

# Cumulus: An Open Source Storage Cloud for Science

John Bresnahan

Mathematics and CS Division

Argonne National Laboratory

bresnahan@mcs.anl.gov

David LaBissoniere

Computation Institute

University of Chicago

labisso@uchicago.edu

Tim Freeman

Computation Institute

University of Chicago

freeman@mcs.anl.gov

Kate Keahey

Mathematics and CS Division

Argonne National Laboratory

Computation Institute

University of Chicago

keahey@mcs.anl.gov

## ABSTRACT

Amazon's S3 protocol has emerged as the de facto interface for storage in the commercial data cloud. However, it is closed source and unavailable to the numerous science data centers all over the country. Just as Amazon's Simple Storage Service (S3) provides reliable data cloud access to commercial users, scientific data centers must provide their users with a similar level of service. Ideally scientific data centers could allow the use of the same clients and protocols that have proven effective to Amazon's users. But how well does the S3 REST interface compare with the data cloud transfer services used in today's computational centers? Does it have the features needed to support the scientific community? If not, can it be extended to include these features without loss of compatibility? Can it scale and distribute resources equally when presented with common scientific the usage patterns?

We address these questions by presenting Cumulus, an open source implementation of the Amazon S3 REST API. It is packaged with the Nimbus IaaS toolkit and provides scalable and reliable access to scientific data. Its performance compares favorably with that of GridFTP and SCP, and we have added features necessary to support the econometrics important to the scientific community.

## Keywords

Storage Cloud, Private Cloud, Infrastructure as a Service (IaaS), Data Transfer, Amazon's Simple Storage Service (S3)

## 1. INTRODUCTION

Storage clouds represent a fusion between data transfer and storage; two actions that up to now were usually considered and optimized separately. The emergence of storage clouds as a useful model raises several questions. To what extent can the existing scientific storage systems be adapted to fit this model? Are existing file/storage management tools suitable for cloud computing? Can we build a storage cloud using a combination of existing tools? How will such a combination need to be adapted to satisfy the expectations of scientific users? What are the performance characteristics of such adaptations, and how can they be improved? Answering these questions provides a path to better leverage the existing knowledge and experience in building storage clouds.

Outsourcing compute and storage infrastructure has many potential benefits. It can provide access to more sophisticated resources than the outsourcing institution can afford to own and operate, it supports more flexible use of such resources, it creates the potential for leveraging economies of scale via consolidation, and it eliminates the overhead of system acquisition and operation. Many outsourcing models have been tried, from multi-institutional sharing to grid computing and commercial hosting

services. Recently, cloud computing [1] emerged as a new outsourcing paradigm that quickly became successful in many commercial venues. Infrastructure as a Service (IaaS) is the most flexible of the mechanisms collectively known as cloud computing; it offers scientists access to computational and storage resources on a on-demand, pay-as-you-go basis.

Storage outsourcing is of particular importance to scientific research, where volumes of data produced by one community can reach the scale of terabytes per day [2, 3]. Sharing and processing of such data require careful planning and trade-off considerations that could be greatly facilitated by storage on-demand services such as those provided by Amazon Simple Storage Service (S3) [4] or Rackspace [5]. For this reason, the study of such services from the perspective of scientific needs attracted early attention [6, 7]. The commercially offered services are closed, however, and thus can be only partially studied. Deep evaluation of the potential of cloud computing as an outsourcing model requires the ability to experiment with the paradigm.

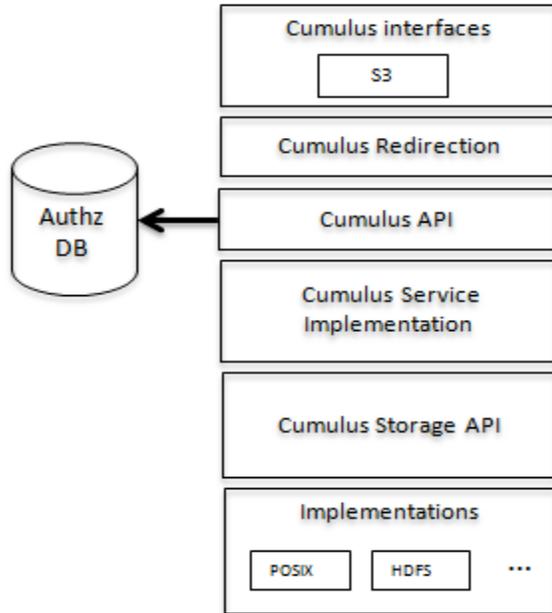
In this paper, we present Cumulus—a storage cloud system that adapts existing storage implementations to provide efficient upload/download interfaces compatible with S3, the de facto industry standard. While this compatibility enables users to easily move between academic and commercial clouds, Cumulus also conforms to scientific community expectations by providing such features as quota support, fair sharing among clients, and an easy-to-use, easy-to-install approach for maintenance. The most important feature of Cumulus is its well-articulated back-end extensibility module. It allows storage providers to configure Cumulus with existing systems such as GPFS [8], PVFS [9], and HDFS [10], in order to provide the desired reliability, availability or performance trade-offs. Cumulus is part of the open source Nimbus toolkit [11, 12], where this “use what you have” approach has also been successfully used to provide a compute cloud service that can be used with batch schedulers [13].

We first describe Cumulus architecture and implementation. We next evaluate Cumulus from the perspective of upload/download efficiency and compare it with representative tools used in the scientific community. We then demonstrate how Cumulus scales over multiple storage servers, and we evaluate the efficiency of such scaling in the context of the GPFS storage system used on many scientific clusters.

## 2. CUMULUS DESIGN

Cumulus provides two functions. First, it allows users to accumulate and manage data: upload data to the cloud, monitor its status, and download it from the storage cloud as needed. Second, since this data can in particular represent VM images, Cumulus also provides an image store for Nimbus compute clouds. Users can use client tools—provided either by Amazon for interaction with S3 or by other third-party tool providers—to access those functions. Since Cumulus is integrated with the Nimbus workspace service, the users can also access Cumulus functions

through the Nimbus cloud-client features to upload and download images.



**Figure 1: Cumulus architecture**

The architecture of Cumulus, shown in Figure 1, is simple and modular, with particular care taken to provide extensibility options at various design levels. The *Cumulus Interfaces* layer exposes the interface to the service and contains modules interpreting and authorizing client commands. The current implementation supports only Amazon’s S3 REST protocol, the de facto commercial standard storage cloud interface. Thus, many client libraries and tools, including `s3cmd` [30], `boto` [31], and `jets3t` [32], can be leveraged by Cumulus users. Simply put, the S3 interface allows clients to write, read, and delete objects (equivalent of files) or organize them into buckets (equivalent of directories). Furthermore, Cumulus interfaces support simultaneous upload to a single object without the risk of data corruption; and, like S3, it supports the notion of eventual consistency [14].

Authentication mechanisms, based on request signature by symmetric key, are provided to ensure that data is kept secure. When a request is made to the service, the authorization database is checked to verify that the user is known and is allowed to perform the requested action on the specified bucket and object. Object permissions are set with an access control list (ACL), which allows a group of users to share data with a rich set of controls according to the S3 protocol [4]. For example, a user can grant read access to one user and write access to another without having to worry about defining user groups. The authorization module handles this functionality without exposing the remainder of the body of code to these details. This allows for different authorization implementations to be created and enabled.

The *Cumulus Redirection* module is used to handle scalability. This portion of Cumulus keeps track of the workload of the service and decides to either accept a new client connection or redirect that client to a replicated server.

The *Cumulus Service Implementation* encompasses a set of modules that implement the functionality needed to convert an API request into an action on the storage system and record all important events along the way. When a user requests that a bucket be created, this component logs the request, checks that the

request is allowed against quota and ACL restrictions, and then invokes the appropriate method on the *Storage API*. Cumulus configuration files are interpreted and respected in this module. Along with this component is a set of command-line tools that allow a user easily to create new users, alter and delete existing users, and manage the users storage quotas.

The *Cumulus Storage API* is a modular system that allows administrators to choose what backend storage system they wish to use. It abstracts the details of sourcing and sinking data from the rest of Cumulus via the various *Implementations*—Cumulus plug-in modules. Similarly, these plug-ins need not be concerned with the details of security, HTTP, or S3. This design allows for fairly easy creation of a storage module and thus integration with most storage system, whether they are simple like local file systems or sophisticated like HDFS. This modular approach to storage system is a key design feature of Cumulus.

Creating a storage module involves the implementation of two abstract classes. The first is a file management class with interface operations for actions such as the creation or deletion of a bucket or the uploading, downloading, or deleting of an object. The second class is responsible for streaming the data to the storage object. This class behaves similarly to an open file. The creation of a storage module is expected to be a moderately easy task involving about two to three hundred lines of python code.

In order to achieve Amazon levels of availability, vast amounts of hardware expenses need to be incurred. Some users may need such levels, but others have more modest requirements. Some storage clouds have the resources simply for a single node; for these, a local disk is all that is needed to back their system, without having to deal with complicated setups. Other storage clouds have a cluster of nodes to back their storage system; such setups typically have shared or parallel file systems like GPFS [8], PVFS [9], or NFS [16] and can use them to back their Cumulus storage cloud. More sophisticated clouds may have highly available data stores like HDFS [10], Sector [17], or Cassandra [18] to back their systems. Our goal is to enable all those different types of storage to be used in the creation of clouds.

### 3. IMPLEMENTATION

Cumulus is implemented in the python programming language as a REST service. Twisted Web [19] is used to handle the marshaling of the HTTP and HTTPS protocols. The Cumulus service implementation handles interpreting the REST API and converting it to the Cumulus API. The Cumulus API is a set of python objects that are responsible for handling specific user requests.

The Cumulus default authorization module is implemented with the database `sqlite` [20]. Because `sqlite` is often limited to a local file system (the details here depend on the use of a shared file system that safely supports locking), this module cannot be used in a replicated configuration. However, a slightly modified version of this module exists that uses `postgres` [21]; this allows Cumulus to be configured as a replicated and scalable service. An additional light-weight module has been created that uses simple text files for managing access to security information.

The Cumulus package released by the Nimbus project includes the POSIX data storage module. Because of the success of FUSE [22], this module is powerful and allows for immediate integration with HDFS, SECTOR, and many other file systems. Additionally we have a `BlobSeer` [33] module that is a research extension.

Many Cumulus servers can be configured to run on separate machines all accessing a shared data store such as GPFS. This replicated pool of Cumulus servers can then be used to handle a

higher client load. The replication module decides whether a redirection is needed and, if so, to which replicated server the client will be redirected. The current release of Cumulus has two redirection modules (both used in our experiments): random and round robin. In both modules a list of all replicated servers is stored in a file (the contents of this file can change without needing to restart the Cumulus servers). Each service is given this file and an integer known as the redirection point. The modules keep track of the total number of current client connections. If that number exceeds the redirection point, a new host is chosen from the list.

The modules differ in how they choose a new host. The random module selects a new host from the list at random. The round-robin module iterates through the list, selecting a new host every time and starting over at the beginning once the list is exhausted. Both algorithms include the current host in their redirection candidates. If the current host is selected, the request is handled without redirection.

## 4. EXPERIMENTS

To assess the viability of Cumulus as a storage cloud for science, we ran a set of experiments on the FutureGrid’s Hotel resource [23]. The compute nodes used in the study were 8-core 4 Xeon 2.40 GHz processors equipped with 24 GB of RAM and 1 Gbps network interfaces (including the c1.uc.futuregrid.org and c2.uc.futuregrid.org nodes we refer to below), connected via a Juniper EX4200 network switch. At the time the fair sharing and performance experiments were run, the nodes were configured to use just 512 MB of memory in order to create resource contention conditions. For the remaining experiments we used the full 24 GB of RAM.

In our experiments we refer to the “client node” (c1.uc.futuregrid.org) host and the “service node” (c2.uc.futuregrid). In our scalability study we had a total of 20 nodes at our disposal for the experiment. In this case all nodes were identically configured as those described above with the exception that they had the full 24 GB of RAM.

### 4.1 Performance

To evaluate the performance of Cumulus, we ran a series of experiments comparing it with the two most commonly used data transfer services in the scientific community: GridFTP [24] and SCP. GridFTP has set the standard for data transfer performance, and SCP is the ubiquitous transfer service that users turn to when they want a simple and immediate solution.

Our experiments compared the throughput obtained by all three tools in uploading and downloading a file. We used the following standard command-line interface tools to perform the transfer with their respective service: for GridFTP we used `globus-url-copy`; for SCP, the `scp` command; for Cumulus, the `s3cmd`. The speed of the file system was measured by using `Bonnie++` [25]. The throughput was calculated by dividing the file size by the time each command took to complete.

Typically, data transfer services perform best on files of a significant size, because small files have a low payload-to-overhead ratio. The optimal file size for any given transfer service varies slightly depending on details of the protocol and implementation. Hence, we chose to show the results starting with a small file and gradually increasing the file size, allowing us to see the relative trends. Specifically, in our experiments we measured the time all three services took to both upload and download a range of file sizes from 2 MB to 2 GB, doubling the file size for each new measurement. We took 10 measurements

for each file size; our results display the mean of 10 trials (with standard deviation less than 1 for most measurements).

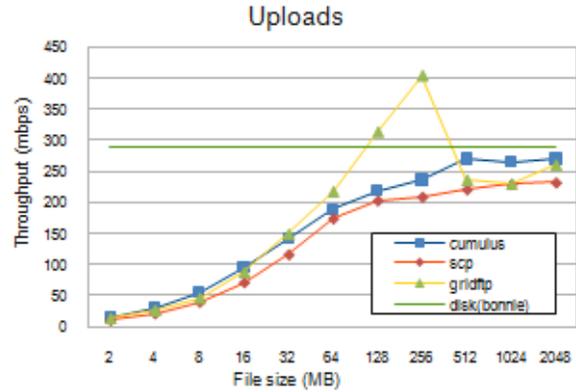


Figure 2: Comparison of upload throughput

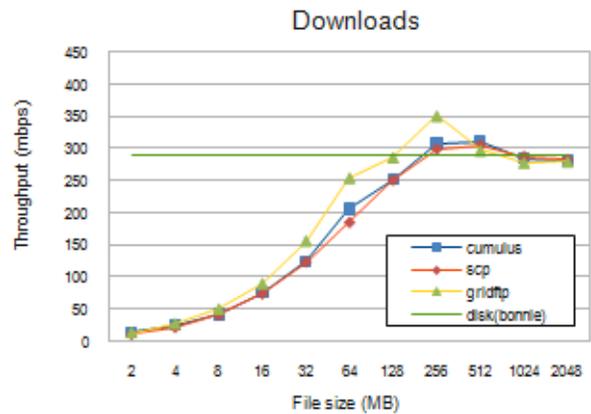
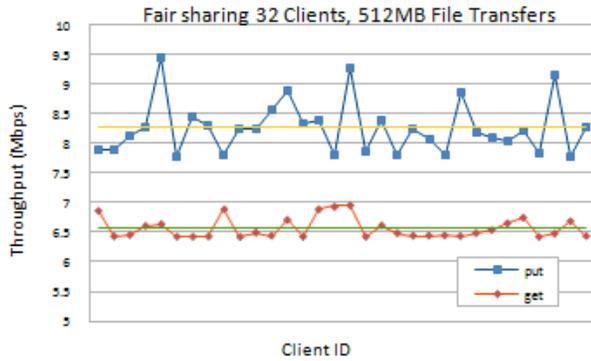


Figure 3: Comparison of download throughput

The results in Figures 2 and 3 show that the performance of Cumulus is on a par with the most common and best-performing data transfer services in use today. GridFTP and Cumulus display similar performance characteristics; in fact, Cumulus slightly outperforms GridFTP for file sizes larger than 512 MB. The spike in GridFTP performance just before the 512 MB file size (visible particularly in Figure 2) is an artifact of GridFTP memory buffering techniques. We plan to investigate whether applying similar buffering strategies would be useful in Cumulus without breaking its storage semantics.

### 4.2 Fair Sharing

The fair sharing experiment evaluates to what extent we meet our goal to equally ration resources to clients. In the study we operated a single Cumulus service instance and had 32 clients simultaneously upload or download a single 512 MB file; we measured the upload and download time for each client. As before, the Cumulus service was located on the service node. All of the clients were run on the client node. Recall that each machine has only 512 MB of RAM; this limitation introduces resource contention conditions.



**Figure 4: Comparison of upload and download times for multiple clients**

The results of the study are shown in Figure 4. Each data point on the graph is the achieved throughput of one of the clients; the solid lines show the average throughput of all clients. We see that the achieved bandwidth is stable for downloads: the largest deviation from the mean is 0.40. The upload case is more variable, with a maximum deviation from the mean of 1.18 mbps. The standard deviation is for downloads is 0.18 and for uploads is 0.45. The higher variance of the upload case is due to kernel caching. In the upload case, all 32 clients opened the same file on the same host. Inevitably, some clients read portions of the file before others. The kernel has the ability to cache portions of the file that early readers read in memory, allowing later readers to gain performance benefits when reading. Fair sharing as visible to the client will thus rely not only on the mechanisms implemented in Cumulus but also on how the service is used.

To evaluate how much overhead is introduced when many simultaneous clients are consuming resources, we looked at the total bandwidth consumed by all clients in the fairness study, and we compared it with the bandwidth consumed by a single client in the performance study. In the download case, the throughput is 310 mbps for a single client and 210 mbps is the sum total for 32 clients (i.e., overhead of roughly 30%). In the upload case, however, the throughput for a single client is 269 mbps and for 32 clients is 264 mbps. In this case, the efficiency is likely helped by the caching effect explained above.

### 4.3 Scalability

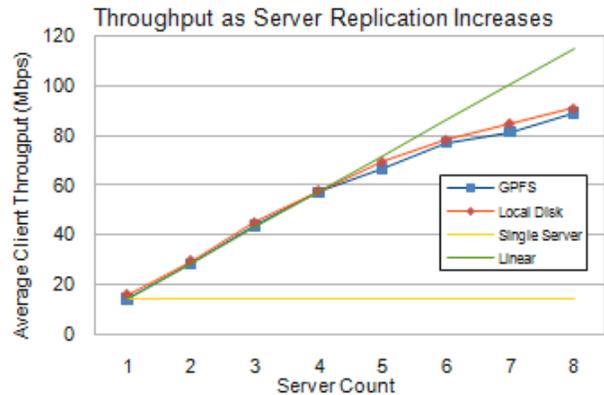
To meet the needs of large-scale storage clouds, Cumulus must be able to scale horizontally across many nodes. By leveraging the S3 protocol's redirect feature, Cumulus can be configured to run as a set of replicated hosts. In our experiments, replicated instances of Cumulus were configured in two ways. The first used the parallel file system GPFS, which allows many hosts to mount the same file system and thus have the exact same view of the data store. Hence, Cumulus servers can be placed on all nodes that have the GPFS file system mounted and be fully replicated Cumulus servers. Because GPFS is a file system commonly found on scientific data clusters, configuring Cumulus in this way is representative of real-world scenarios.

We note, however, that GPFS is a shared resource. How fairly it shares data streams with competing clients depends on its implementation details. Further, GPFS uses a network that is also a shared resource. Because of all these variables, it is important that we study Cumulus in a less dependent configuration. In our second configuration, therefore, we have 8 Cumulus servers associated with local disk partitions. To present the same view of

the data store to all clients, we copied a single data set to all of the nodes. Thus, every server became a read-only mirror of the entire dataset. While this is a less realistic configuration, it provides a baseline for our study.

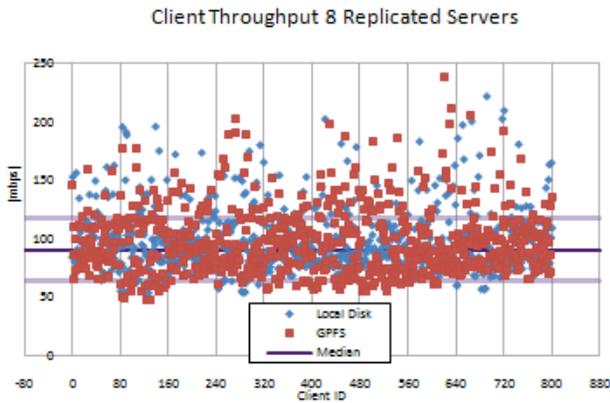
#### 4.3.1 Increasing Replication Factor

In the first scalability evaluation, we ran an experiment in which 80 clients, run on 8 machines (10 clients on each), downloaded a 512 MB file at the same time. In this experiment all previously described machines, both clients and servers, were configured with 24 GB of RAM. We steadily increased the number of replicated servers from 1 to 8, and we used the same method described previously to measure the throughput that each client achieved. Redirects were handled with the round-robin algorithm. The mean of 10 trials was recorded; Figure 5 shows the results.



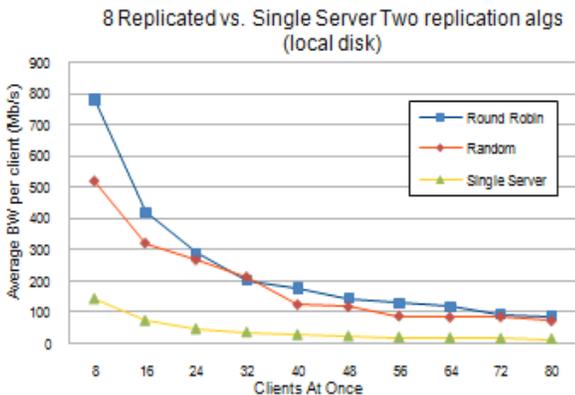
**Figure 5: Server replication**

The solid horizontal line is an extension of the single-server case and is intended to be a baseline showing what the results would be if a single server was used. The solid green line indicates what the performance would be if the single-server case scaled linearly as new resources joined the pool. The other two lines show the measured performance of the experiment when storing to a local disk and to GPFS. In both cases throughput steadily increases as more servers are added. There is a linear increase for the first four replicated Cumulus services, but after that point each additional server has less effect. Ideally, we would see a linear increase in performance, so that eight servers were 8 times faster than one server. In practice, this is not the case. One reason is that the HTTP redirect incurs some overhead. A new connection must be formed to a new Cumulus service at a time of heavy network contention. The set of clients that were not redirected at this point are streaming data, and the set of clients that were redirected are competing with these data flows when trying to form connections. Another reason is that as each server is added, the overall bandwidth that the network must be able to switch is increased by 1 Gbps. We conjecture that as this overall availability is consumed, stress in lower layers of the network stack is introduced.

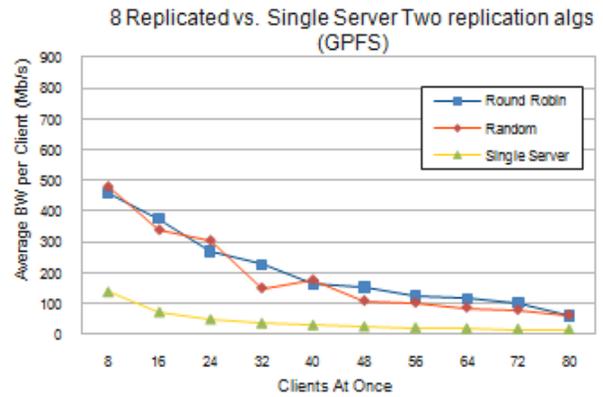


**Figure 6: Individual client measurements for server replication**

Figure 6 shows a scatter graph of the achieved throughput of each client when eight replicated servers are used at once. The median of all performance is indicated on the graph with the solid line at 96 Mbps. We see that 80% of the clients are +/- 27 mbps of the median; a few clients achieve much higher throughput, but none suffer very poor transfer rates. All the individual experiments show a consistent pattern: the high-achieving cases occur when a client contacts a server and is not redirected, so it can start transferring immediately. Not only do these clients avoid the penalty of redirection, but for a brief period (before the other clients are able to connect to their redirected hosts, authenticate, and begin transferring) they enjoy a noncongested network.



**Figure 7: Increasing client load for local disk**



**Figure 8: Increasing client load for GPFS**

### 4.3.2 Increasing Client Load

The graphs in Figures 7 and 8 show the scalability study from another angle. Here we have a static number of eight replicated servers, and we vary the number of clients simultaneously requesting a transfer from 8 to 80. We again use eight client machines, and in each data point we add another client to every machine, increasing the total by 8 for each data point. The achieved throughput is plotted against the number of clients. We show the two redirection algorithms and the two file systems. As more and more clients are added, the available network bandwidth is divided, giving each client a smaller slice. Thus the throughput trails off under the heavier load. However, the replicated service consistently outperforms the single service by a significant factor. As mentioned above, we would ideally see a consistent factor of 8. However, as the bar graphs in Figures 9 and 10 show, in the best case we see a factor of roughly 6.5 and in the worst case a factor of 4. While the redirection overhead does explain some portion of this discrepancy, it cannot account for all the discrepancy.

### 4.3.3 Effects of the File System and Redirection Algorithm

When studying Figures 7 and 8, we see that with one exception all the data points have fairly similar lines. The exception is the first data point for the round-robin line on the local disk graph (Figure 7), which is greater than 250 mbps higher. The reason is that the case in question provides ideal conditions. There are eight servers and eight clients. Since we are using a local disk, no network contention is introduced by the storage system. The round-robin algorithm has each client redirected to a new host, thus providing equal distribution, with one client associated with one server. The client has the full resources of a single Cumulus service and can move at NIC speeds. Further, all the requests start at once, and all request the same size file, so they all end at roughly the same time.

The Random algorithm does not show such favorable results because it cannot guarantee that each server gets a new host. Nor do we see these ideal results when using the GPFS file system because GPFS itself is a shared resource and it uses the same network that Cumulus is using for transfers. Because it is a shared resource, all transfers must use it at the same time and therefore

cannot perform as fast as they can in the local disk case.

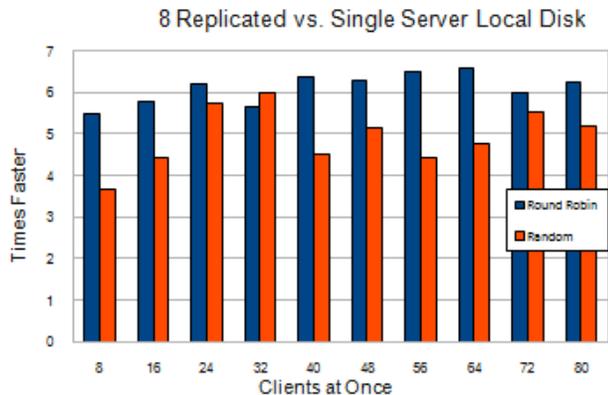


Figure 9: Round-robin and Random comparison (local disk)

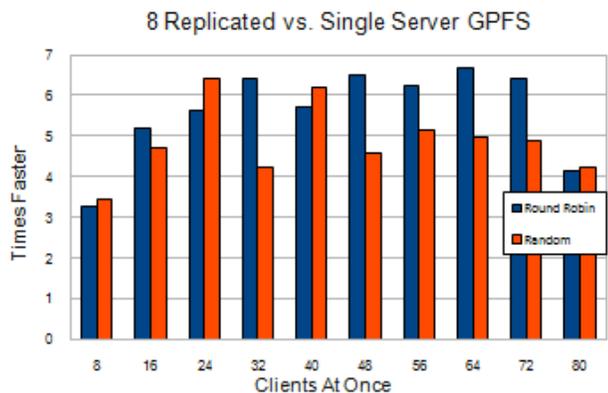


Figure 10: Round-robin and Random comparison (GPFS)

The Random algorithm with GPFS provides a much more realistic indicator for what could be expected in a live Cumulus storage cloud deployment. When comparing GPFS against local disk, we note that only marginal differences exist between the two for the Random algorithm. Local disk does significantly outperform GPFS with the round-robin algorithm, but this is an unlikely event to occur in a real deployment and should be considered a best-case scenario.

## 5. RELATED WORK

The Amazon Web Services S3 is a hosted storage cloud service. Unlike Cumulus, S3 is closed source, and the software is not available for private clouds of any size. Thus, one cannot extend S3 and experiment with its capabilities, as we can do with Cumulus.

The Eucalyptus project [26] has an S3 compatible service similar to Cumulus called Walrus. However, the project is open core, which means that many of its features are not available for extension and experimentation. For example, although the enterprise edition does have support for quotas, the open source version does not. Walrus also does not provide support for replicated service like that of Cumulus shown in the scalability study. In contrast, Cumulus is fully open source and specifically designed with a “use what you have” approach in mind.

Open Stack [27] has a storage cloud component called Swift. At the time of this study no implementation of Swift was available. Swift focuses on providing an integrated storage cloud solution for very large-scale generic clouds, and its architecture has complexity suitable to the task. Cumulus targets mid-size clouds and focuses on the “use what you have” approach

leveraging the existing systems developed specifically in the context of scientific data. Additionally, since Swift does not provide an S3-compliant interface, users cannot leverage the debugged and documented tools available for S3 and therefore cannot fall back to an outsourced storage cloud in times of heavy load or redundancy needs that surpass their hardware budget.

OpenNebula [28] provides an image store facility for uploading and downloading VM images. However, it provides operations only for reading and writing VM images and their associated metadata. It does not provide a general-purpose storage cloud as Cumulus does. Further, it does not have an S3-compatible interface.

GridFTP [29] is a high-speed data transfer service. Its focus is more on network transfer speeds on underutilized high-speed networks and less on the storage of data. It allows for parallel TCP streams to be used, significantly increasing performance but potentially at the expense of fair sharing. We hope to leverage lessons learned and techniques used in GridFTP to increase the performance of Cumulus while still maintaining important semantics associated with a storage cloud.

## 6. CONCLUSIONS AND FUTURE WORK

We have presented the design of Cumulus, an extensible, open source storage cloud implementation designed to adapt existing storage mechanisms for cloud usage. Cumulus implements the AWS S3 interface to provide compatibility with the de facto industry standard. In order to support scientific projects, however, Cumulus also provides support for additional features such as quotas—a widely used mechanism to ensure controlled resource usage in the scientific community. The customizable back-end allows providers to leverage existing scientific domain storage systems and thereby choose what trade-offs—in terms of complexity, reliability, and availability—their storage cloud should have, ranging from a trivially easy installation to highly available, reliable service based on HDFS.

We present an evaluation of various aspects of the Cumulus implementation. We found that the transfer rate of both uploads and downloads is on a par with technologies commonly used in science. In our fairness study we found that Cumulus distributes resources to simultaneous clients in a way consistent with our service level agreement. Further, our scalability study shows that Cumulus can take advantage of multiple storage servers to optimize uploads and downloads and can scale to withstand high levels of client loads.

Our experiments highlight the potential for further study. Fundamentally, the concept of a storage cloud is a fusion between data transfer and storage management: two issues that up to now were usually considered (and optimized) separately. We plan to further examining to what extent techniques used in systems such as GridFTP can be usefully applied to storage clouds and how the current storage systems can be adapted to receive them.

## ACKNOWLEDGEMENTS

This material is based on work supported in part by the National Science Foundation under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed." Partners in the FutureGrid project include U. Chicago, U. Florida, San Diego Supercomputer Center - UC San Diego, U. Southern California, U. Texas at Austin, U. Tennessee at Knoxville, U. of Virginia, Purdue I., and T-U. Dresden. This work also was supported in part by the Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

## REFERENCES

1. Armbrust, M., et al. Above the Clouds: A Berkeley View of Cloud Computing. Tech. report EUB/EECS-2009-28, University of California at Berkeley. 2009.
2. Iamnitchi, A., S. Doraimani, and G. Garzoglio. Filecules in High-Energy Physics: Characteristics and Impact on Resource Management. In High Performance Distributed Computing (HPDC). 2006.
3. Ball, N.M., and D. Schade, Astrominformatics in Canada. White Paper, 2010.
4. Amazon Simple Storage Service (Amazon S3): <http://aws.amazon.com/s3/>.
5. Rackspace: <http://www.rackspace.com/>.
6. Garfinkel, S., An Evaluation of Amazon's Grid Computing Services: EC2, S3, and SQS. 2007.
7. Palankar, M., A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon S3 for Science Grids: A Viable Solution? In International Workshop on Data-Aware Distributed Computing. Boston, MA. 2008.
8. Schmuck, F., and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In 1st USENIX Conference on File and Storage Technologies (FAST '02). Berkeley, CA. 2002.
9. Carns, P. H., I. W. Ligon, R. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In 4th Annual Linux Showcase and Conference. Atlanta, GA. 2000.
10. Shvachko, K., H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In IEEE 26th Symposium on Mass Storage Systems and Technologies. 2010.
11. The Nimbus Toolkit: [www.nimbusproject.org](http://www.nimbusproject.org).
12. Keahey, K., I. Foster, T. Freeman, and X. Zhang. Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid. *Scientific Programming* 13 (4):265-275. 2005.
13. Freeman, T., and K. Keahey, Flying Low: Simple Leases with Workspace Pilot. In EuroPar 2008, 2008.
14. Vogels, W., Eventually Consistent. *ACM Queue*, 2008. 6.
15. Open Cloud Computing Interface (OCCI): <http://occi-wg.org/>.
16. Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In Proceedings of the Summer USENIX Conference. June 1985.
17. Gu, Y., and R. Grossman. Sector and Sphere: the Design and Implementation of a High Performance Data Cloud. In CCA. 2008.
18. Cassandra: <http://cassandra.apache.org/>.
19. Twisted Matrix Labs: <http://twistedmatrix.com/trac/wiki>.
20. SQLite Home page: <http://sqlite.org/>.
21. PostgreSQL: <http://www.postgresql.org/>.
22. FUSE: Filesystem in Userspace: <http://fuse.sourceforge.net/>.
23. FutureGrid: [www.futuregrid.org](http://www.futuregrid.org).
24. Allcock, W., GridFTP: Protocol Extensions to FTP for the Grid. In Global Grid Forum. 2003.
25. Bonnie Disk I/O Benchmark: <http://www.textuality.com/bonnie/>.
26. Nurmi, D., R. Wolski, C. Grzegorzczak, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In CCGrid. 2008.
27. OpenStack: The open source, open standards cloud: <http://openstack.org/>.
28. The OpenNebula Project: <http://www.opennebula.org/>.
29. Allcock, W., J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The Globus Striped GridFTP Framework and Server. In SC '05. 2005.
30. s3cmd : command line S3 client: <http://s3tools.org/s3cmd>.
31. boto: Python interface to Amazon Web Services: <http://code.google.com/p/boto/>.
32. jets3t: An open source Java toolkit for Amazon S3 and CloudFront: <http://jets3t.s3.amazonaws.com/>.
33. Bogdan Nicolae. High Throughput Data-Compression for Cloud Storage. Pages 1-12 in Proceedings of the Third International Conference on Data Management in Grid and Peer-to-Peer Systems (Globe'10). Abdelkader Hameurlain, Franck Morvan, and A. Min Tjoa (eds.). Springer-Verlag, Berlin. 2010.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.