each log must be merged together. In BlueSky, the proxy takes all responsibility for merging divergent files together (Section 4.4.3).

4.3.4 Backups

The log-structured design allows BlueSky to integrate file system snapshots for backup purposes easily. In fact, so long as a cleaner is never run, any checkpoint record ever written to the cloud can be used to reconstruct the state of the file system at that point in time. Though not implemented in our prototype, the cleaner or a snapshot tool could record a list of checkpoints to retain and protect all required log segments from deletion. Alternatively, those segments could be archived elsewhere for safekeeping.

Because all file system data is already stored in the cloud, creating a backup snapshot in this way is a quick operation that does not require uploading all the data from the client site.

4.4 BlueSky Proxy

In this section we describe the design and implementation of the BlueSky proxy, including its approach to managing its data cache on disk, its network connections to the cloud, and how it indirectly cooperates with the cleaner.

4.4.1 Cache Management

The proxy uses its local disk storage to implement a write-back cache. The proxy logs file system write requests from clients (both data and metadata) to a journal on local disk. The proxy ensures that data is safely on disk before replying to clients that data has been committed. The proxy sends these updates to the cloud asynchronously in the background. Physically, the journal is broken apart into sequentially-numbered files on disk (journal segments) of a few megabytes each.

This write-back caching at the proxy does mean that in the event of a catastrophic failure of the proxy—if the proxy's storage is lost—that some data may not have been written to the cloud and will be lost. In the event of an ordinary crash in which local storage is intact, no data will be lost as the proxy will replay the changes recorded in the journal to reconstruct file system state. Periodically, the proxy takes a snapshot of the file system state, as it has been committed to the journal, collects new file system objects and any inode map updates into a number of log segments, and uploads those log segments to cloud storage. If a very large amount of data has changed, the changes are uploaded as several segments. (Our prototype proxy implementation does not currently perform deduplication, which we would likely do at the file level [29] but leave for future work for evaluating its tradeoffs.)

There are tradeoffs in choosing how quickly to flush data to the cloud. Writing data to the cloud quickly minimizes the window for data loss. However, a longer timeout has advantages as well. First, it enables large log segment sizes. Each log segment in the cloud must be written in a single upload, so the proxy must be able to group updates together; a longer delay allows more writes to be batched together into a single log segment. It also enables overlapping writes to be combined, minimizing the data that must be uploaded. In the extreme case of short-lived temporary files, no data need be uploaded to the cloud. Currently the BlueSky proxy commits data as frequently as once every five seconds. BlueSky does not start writing a new checkpoint until the previous one completes, so under a heavy write load checkpoints may commit less frequently.

The proxy maintains a cache on disk of file system data to satisfy many read requests without going to the cloud. This cache consists of old journal segments and log segments downloaded from cloud storage. Cloud log segments may be downloaded partially, and in this case are stored as sparse files so they take up only approximately as much space on disk as the data downloaded.

The proxy uses a unified LRU policy to manage the cache, both journal segments and log segments from the cloud. The proxy tracks the most recent access to each journal and cloud log segment, and deletes the least-recently accessed segment to maintain the target cache size. The one exception to LRU is that journal segments containing data not yet flushed to the cloud are kept pinned until data is in the cloud. The proxy allows at most half of the disk cache to be pinned in this way.

When reading data from its disk cache, the BlueSky proxy will memory-map the file data. Doing so avoids double-caching of file data, with one copy in the operating system's page cache and another copy in the memory of the BlueSky proxy, and gives the operating system full control over memory cache replacement.

The proxy also keeps in memory an index for locating file system data in the journal and cache. While data flushed to the cloud is indexed for random access, the journal contents are not indexed and so the proxy must maintain this index information in memory at least until data is written to the cloud. (On a system crash and restart, the index of data in the journal is rebuilt via journal replay.)

The BlueSky proxy uses partial reads (HTTP range requests) to decrease the latency and cost to selectively read data objects. When possible, if there are multiple objects to fetch from a single log segment the proxy will download them using a single larger range request.

4.4.2 Connection Management

The BlueSky storage backends keep and reuse HTTP connections when sending and receiving data from the cloud; the CURL library handles the details of this connection pooling. Each upload or download is performed in a separate thread. BlueSky limits uploads to no more than 32 segments concurrently, both to limit contention among TCP sessions and to limit memory usage in the proxy (since each segment is buffered entirely in memory when it is sent).

4.4.3 Merging System State

As discussed in Section 4.3.3, the proxy and the cleaner operate independently of each other. When the cleaner begins execution, it begins working from the most recent checkpoint written by the proxy. The cleaner only ever accesses data relative to this file system snapshot, even if the proxy writes additional updates to the cloud. As a result, the proxy and cleaner each may make updates to the same objects (e.g., inodes) in the file system. Since reconciling the updates requires unencrypted access to the objects, the proxy assumes responsibility for merging file system state.

When the cleaner finishes execution, it writes an updated checkpoint record to its log; this checkpoint record identifies the snapshot on which the cleaning was based. When the proxy sees a new checkpoint record from the cleaner, it begins merging updates made by the cleaner with its own updates.

The general case of merging file system state from multiple writers is a difficult one which we leave for future work. But the case of a single proxy and cleaner is particularly straightforward since only the proxy is making logical changes to the file system—the cleaner merely performs a physical reorganization. In the worst case, if the proxy has difficulty merging changes by the cleaner it can simply discard the cleaner's changes.

The persistent UIDs for objects can optimize the check for whether merging is needed. If both the proxy and cleaner logs use the same UID for an object, the cleaner's version may be used. The UIDs will differ if the proxy has made any changes to the object, in which case the objects must be merged or the proxy's version used. For data blocks, the proxy's version is always used. For inodes, file data is merged block-by-block.

4.4.4 Implementation

Our BlueSky prototype is implemented primarily in C, with small amounts of C++ and Python. The core BlueSky library which implements the file system but not any of the front-ends to it consists of 8500 lines of code (including comments and whitespace). BlueSky uses GLib for data structures and utility functions, libgcrypt for cryptographic primitives, and libs3 and libcurl for interaction with Amazon S3 and Windows Azure.

Our NFS server consists of another 3000 lines of code, not counting code entirely generated by the rpcgen RPC protocol compiler. The CIFS server builds on top of Samba 4, adding approximately 1800 lines of code as a new backend. These interfaces do not fully implement all file system features such as security and permissions handling, but are sufficient to evaluate the performance of the system.

The prototype in-cloud file system cleaner is implemented in just 600 lines of portable Python code and does not depend on the BlueSky core library.

4.5 Evaluation

In this section we evaluate the BlueSky proxy prototype implementation. We explore performance from the proxy to the cloud, the effect of various design choices on both performance and cost, and how BlueSky performance varies as a function of its ability to cache client working sets for reads and absorb bursts of client writes.

We start with a description of our experimental configuration.

4.5.1 Experimental Setup

We ran experiments on Dell PowerEdge R200 servers with 2.13 GHz Intel Xeon X3210 processors (four processor cores), a 7200 RPM 80 GB SATA hard drive, and gigabit network connectivity (between the servers and to the Internet). One machine, with 4 GB of RAM, is used as a load generator. The second machine, with 8 GB of RAM and an additional 1.5 TB 7200 RPM disk drive, acts as a standard NFS/CIFS server or a BlueSky proxy. Both servers run Debian testing; the load generator machine is a 32-bit install (required for SPECsfs) while the proxy machine uses a 64-bit operating system.

We focused our efforts on two providers: Amazon's Simple Storage Service (S3) [3] and Windows Azure storage [30]. For Amazon S3, we looked at both the standard US region (East Coast) as well as S3's West Coast (Northern California) region.

We use the SPECsfs2008 [46] benchmark in many of our performance evaluations. SPECsfs can generate both NFSv3 and CIFS workloads patterned after real-world traces. In these experiments, SPECsfs subjects the server to increasing loads (measured in operations per second) while simultaneously increasing the size of the working set of files accessed. Our use of SPECsfs for research purposes does not follow all rules for fully-compliant benchmark results, but should allow for relative comparisons. System load on the load generator machine remains low, and the load generator is not the bottleneck. Most tests use the NFSv3 protocol, though we also test with CIFS.

In several of the benchmarks, the load generator mounts the BlueSky file system with the standard Linux NFS client. In the read benchmarking in Section 4.5.4, we use a synthetic load generator which directly generates NFS read requests (bypassing the kernel NFS client) to provider better control.

4.5.2 Cloud Provider Bandwidth

To understand the performance bounds on any implementation and to guide our specific design, we measured the performance our proxy is able to achieve writing data to Amazon S3. Figure 4.4 shows that the BlueSky proxy has the potential to fully utilize its gigabit link to S3 if it uses large request sizes and parallel TCP connections. The graphs shows the total rate at which the proxy could upload data to S3 for a variety of request sizes and number of parallel connections. Round-trip time from the proxy to the standard S3 region, shown in the graph, is around 30 ms. We use non-pipelined requests—we wait for confirmation for each object on a connection before sending another one—and so when uploading small objects each connection is mostly idle. Larger objects better utilize the network, but objects of one to a few megabytes are sufficient to capture most gains. A single connection utilizes only a fraction of the total bandwidth, so to fully make use of the network we need multiple parallel TCP connections. These

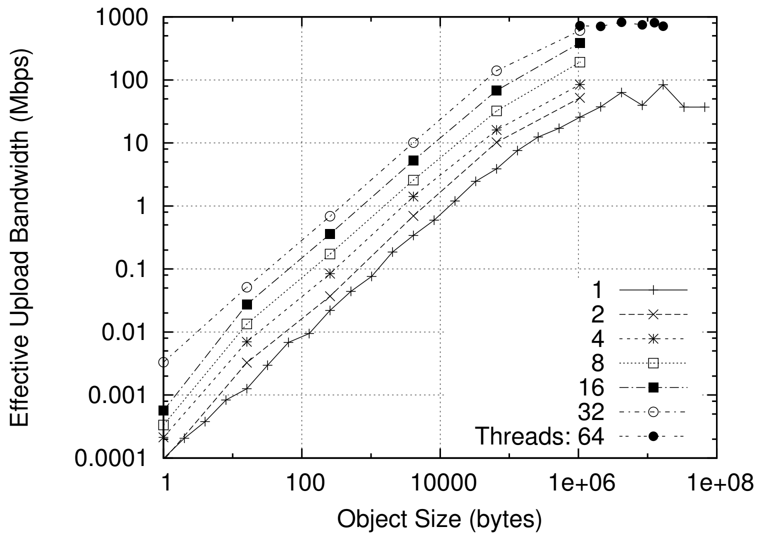


Figure 4.4: Measured aggregate upload performance to Amazon S3, as a function of the size of the objects uploaded (x-axis) and number of parallel connections made (various curves). A gigabit network link is available. Full use of the link requires parallel uploads of large objects.

measurements helped inform the choice of 4 MB log segments (Section 4.3.1) and a pool size of 32 connections (Section 4.4.2).

We also measured the latency to the S3 US-West location is and found it to be lower, around 12 ms. The network round trip time to Azure in our testing was higher, around 85 ms. Network bandwidth was not a bottleneck in either case, with the achievable bandwidth again approaching 1 Gbps. In later benchmarks, we focus primarily upon the Amazon US-West region.

4.5.3 Impact of Cloud Latency

To underscore the impact latency can have on file system performance, we first run a simple, time-honored benchmark of unpacking and compiling a kernel source tree. We measure the time for three steps: (1) extract the sources for version 2.6.37 of the Linux kernel (a write-only workload); the sources consist of roughly 35000 files and 400 MB; (2) checksum the contents of all files in the extracted sources (a read-only workload); (3) build a i386 kernel using the default kernel configuration and the `-j4` flag to permit up to four files to be compiled in parallel (a mixed read/write workload). For a

Table 4.1: Kernel compilation benchmark times for various file server configurations. Steps are (1) unpack sources, (2) checksum sources, (3) build kernel. Times are given in minutes:seconds. Cache flushing and prefetching are only relevant in steps (2) and (3).

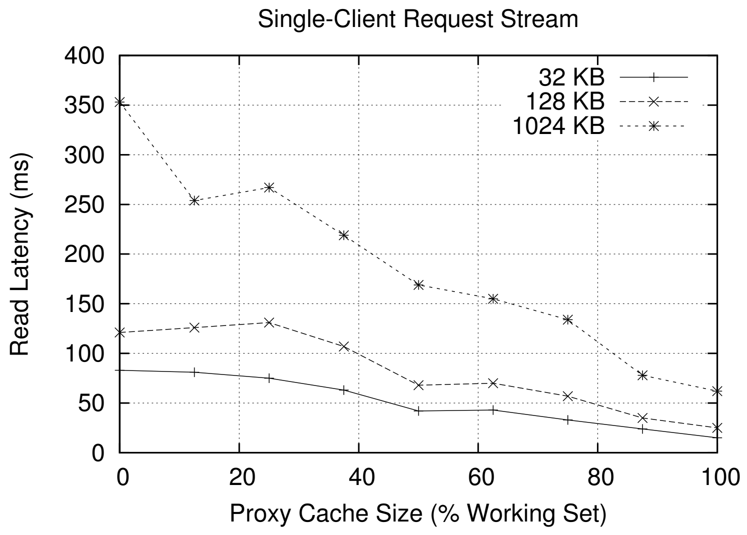
	Unpack	*Check*	*Compile*
Local file system			
warm client cache	0:30	0:02	3:05
cold client cache		0:27	
Local NFS server			
warm server cache	10:50	0:26	4:23
cold server cache		0:49	
NFS server in EC2			
warm server cache	65:39	26:26	74:11
BlueSky/S3-West			
warm proxy cache	5:10	0:33	5:50
cold proxy cache		26:12	7:10
full segment		1:49	6:45
BlueSky/S3-East			
warm proxy	5:08	0:35	5:53
cold proxy cache		57:26	8:35
full segment		3:50	8:07

range of comparisons, we repeat this experiment on a number of system configurations. In all cases with a remote file server, we flushed the client's cache by unmounting and remounting the file system in between steps.

Table 4.1 shows the timing results of the benchmark steps for the various system configurations. Recall that the network links client↔proxy and proxy↔S3, are both 1 Gbps—the only difference is latency (12 ms from the proxy to BlueSky/S3-West and 30 ms to BlueSky/S3-East). Using a network file system, even locally, adds considerably to the execution time of the benchmark compared to a local disk. However, running an NFS server in EC2 compared to running it locally increases execution times by a factor of 6–30$\times$ due to the high latency between the client and server and a workload with operations on many small files.

The substantial impact latency can have on workload performance motivates the need for a proxy architecture. Since clients interact with the BlueSky proxy with low latency, BlueSky with a warm disk cache is able to achieve performance similar to a local NFS server. (In this case, BlueSky performs slightly better than NFS because its log-structured design is better-optimized for some write-heavy workloads.) With a cold cache, it has to read small files from S3, incurring the latency penalty of reading from the cloud. Incidental prefetching from fetching full 4 MB log segments when a client requests data in any part of the segment greatly improves performance (since in this particular benchmark there is a great deal of locality—later on we will see that in workloads with little locality, full fetches hurt performance), but execution times are still multiples of BlueSky with a warm cache. The differences in latencies between S3-West and S3-East for the cold cache and full segment cases again underscores the sensitivity to cloud latency.

In summary, greatly masking the high latency to cloud storage—even with high-bandwidth connectivity to the storage service—requires a local proxy to minimize latency to clients, while fully masking high cloud latency further requires an effective proxy cache.



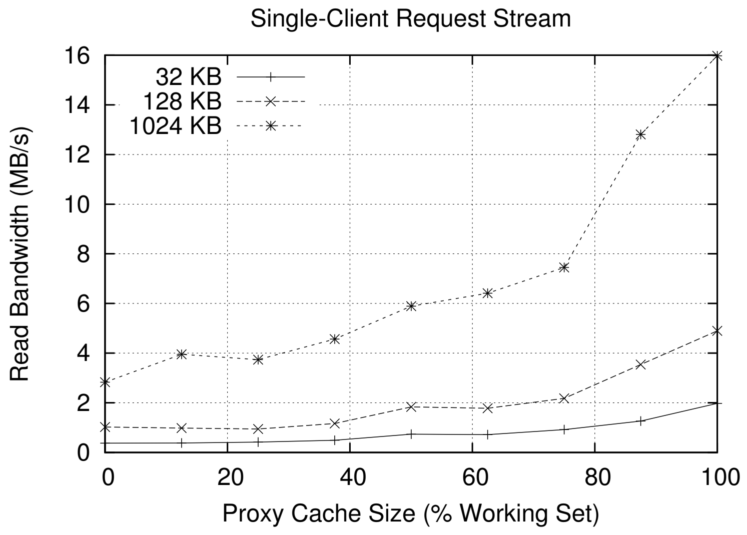


Figure 4.5: Read performance as a function of working set captured by proxy. Both the average latency to respond to the requests and the effective total read bandwidth are shown. Results shown are from a single benchmark run.

4.5.4 Caching the Working Set

The BlueSky proxy can mask the high latency overhead of accessing data on a cloud service by caching data close to clients. The more effective the cache, the closer the performance (as perceived by clients) to that of a local network file server.

For what kinds of file systems can such a proxy be an effective cache? Ideally, the proxy needs to cache the working set across all clients using the file system to maximize the number of requests that the proxy can satisfy locally. Although a number of factors can make generalizing difficult, previous studies have estimated that clients of a shared network file system typically have a combined working set that is roughly 10% of the entire file system in a day, and less at smaller time scales [40, 55]. For BlueSky to provide acceptable performance, it must have the capacity to hold this working set. As a rough back-of-the-envelope using this conservative daily estimate, a proxy with one commodity 3 TB disk of local storage could capture the daily working set for a 30 TB file system, and five such disks raises the file system size to 150 TB. Many enterprise storage needs fall well within this envelope, so a BlueSky proxy can comfortably capture working sets for such scenarios.

In practice, of course, workloads are dynamic. Even if proxy cache capacity is not an issue, clients shift their workloads over time and some fraction of the client workload to the proxy cannot be satisfied by the cache. To evaluate what happens in these cases, we use synthetic read and write workloads—and do so separately because they interact with the cache in different ways.

We start with read workloads. Reads that hit in the cache achieve local performance, while reads that miss in the cache incur the full latency of accessing data in the cloud, stalling the clients accessing the data (unless BlueSky incorporates aggressive prefetching, which we leave as future work). The ratio of read hits and misses in the workload determines overall read performance, and fundamentally depends on how well the cache capacity is able to capture the file system working set across all clients in steady state.

We populate a BlueSky file system on S3 with 32 GB of data using 16 MB files. We then generate a steady stream of fixed-size NFS read requests to random files through the BlueSky proxy. We vary the size of the proxy disk cache to represent different working set scenarios. In the best case, the capacity of the proxy cache is large enough to hold the entire working set: all read requests hit in the cache in steady state, minimizing latency. In the worst case, the cache capacity is zero, no part of the working set fits in the cache, and all requests go to the cloud service. In practice, a real workload falls in between these extremes.

In this experiment, we make uniform random requests to any of the files, so the working set is effectively equivalent to the size of the entire file system. The experiment measures the steady-state performance of client read requests. We also vary the size of client requests: larger requests benefit clients by taking better advantage of network bandwidth.

Figure 4.5 shows that BlueSky with S3 provides good latency even when it is able to cache only 50% of the working set: with a local NFS latency of 21 ms for 32 KB requests, BlueSky is able to keep latency within 2$\times$ that value. Given that cache capacity is not an issue, this situation corresponds to clients dramatically changing the data they are accessing such that 50% of their requests are to new data objects not cached at the proxy. Larger requests have larger absolute latencies, but take better advantage of bandwidth: 1024 KB requests are 32$\times$ larger than the 32 KB requests, but have latencies only 4$\times$ longer.

4.5.5 Absorbing Writes

The BlueSky proxy represents a classic write-back cache scenario in the context of a cache for a wide-area storage backend. In contrast to reads, the BlueSky proxy can absorb bursts of write traffic entirely with local performance since it implements a write-back cache. Two factors determine the proxy's ability to absorb write bursts: the capacity of the cache, which determines the instantaneous size of a burst the proxy can absorb; and the network bandwidth between the proxy and the cloud service, which determines the rate at which the proxy can clear the cache by writing back data. As long as the write workload from clients falls within these constraints, the BlueSky proxy can entirely mask the high latency to the cloud service for writes. However, if clients instantaneously burst more data than can fit in the cache, or if the steady-state write workload is higher than the proxy$\leftrightarrow$cloud service bandwidth, client writes start to experience de-

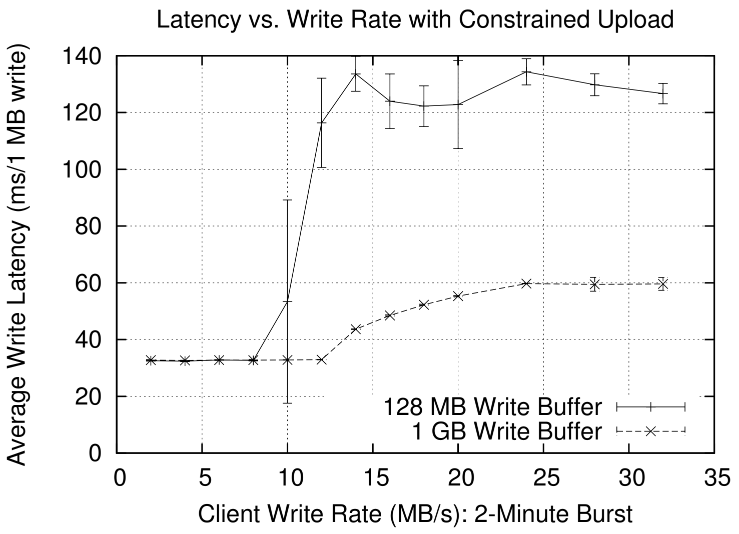


Figure 4.6: Write latencies when the proxy is uploading over a constrained ($\approx$ 100 Mbps) uplink to S3 as a function of the write rate of the client and the size of the write cache to temporarily absorb writes. The y-axis shows the average latency of each of a series of 1 MB write operations for write bursts two minutes in duration, and error bars show the standard deviation over three benchmark runs.

lays that depend on the performance of the cloud service.

We populate a BlueSky file system on S3 with 1 MB files and generate a steady stream of fixed-size 1 MB NFS write requests to random files in the file system. The client bursts writes at different rates for two minutes and then stops. So that we can overload the network between the BlueSky proxy and S3, we rate limit traffic to S3 at 100 Mbps while keeping the client$\leftrightarrow$proxy link unlimited at 1 Gbps. We start with a rate of write requests well below the traffic limit to S3, and then steadily increase the rate until the offered load is well above the limit.

Figure 4.6 shows the average latency of the 1 MB write requests as a function of offered load. At low write rates the latency is determined by the time to commit writes to the proxy's disk. The proxy can upload at up to about 12 MB/s to the cloud (due to the rate limiting), so beyond this point latency increases as the proxy must throttle writes by the client when the write buffer fills. With a 1 GB write-back cache the proxy can temporarily sustain write rates beyond the upload capacity.

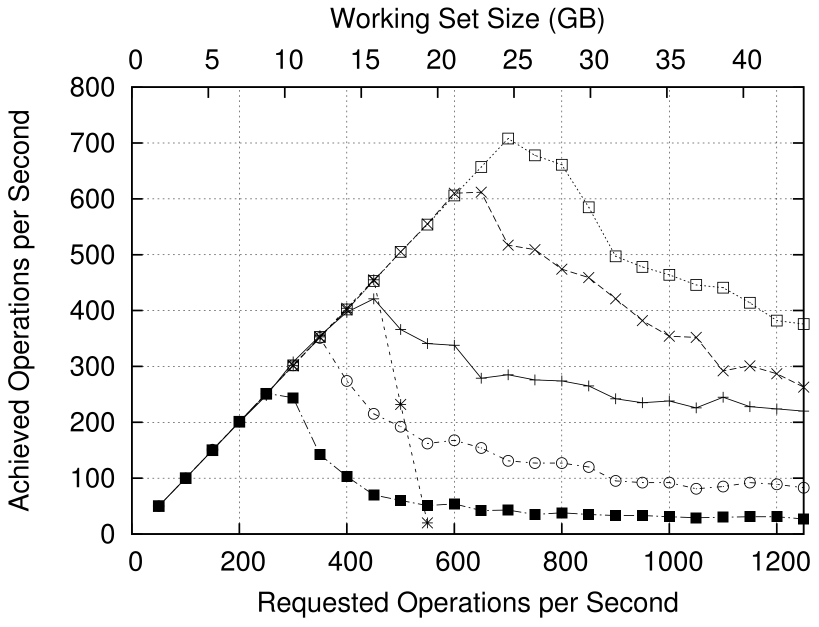
4.5.6 More Elaborate Workloads

Using the SPECsfs2008 benchmark we next examine the performance of BlueSky under more elaborate workload scenarios, both to subject BlueSky to more interesting workload mixes as well as to highlight the impact of different design decisions in BlueSky. We evaluate a number of different system configurations, including a native Linux nfsd in the local network (Local NFS) as well as BlueSky communicating with both Amazon S3's US-West region and Windows Azure's blob store. Unless otherwise noted, BlueSky evaluation results are for communication with Amazon S3. In addition to the base BlueSky setup we test a number of variants: disabling the log-structured design to store each object individually to the cloud (noseg), disabling range requests on reads so that full segments must be downloaded (norange), and using 4 KB file system blocks instead of the default 32 KB (4K).

We run the SPECsfs benchmark in two different scenarios, modeling the case where there is a both a low degree of client parallelism and a high degree of parallelism. In the low-parallelism case, 4 client processes make requests to the server, each with at most 2 outstanding reads or writes. In the high-parallelism case, there are 16 client processes each making up to 8 reads or writes.

Figure 4.7 shows several SPECsfs runs under under the low-parallelism case. In these experiments, the BlueSky proxy was configured to use an 8 GB disk cache. The top graph shows the delivered throughput (in operations per second) against the load offered by the load generator, and the bottom graph shows the corresponding average latency for the operations. At a low requested load, the file servers can easily keep up with the requests and so the achieved operations per second are equal to the requested load. At some point, the server becomes saturated and the achieved performance levels off.

The solid curve in the graphs shows the performance of an NFS server running locally, using one of the disks of the proxy machine for storage. This machine can sustain a rate of up to 420 operations/second, at which point the disk is the performance bottleneck. The BlueSky server achieves a low latency—comparable to the local server case—at low loads since many operations hit in the proxy's cache and avoid wide-area network communication. At higher loads, performance degrades as the working set size



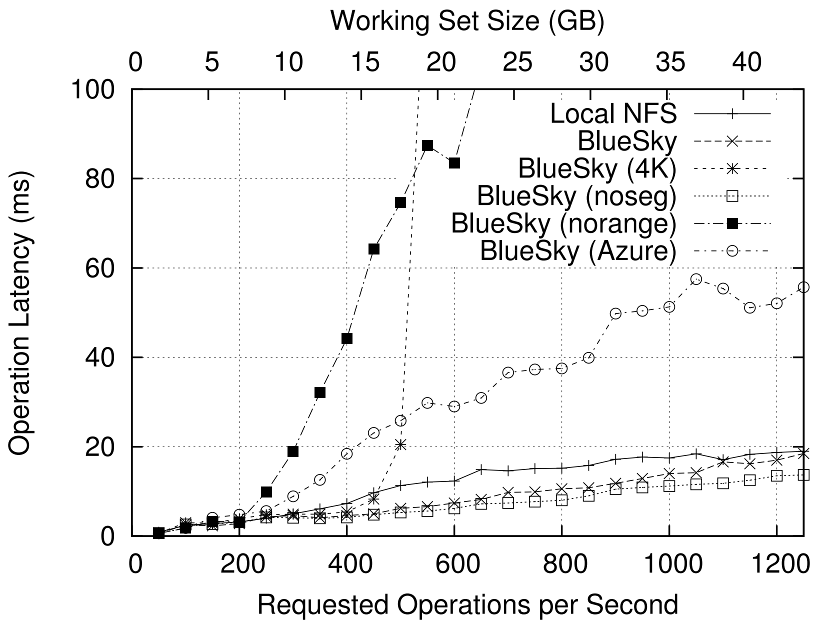
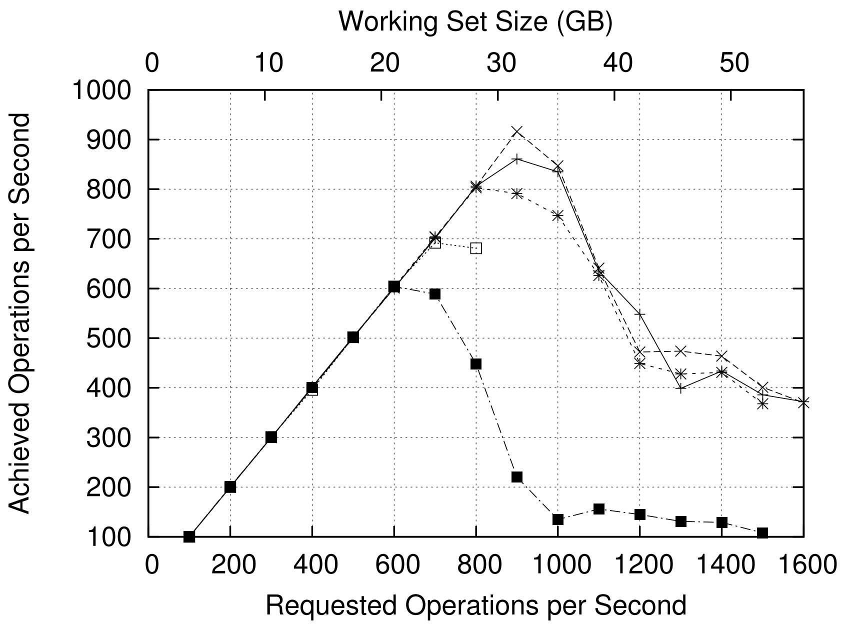


Figure 4.7: Comparison of various file server configurations subjected to the SPECsfs benchmark, with a low degree of parallelism (4 client processes). In these experiments cryptography is enabled for all BlueSky runs, and most tests use the Amazon US-West region. “4K” tests using 4 KB file system blocks instead of the default of 32 KB. “noseg” disables the log-structured design and uses a cloud file per object. “norange” disables range read requests and fetches complete segments from the cloud. “Azure” is just like the standard BlueSky test, but using Azure blob storage instead.



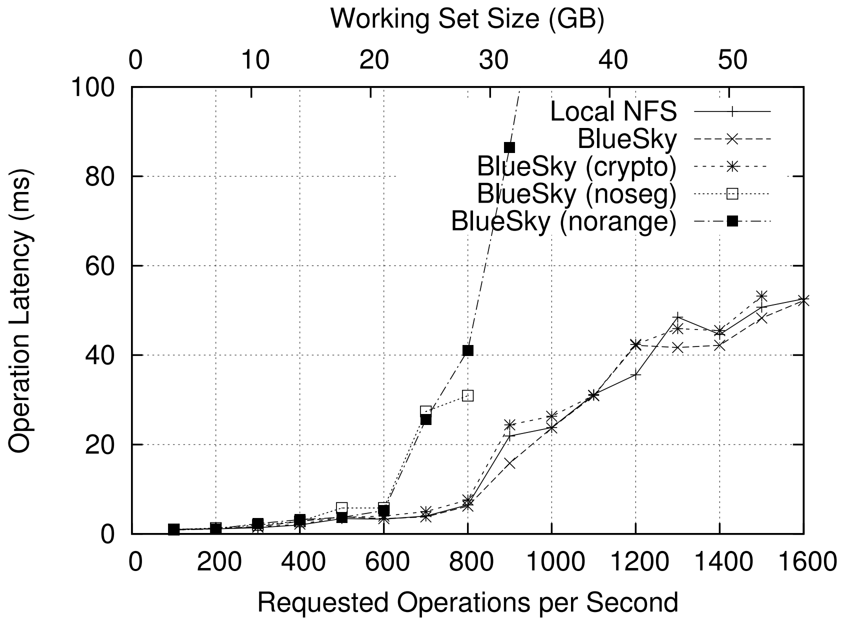


Figure 4.8: Comparison of various file server configurations subjected to the SPECsfs benchmark, with a high degree of parallelism (16 client processes). Each of the tests below "BlueSky" evaluates a single variation from the base BlueSky configuration. Most tests have cryptography disabled, but the "+crypto" test re-enables it. The "noseg" test disables the log-structured design and uses a cloud file per object. "norange" disables range read requests, requiring complete segments to be fetched from the cloud.

increases. In write-heavy workloads, BlueSky performs better than the native Linux NFS server with local disk, since BlueSky commits operations to disk in a single journal and can make better use of disk bandwidth. Fundamentally, though, we consider this approach to using cloud storage successful as long as it provides performance commensurate with standard local network file systems.

BlueSky's aggregation of written data into log segments, and partial retrieval of data with byte-range requests, are important to achieving good performance and low cost with cloud storage providers. As discussed in Section 4.5.2, transferring data as larger objects is important for fully utilizing available bandwidth. As we show below, from a cost perspective as well larger objects are better since small objects require more costly operations to store and retrieve an equal quantity of data.

In this experiment we also used BlueSky with Amazon S3 as the cloud storage provider and with Windows Azure as the cloud provider. Although Azure performed noticeably worse than S3, we attribute the difference primarily to the higher latency (85 ms RTT) to Azure from our proxy.

Figure 4.8 shows similar experiments but this time with a high degree of client parallelism. In these experiments, the proxy is configured with a 32 GB cache. To simulate the case in which cryptographic operations are better-accelerated, cryptography is disabled in most experiments but re-enabled in the "+crypto" experimental run.

Results in these experimental runs are similar to the low-parallelism case. The servers achieve a higher total throughput when there are more concurrent requests from clients. In the high-parallelism case, both BlueSky and the local NFS server provide comparable performance. In testing with cryptography enabled versus disabled, again there is very little difference, so cryptographic operations are not the bottleneck in performance.

4.5.7 Monetary Cost

We are accustomed to optimizing systems to maximize performance or minimize resource usage, but offloading file service to the cloud introduces monetary cost as another dimension for optimization.

Figure 4.7 showed the relative performance of different variants of BlueSky,

Table 4.2: Cost breakdown and comparison of various BlueSky configurations for using cloud storage. Costs are normalized to the cost per one million NFS operations in SPECsfs. Breakdowns include traffic costs for uploading data to S3 (Up), downloading data (Down), operation costs (Op), and their sum (Total).

	Up	*Down*	*Op*	*Total*
Baseline	\$0.56	\$0.22	\$0.09	\$0.87
4 KB blocks	0.47	0.11	0.07	0.65
Full segments	1.00	31.39	0.09	32.48

using data from the low-parallelism SPECsfs benchmark runs. Table 4.2 shows the cost breakdown of each of the variants, normalized per SPECsfs operation (since the benchmark self-scales, different experiments have different numbers of operations). We use the March 2011 prices (in US Dollars) from Amazon S3 as the basis for the cost analysis: \$0.14/GB stored per month, \$0.10/GB transferred in, \$0.15/GB out, and \$0.01 per 10,000 get or 1,000 put operations. S3 also offers cheaper price tiers for higher use, but we use the base prices as a worst case. Overall prices are similar for other providers.

Unlike the performance comparison, Table 4.2 shows that comparing by cost changes the relative ordering of the different system variants. Using 4 KB blocks had very poor performance, but using them has the lowest cost since they effectively transfer only data that clients request. The BlueSky baseline uses 32 KB blocks, requiring more data transfers and higher costs overall. If a client makes a 4 KB request, the proxy will download the full 32 KB block; many times downloading the full block will satisfy future client requests with spatial locality, but not always. Finally, the range request optimization is essential in reducing cost. When the proxy downloads an entire 4 MB segment when a client requests any data in the segment, the cost for downloading data increases by 150$\times$. If providers did not support range requests, BlueSky would have to use smaller segments in its file system layout.

Although 4 KB blocks have the lowest cost, we argue that using 32 KB blocks has the best cost-performance tradeoff. The costs with 32 KB clocks are 34% higher, but the performance of 4 KB blocks is far too low for a system architecture that relies upon wide-area transfers to cloud storage.

4.5.8 Cleaning

As with other file systems that do not overwrite in place, BlueSky must clean the file system to garbage collect overwritten data—although less to recover critical storage space, and more to save on the cost of storing data unnecessarily at the cloud service. Recall that we designed the BlueSky cleaner to operate in one of two locations: remote from the cloud, running on the BlueSky proxy, or local to the cloud, running on a compute instance in the cloud service. Cleaning locally has compelling advantages: it is faster since it is closer to the data and does not compete with the proxy for network bandwidth, and it is cheaper since cloud services like S3 and Azure do not charge for network traffic when accessing data locally.

The overhead of cleaning fundamentally depends on the workload. The amount of data that needs to be read and written back depends on the rate at which existing data is overwritten and the fraction of live data in cleaned segments, and the time it takes to clean depends on both. Rather than hypothesize a range of workloads, we describe the results of a simple experiment as a sense for how the cleaner operates.

We populate a 1 GB BlueSky file system on S3 with 1 MB files. The initial file system is written to the cloud as 267 log segments totalling 1028 MB in size. We then dirty the file system by overwriting 25% files chosen uniformly randomly. This writes another 70 log files totalling 257 MB. However, as only 1 GB of data is actually live in the file system, some of the segments now contain dead data.

We then launch an instance of the cleaner in EC2. The cleaner must read in information about all inodes, which requires reading from essentially all segments. The cleaner identifies and selects 76 segments with a low utilization ($< 60\%$) for cleaning, and writes back 126 MB of compacted data in 34 segments.

After the cleaner has written this new data to the cloud, the proxy notices the log segments written by the cleaner and merges the updates made by the cleaner. The proxy downloads 4 of the segments written by the cleaner (4.3 MB) and uploads another 0.25 MB to complete the merge. Once this is complete, any subsequent runs of the cleaner will be able to delete the now-unused segments (277 MB in 78 segments).

Cleaning does induce additional data transfers at the proxy, but these are small compared with the total workload and add negligibly to the total cost. When the cleaner

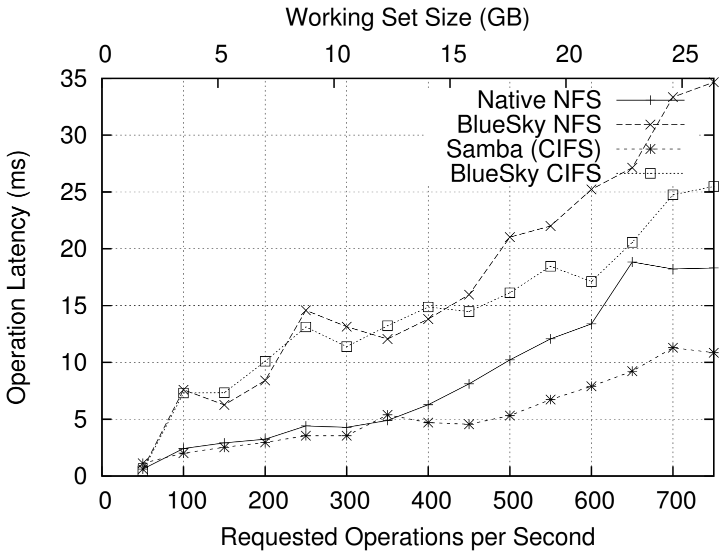


Figure 4.9: Latencies for read operations in SPECsfs as a function of aggregate operations per second (for all operations) and working set size. Both CIFS and NFS protocols are shown, using both the standard implementations as well as the BlueSky variants.

executes in the cloud, the quantity of data transferred has no effect on the cost since data transfers are free. The cloud provider still charges a per-operation cost for access from the cloud, but here at least the log-structured format is helpful: the cleaner can use a single operation to fetch a segment which may contain many inodes. By contrast, a file system integrity checker or other similar tool would have to access many more objects in a non-log-structured design, raising the total cost.

4.5.9 Client Protocols: NFS and CIFS

Finally, we use the SPECsfs benchmark to confirm that the performance of the BlueSky proxy is independent of the client protocol (NFS or CFS) that clients use. The experiments performed above use NFS for convenience, but the results hold for clients using CIFS as well.

Figure 4.9 shows the latency of the read operations in the benchmark as a function of aggregate operations per second (for all operations) and working set size. Because SPECsfs uses different operation mixes for its NFS and CIFS workloads, we focus on the latency of just the read operations for a common point of comparison. We show results for NFS and CIFS on the BlueSky proxy (Section 4.4.4) as well as standard implementations of both protocols (Linux NFS and Samba for CIFS, on which our implementation is based). For the BlueSky proxy and standard implementations, the performance of NFS and CIFS are broadly similar as the benchmark scales, and BlueSky mirrors any differences in the underlying standard implementations. Since SPECsfs uses a working set much larger than the BlueSky proxy cache capacity in this experiment, BlueSky has noticeably higher latencies than the standard implementations due to having to read data from cloud storage rather than local disk.

4.6 Conclusion

The promise of “the cloud” is that computation and storage will one day be seamlessly outsourced on an on-demand basis to massive data centers distributed around the globe, while individual clients will effectively become transient access portals. This model of the future (ironically similar to the old “big iron” mainframe model of the 1960’s and 70’s) may come to pass at some point, but today there are many hundreds of billions of dollars invested in the *last* disruptive computing model: client/server. Thus, in the interstitial years between now and a potential future built around cloud infrastructure, there will be a need to bridge the gap from one regime to the other.

In this chapter, I have explored a solution to one such challenge: network file systems. Using a caching proxy architecture I demonstrate that LAN-oriented workstation file system clients can be transparently served by cloud-based storage services with good performance. However, I show that exploiting the benefits of this arrangement requires that design choices (even low-level choices such as storage layout) are directly and carefully informed by the pricing models exported by cloud providers (this coupling ultimately favoring a log-structured layout with in-cloud cleaning). We describe the design, implementation and performance of a prototype system, BlueSky, that implements this architecture and supports both NFS and CIFS file systems, as well as S3 and Azure storage infrastructures. Finally, based on a variety of benchmark results, I project that a wide range of enterprise workloads are likely to perform well in this regime.

Chapter 4, in part, has been submitted for publication as “BlueSky: A Cloud-Backed Filesystem for the Enterprise” by Michael Vrable, Stefan Savage, and Geoffrey

M. Voelker. The dissertation author was the primary investigator and author of this material.

Chapter 5

Conclusion

In this chapter I discuss several possible directions that new researchers could explore, then summarize my contributions.

5.1 Future Work

There are several directions for research that builds on my work in this dissertation. These include both improving and broadening the set of applications built on the cloud as well as exploring changes to the cloud itself.

In this dissertation I have built applications on top of and within the constraints of existing cloud infrastructure. It is worth investigating the interface to cloud storage more closely—if the infrastructure itself can be changed, how can the applications be improved? I have shown that a simple storage interface is sufficient to capture almost all the benefits in a system like Cumulus, but perhaps a more expressive storage interface could still simplify application-building or allow additional features. If so, how should the cloud interface be changed? Should the storage interface be made more expressive, for example to allow simple instructions for data manipulation to be executed in the cloud? Is the VM model the right one to use for cloud computation, or is some other platform better for offloading tasks to the cloud?

In my work, data stored in the cloud is kept private by protecting encryption keys—the cloud provider cannot read or corrupt data because it lacks the necessary keys. Doing so, however, limits computation that can be performed in the cloud. With Blue-

Sky I show that some tasks—such as log cleaning and garbage collection—can be safely run in the cloud. I have not fully explored all possible trade-offs between outsourcing to the cloud and data security. Depending on what security guarantees are needed, how much computation can be shifted to the cloud? If security guarantees are loosened, how much can efficiency be improved, or what other features can be implemented? Can the way that encryption is done be reworked to provide better guarantees?

Finally, of course, I have investigated two particular applications: backup and shared file systems. There are other possible enterprise storage applications as well, such as databases and database applications. While many of the ideas discussed in this dissertation are likely applicable to other systems, new challenges might arise.

5.2 Contributions

Cloud computing has seen tremendous growth in recent years. Software as a Service offerings continue to thrive. Infrastructure providers continue to roll out additional capacity and capabilities. Many companies have found cloud provider offerings an excellent fit for their needs, and have built many applications on the cloud. Yet despite this, many types of applications have been ignored in the push for the cloud.

Enterprise storage is one area where the cloud has not replaced traditional solutions. The thesis of this dissertation is that the cloud can effectively be used for enterprise storage applications. There are numerous challenges in doing so—security, performance, cost, and dealing with legacy systems—but in this dissertation I have shown that these challenges are surmountable by presenting two systems which do so.

In Chapter 3, I described Cumulus, a tool file system backup to the cloud. In this work with Cumulus, I explore the constraints of current cloud storage interfaces and demonstrate how to build on cloud storage while minimizing cost. In Chapter 4, I build on ideas from Cumulus to implement BlueSky, a network file system backed by cloud storage. BlueSky can be easily deployed in an organization, without changes to legacy clients. In addition to the challenges in Cumulus, BlueSky addresses performance concerns and considers security concerns in more depth.

Cloud computing has already led to many improvements in computing, and in the future seems likely to continue to grow in prevalence. Together, the two systems I have presented demonstrate how the challenges of the cloud can be overcome for enterprise storage applications and provide guidance for future systems—expanding the range of services that can be built on the cloud.

Bibliography

[1] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *ACM Trans. Storage*, 3(3):9, 2007.

[2] Amazon. Amazon web services. http://aws.amazon.com/.

[3] Amazon Web Services. Amazon Simple Storage Service. http://aws.amazon.com/s3/.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53:50–58, April 2010.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[6] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. In *EuroSys 2011*, 2011.

[7] boto: Python interface to Amazon Web Services. http://code.google.com/p/boto/.

[8] Yao Chen and Radu Sion. To Cloud Or Not To Cloud? Musings On Costs and Viability. http://www.cs.sunysb.edu/~sion/research/cloudc2010-draft.pdf.

[9] Cirtas. Cirtas bluejet cloud storage controllers. http://www.cirtas.com/.

[10] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–298. USENIX, 2002.

[11] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. Technical Report MSR-TR-2002-30.

[12] Enomaly. ElasticDrive Distributed Remote Storage System. http://www.elasticdrive.com/.

[13] Ben Escoto. rdiff-backup. http://www.nongnu.org/rdiff-backup/.

[14] Ben Escoto and Kenneth Loafman. Duplicity. http://duplicity.nongnu.org/.

[15] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[16] Brad Fitzpatrick. Brackup. http://code.google.com/p/brackup/, http://brad.livejournal.com/tag/brackup.

[17] FUSE: Filesystem in userspace. http://fuse.sourceforge.net/.

[18] Google. Google app engine. http://code.google.com/appengine/.

[19] I. Heizer, P. Leach, and D. Perry. Common Internet File System Protocol (CIFS/1.0). http://tools.ietf.org/html/draft-heizer-cifs-v1-spec-00.

[20] Val Henson. An analysis of compare-by-hash. *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, 2003.

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

[22] J.H. Howard, M.L. Kazar, S.G. Nichols, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[23] IDC. Global market pulse. http://i.dell.com/sites/content/business/smb/sb360/en/Documents/0910-us-catalyst-2.pdf.

[24] Jungle disk. http://www.jungledisk.com/.

[25] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. SafeStore: A durable and practical storage system. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 10:1–10:14, Berkeley, CA, USA, 2007. USENIX Association.

[26] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, 2004.

[27] librsync. http://librsync.sourcefrog.net/.

[28] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[29] Dutch T. Meyer and William J. Bolosky. A Study of Practical Deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[30] Microsoft. Windows Azure. http://www.microsoft.com/windowsazure/.

[31] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 174–187. ACM, 2001.

[32] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, 2002.

[33] Nasuni. Nasuni: The gateway to cloud storage. http://www.nasuni.com/.

[34] Panzura. Panzura. http://www.panzura.com/.

[35] W. Curtis Preston. *Backup & Recovery*. O'Reilly, 2006.

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

[37] Rackspace. Rackspace Cloud. http://www.rackspacecloud.com/.

[38] Randy Rizun. s3fs: FUSE-based file system backed by Amazon S3. http://code.google.com/p/s3fs/wiki/FuseOverAmazon.

[39] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[40] Chris Ruemmler and John Wilkes. A trace-driven analysis of disk working set sizes. Technical Report HPL–OSR–93–23, HP Labs, April 1993.

[41] Samba. http://www.samba.org/.

[42] Russel Sandberg, David Goldberg, Steve Kleirnan, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Technical Conference*, pages 119–130, 1985.

[43] Julian Satran, Kalman Meth, Costa Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. Internet small computer systems interface (iSCSI), April 2004. RFC 3720, http://tools.ietf.org/html/rfc3720.

[44] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. Network file system (NFS) version 4 protocol, April 2003. RFC 3530, http://tools.ietf.org/html/rfc3530.

[45] Sqlite. http://www.sqlite.org/.

[46] Standard Performance Evaluation Corporation. SPECsfs2008. http://www.spec.org/sfs2008/.

[47] StorSimple. StorSimple. http://www.storsimple.com/.

[48] Ben Summers and Chris Wilson. Box backup. http://www.boxbackup.org/.

[49] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, February 1999.

[50] TwinStrata. TwinStrata. http://www.twinstrata.com/.

[51] Werner Vogels. Eventually consistent, December 2007. http://www.allthingsdistributed.com/2007/12/eventually_consistent.html.

[52] Jun Wang and Yiming Hu. 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, 2002.

[53] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, and John Kubiatowicz. Antiquity: Exploiting a secure log for wide-area distributed storage. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 371–384, New York, NY, USA, 2007. ACM.

[54] David A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/.

[55] Theodore M. Wong, Gregory R. Ganger, and John Wilkes. My cache or yours? Making storage more exclusive. Technical Report CMU-CS-00-157, CMU, November 2000.

[56] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 269–282. USENIX Association, 2008.