Adapting a Scientific Data Server to Life in the Cloud-Computing Environment

James Gallagher
and David Fulker
and Nathan Potter
OPeNDAP
165 Dean Knauss Dr.
Narragansett, RI 02882-1124
jgalagher@opendap.org
dfulker@opendap.org
npotter@opendap.org

Deirdre Byrne
and Jefferson Ogata
and John Relph
NOAA NODC
1315 East-West Highway
Silver Spring, MD 20910-3282
deirdre.byrne@noaa.gov
Jefferson.Ogata@noaa.gov
John.Relph@noaa.gov

***Abstract*—This paper provides a work-in-progress report on a collaboration between OPeNDAP, Inc and NOAA's National Oceanographic Data Center to prototype serving data stored in cloud computing environments. We present the design of a prototype that supports both a web-browser and an OPeNDAP server accessing those data. An important constraint on the design is to limit vendor lock-in so a final implementation can be realized on a cloud platform internal to NOAA should that be required.**

I. Introduction

In mid-October 2012 OPeNDAP, Inc and NOAA's National Oceanographic Data Center (NODC) began looking at using cloud computing services to provide access to data that NODC currently hosts on its own computing resources and serves using OPeNDAP's software. This is a work-in-progress report on that project. In this paper we will describe the OPeNDAP data server that implements the Data Access Protocol (DAP) which provides I/O middleware for scientific data, its use within NOAA and some background on Cloud Computing systems. Following that, we will describe the project and present our results to date.

A. OPeNDAP background

OPeNDAP is a non-profit corporation that was formed at the conclusion of the Distributed Oceanographic Data System (DODS) project started at The University of Rhode Island Graduate School of Oceanography. One key result of the DODS project was to develop an I/O middleware layer that could be used to move scientific information stored in a variety of formats (often, but not always 'self describing') from a remote server to a client application. Designed in 1994, this I/O layer was dubbed the Data Access Protocol (DAP) and has been in use for nearly twenty years in the oceanographic and atmospheric communities; in 2007 NASA adopted DAP as a 'community standard' [1]. DAP defines three request/response pairs that, taken together, provide access to the data and metadata that make up a particular dataset. Using DAP, clients access datasets using URLs, where each dataset has a single URL and different requests for responses are made by appending specific extensions to the URL as well as adding additional 'constraint' information to the URL in the query string. For both data and metadata access, clients can subset any DAP-accessible dataset on the fly, retrieving just the variables, or parts of variables, they want. Each request made using DAP is independent of any previous request (DAP is a stateless protocol) although many clients request one or both of the metadata responses before making any data requests as this helps formulate sensible constraints for the request. A complete description of the protocol is beyond the scope of this paper, but one can be found in DAP 2.0 specification document [1]. Several other organizations have implemented DAP-enabled data servers or clients using several different programming languages.

Part of the interest in adapting OPeNDAP's data server to the cloud stems from the DAP's ability to subset data. Because the protocol supports subset operations on-the-fly, users often transfer smaller data volumes than when downloading complete files. Since commercial cloud services charge, in part, based on data volume transferred, a data server/protocol that helps minimize transfer volumes is attractive.

II. Overview

The Silver Spring (MD) office of NODC uses both web/ftp and OPeNDAPs Hyrax to provide data-access services for some 80 TB of oceanographic data, and access statistics show average monthly transfers totaling 17.8 TB (with significant month-to-month variation).[1] As the data volumes kept by NOAA increase, and interest in long-term (climate) observations and reanalyses rises, they continue to look for cost effective ways to provide as much information on-line (or near-line) as possible. Cloud computing offers one technology that might help balance accessibility, flexibility and cost. In this project to-date, we have examined several issues surrounding

[1]This is only a partial picture of NODC's activities and holdings, as NODC includes NCDDC, the NOAA Central Library, and a data distribution site in Hawaii.

running the OPeNDAP data server in a cloud computing system. While focused on a specific data server and cloud provider, our results can be applied to other systems that handle data that are roughly equivalent in size and granularity.

Our work so far has used Amazon, Inc.'s cloud system, but we strived to not base our investigation on its proprietary features. We want these results to be applicable to as many of the emerging cloud computing software stacks as possible, including open source alternatives to commercial software that NOAA itself might deploy internally.

By 'cloud computing system' we mean computing resources where virtual computers can be used for very short periods of time, down to the granularity of a single processing task. Similarly, we assume these systems include storage capable of scaling almost infinitely to meet demand. In both cases these resources can be acquired and released incrementally on an as-needed basis using either manual (i.e., operator-configured) or software control.

Because cloud systems provide 'virtual hardware,' some architectural components that are conceptually familiar take on novel characteristics in cloud systems. For example, while Amazon's cloud infrastructure provides virtual computers (called Elastic Compute Cloud (EC2) instances) that are effectively drop-in replacements for conventional hardware running Linux, it also provides two kinds of storage that offer features most users don't have on their desktops. Amazon's S3 and Glacier storage systems provide storage for very large data volumes but with significant restrictions when compared to spinning disks. The S3 system provides a 'bucket based' storage system where each bucket can store any data object up to 5GB and where there is (in most scenarios) no limit to the number of buckets. Unlike a traditional Unix file system, S3 does not support a hierarchical organization scheme; in many respects it more closely resembles a highly-scalable key-value pair storage system [2].

The Glacier storage system is similar to S3 in that it provides a 'flat' storage model, but it behaves more like a near-line storage device (e.g., like a robotic tape drive). With Glacier, data-access latencies can be four hours. [3, p.40] Amazon's cloud environment offers two other storage technologies: virtual disks, bundled with EC2 compute instances, which lose their contents when the instance is retired; and Elastic Block Store (EBS), which simulates a disc drive, retains data between uses, and may be bound to different EC2 instances at different times. Lack of persistence makes virtual disks inappropriate for the problem at hand. Though EBS offers lower latency, its size limitations and costs make it less interesting than the S3 and Glacier alternatives in the NODC context.

In the first part of this project, described by the rest of this paper, we have extended the OPeNDAP data server to work with S3. In continuing work we are expanding the project so that the server works with data stored in Glacier as well. We found that mapping traditional hardware concepts onto the cloud environment worked well for computing processes but less so with the S3 or Glacier storage devices. Several systems that emulate traditional Unix file systems while using S3 as the underlying data store were evaluated in this study, but they proved to be cumbersome to use with S3. We implemented an initial prototype based on the S3FS [4] user-space filesystem but found that it had slower data access times than the native interface provided by Amazon. We developed an approach that leveraged the inherent characteristics of S3 and provided, using XML documents to map data files to S3 items, more flexibility than S3FS.

III. Implementation

A. Constraints

There are several APIs available from Amazon and third parties that can be used to read and write information with S3. We want to limit our implementation to tools that will introduce the minimum limitations going forward. A tool that interoperates with other tools would be favored over one that does not.

Even though this project used Amazon's cloud system, we do not want to limit our results to any one vendor. For example, Amazon's system is distinctly different from the OpenStack cloud system used by RackSpace; we would not want our results to apply only to Amazon and be meaningless in the context of RackSpace/OpenStack. Its worth noting that OpenStack is open source software, and it is reasonable to imagine NOAA using it to run its own cloud infrastructure and thus compatibility (at least conceptually) with OpenStack, in particular, is very desirable.

B. Use cases

There are two essential use cases we examined: Simple web access and Access using OPeNDAP. Note that we did not examine use cases regarding moving data into the cloud. This was intentional because we wanted solutions that were not predicated on how data got in the data stores.

C. Design

We found that Amazon and OpenStack both have a number of APIs—both bundled and third party—that simulate traditional Unix device behavior for the various CPU and storage devices. The CPUs (i.e., Virtual Machines) are, in fact, Linux VMs and are very easy to work with.[2] The storage devices are a different matter, however. The EBS behaves most like typical Unix disk drives, but its cost and—especially—its limited capacity 1 TB per virtual disk) make it quite impractical for large-scale data provision as at NODC. The S3 and Glacier storage devices are noteworthy because they provide nearly limitless capacity at low cost, but their interfaces and behaviors are very different from typical Unix file systems.

Furthermore S3 and Glacier are so distinct that we decided to study them separately, postponing the latter as discussed in the section titled "Future Directions: Glacier". S3 provides a flat data store where the "total volume of data and number of objects you can store are unlimited. Individual Amazon S3

[2]MicroSoft's Azure cloud provides Windows VMs. In other work we have run Hyrax on these using Cygnus, so they can be treated as if they provide a Unix environment as well.

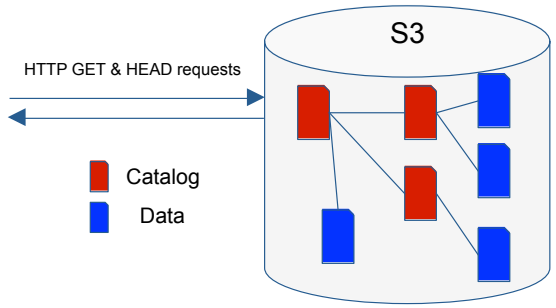

Fig. 1. XML catalogs and data files within S3. Web access is completely transparent given that the catalogs can be rendered as HTML by a browser.

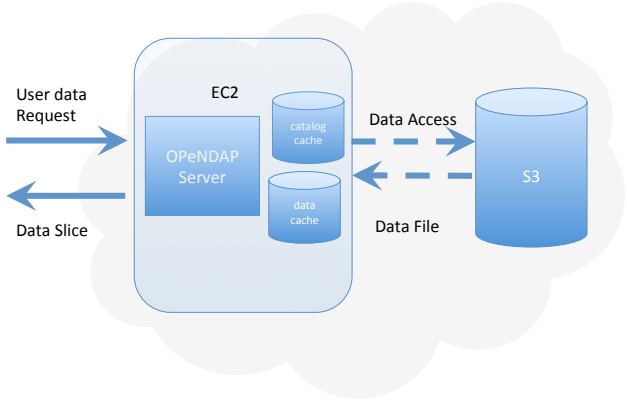

Fig. 2. Data requests made to the OPeNDAP server can easily be handled using a simple modification of the server to process the catalog files and redirect accesses made by the server.

objects can range in size from 1 byte to 5 terabytes. The largest object that can be uploaded in a single PUT is 5 gigabytes" [5, "How much data can I store?"] S3 provides a REST/HTTP API and a SOAP API. S3 also provides access control at the bucket level. There are a number of User-space file systems that provide a traditional Unix file system interface for S3. We decided against these because they violate our implementation constraint against vendor lock-in. All of the S3FS tools we found use proprietary encoding—storing files in buckets along with other metadata—in a way roughly similar to how a file system formats a hard disk. This means that for any given S3FS to read data, it has to be used to write those data too. This level of lock in was something we wanted to avoid very much. Of course, for projects where data are ephemeral, this poses no problem and it is easy to see how these interfaces would be very appealing (an affordable and virtually limitless Unix file system) that can be used by existing software with no modification and an interface well understood by programmers.

Our solution is a) to map the file structures of NODC's source datasets—which currently reside on typical file systems (sometimes in very large files)—onto unique sets of S3 Objects, b) to fully describe these mappings via XML documents (dubbed Catalogs) stored in separate Objects and c) to perform all actions on data and Catalogs via the S3 REST HTTP interface. By allowing Catalogs to reference other Catalogs (recursively), as well as to reference data-containing Objects, this solution yields a simple, hierarchical storage and retrieval system (Figure 1). In fact, without relying on proprietary components, this is similar enough to a Unix file system for OPeNDAP's Hyrax to employ it with minimal modification (Figure 2).

IV. Analysis

The simple solution of using XML files as catalogs provides a number of interesting benefits. The data can be both written and read using only the S3 HTTP API. By accessing the XML catalog files and rendering them as HTML, a browser can be used to download data from S3 with no client software beyond a web browser to perform simple HTML link traversal. The Web browser experience using this solution is no different (for the user) than any website that provides file downloads using HTTP. Because HTTP is part of the S3 interface, there is no need to run a separate web server process to support browser-based access using these XML catalog files. The only software that is specific to this solution is the code that writes the XML files. Note that our XML catalogs (currently stored in S3) could just as easily be kept on any of Amazon's alternative (lower latency) storage systems, potentially reducing the costs of indirection. Because our Catalogs are small, the cost of doing this would be insignificant.

OPeNDAP's data server has modules that access different kinds of data stores, including one that can read from a web service using HTTP GET. The data server module reads files named in the XML catalogs from S3 using HTTP, caches those files and then passes control to one of its other modules to extract data from the files. Of course, the file format must be understood, but for all of the data used in this trial, that was the case (netCDF, HDF4). The only modification we made to the Hyrax software was so that it could process the catalog files.

The combination of these two access methods is particularly strong because many of NODC's uses are accustomed to a simple web browser interface for downloading files. NODC statistics indicate that data transfer-volumes are significantly less[3] when acquisition occurs via OPeNDAP protocols(2) because the DAP subset-creation options allow users to retrieve only the data they need (in contrast to web/ftp, where retrieving whole files is the only option). This difference may be especially important in cloud environments such as Amazon's, where outbound traffic volume is a major cost driver.

A. Comparing HTTP GET Against S3FS (as Access Methods for S3)

Because our initial prototype (see the "Overview" section) was constructed using S3FS, the more recent implementation over HTTP GET created an opportunity to study performance

[3]In April 2013, 21 million files were accessed, of which about 5% (1.1 million) were DAP-style retrievals.

TABLE I
ACCESS TIMES USING S3FS AND HTTP GET FOR 100 SATELLITE IMAGERY DATA FILES.

	S3FS	HTTP/GET
Total transfer time	405s	27s

differences between these two mechanisms for S3 data access. Our comparison was built with command-line tools (`cp`, `curl` and `time`) and the copying of 100 "files" (from S3 to EBS), where each file was drawn from a typical NODC data collection. Under multiple and varied tests, the latencies of S3FS (with caching disabled) were an order of magnitude higher than those measured with HTTP GET (see Table I), despite the expense of our Catalog-based indirection.

Unsurprisingly, similar tests of S3FS with caching enabled showed enhanced performance, but only when the same file was accessed multiple times. In those cases the S3FS latencies were actually better (by over 50%) than those measured with raw HTTP GET, but the differences are indistinguishable when HTTP GET is augmented with front-end caching software.

Clearly, S3FS simplifies development by simulating a traditional, cached Unix file system, but the performance penalty is very high unless it is known that most of the requested data will remain cached. For NODC it is clear that HTTP GET (with a caching front end) is the optimal choice, offering the lowest (mean) latencies for any given cache size without introducing unreasonable implementation complexity.

As expected, turing on S3FS's caching system increases its performance when the same file is accessed more than once. Our tests show that while the initial access is slightly more costly with caching turned on, subsequent access to the same data are much faster (458 seconds for the initial copy of 100 data files compared to 13 seconds for copying files cached by S3FS). Note that for these tests, S3FS was configured to cache to an Amazon EBS instance—a realistic scenario. However, if HTTP GET is paired with a software front-end that caches the retrieved files, its cached accesses costs are indistinguishable from those of S3FS.

From these results we conclude that S3FS presents a classical tradeoff in ease of use versus performance. It enables a system to work with data in S3 as if it is reading from a traditional Unix file system, but optimal performance is highly dependent on having most of the data to be accessed stored in its cache. Using HTTP offers much faster initial access times (and equal performance for cached files) and thus can achieve the same mean performance with a smaller cache size or better performance with the same cache size.

B. Security

Security issues are always present with systems that use remote servers to provide access to data. One issue our solution presents is that data previously accessed using only a local file system are now accessed using URLs and thus those data might be anywhere on the web. While our implementation limited the data to our S3 instance, in a more general case one of the catalog files could contain URLs to places other than S3. The data server used a white-list to ensure that it only read data from known trusted sources (i.e., the S3 data store), and that mitigates the risk considerably, but the design potentially exposes what has previously been an unexposed interface of the server. That is, in previous uses of the server, the data files and the interfaces that access them are assumed to be private and not writeable by third-parties. However, using this design, if the XML catalog files are compromised and the server's white-list not used, it would be possible to expose the server to a malicious data file. The data server module we used is specifically designed for use within a LAN and to only access known hosts, but these protections are only as secure as the server's configuration files. This illustrates a vulnerability, not just for OPeNDAP's server, but for any system that accesses S3 via HTTP GET: such systems must be protected against spoofs that could trigger reading from arbitrary web locations.

C. Future directions: Glacier

Adapting OPeNDAP's data server so that it can efficiently serve data from S3 proved to be fairly straightforward, largely because the data server supports reading data using HTTP GET; the only additions to the software involved processing the XML catalog files. Glacier, however, will likely be another matter. Because Amazon's Glacier data store resembles a near-line tape system, the server's behavior will have to change at a fairly fundamental level. The DAP was designed for use with *on-line* data, so the servers and clients expects all responses to be sent or received immediately. Serving data from Glacier, or any near-line device, will violate this assumption. To address this issue, as part of the NOAA-funded project that has support the work reported here, we have extended the DAP to support asynchronous accesses and are working on a prototype interface to Glacier using those extensions.

V. CONCLUSION

So far we have implemented a prototype interface that can efficiently serve scientific data stored in Amazon's S3 data store. By using the simple HTTP GET interface Amazon provides, we are able to devise a single catalog system that provides a seamless web interface to the data and a high-performance interface for our data server. We did this while avoiding vendor lock-in associated with user-space filesystems, which store data in S3 in what is effectively a proprietary form. The cost of the design was some effort to write the XML catalog files that organize the data in S3 and a modification of our data server to use those catalogs

ACKNOWLEDGMENT

We would like to thank NOAA's National Environmental Satellite Data and Information Service Program Office (NESDISPO) for funding "OPeNDAP-Unidata Linked Servers (OPULS): Aligning, Linking & Integrating Open-Source Software for Web-Based Scientific Data Exchange" (CFDA Number: 11.440) that enabled this work.

REFERENCES

[1] J. Gallagher, N. Potter, T. Sgouros, S. Hankin, and G. Flierl, "The data access protocol—DAP 2.0," NASA ESE-RFC-004.1.1, 2007, October 2007, available at: https://earthdata.nasa.gov/our-community/esdswg/standards-process-spg/rfc/esds-rfc-004-dap-20.

[2] Amazon, Inc., "Working with amazon s3 objects," http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingObjects.html, March 2006, accessed on 30 August 2013.

[3] ——, "Amazon glacier developer guide api version 2012-06-01," http://awsdocs.s3.amazonaws.com/glacier/latest/glacier-dg.pdf, June 2012, accessed on 5 September 2013.

[4] "Fuse-based file system backed by amazon s3," https://code.google.com/p/s3fs/wiki/FuseOverAmazon, accessed on 30 August 2013.

[5] Amazon, Inc., "Amazon simple storage service faqs," http://aws.amazon.com/s3/faqs/, 2013, accessed on 30 August 2013.