Chapter 26

USING HYBRID GRID/CLOUD COMPUTING TECHNOLOGIES FOR ENVIRONMENTAL DATA ELASTIC STORAGE, PROCESSING, AND PROVISIONING

Raffaele Montella
Department of Applied Science
University of Napoli Parthenope, Napoli, Italy

Ian Foster
Argonne National Laboratory, Argonne (IL), USA, and
The University of Chicago, Chicago (IL), USA

1. Introduction

High-resolution climate and weather forecast models, and regional and global sensor networks, are producing ever-larger quantities of multidimensional environmental data. To be useful, this data must be stored, managed, and made available to a global community of researchers, policymakers, and others.

The usual approach to addressing these problems is to operate dedicated data storage and distribution facilities. For example, the Earth System Grid (ESG) [4] comprises data systems at several US laboratories, each with large quantities of storage and a high-end server configured to support requests from many remote users. Distributed services such as replica and metadata catalogs integrate these different components into a single distributed system.

As both environmental data volumes and demand for that data grows, servers within systems such as ESG can easily become overloaded. Larger datasets also lead to consumers wanting to execute analysis pipelines "in place" rather than downloading data for local analysis—further increasing load on data servers. Thus, operators are faced with important decisions as to how best to configure systems to meet rapidly growing, often highly time-varying, loads.

The emergence of commercial "cloud" or *infrastructure on demand* providers [13]—operators of large storage and computing farms supporting quasi-instantaneous on-demand access and leveraging economics of scale to reduce cost—provides a potential alternative to the servers operated by systems such as ESG. Hosting environmental data on cloud storage (e.g., Amazon S3) and running

analysis pipelines on cloud computers (e.g., Amazon EC2) has the potential to reduce costs and/or improve the quality of delivered services, especially when responding to access peaks [1].

In this chapter, we present the results of a study that aims to determine whether it is indeed feasible and cost-effective to apply cloud services to the hosting and delivery of environmental data. We approach this question from the twin perspectives of architecture and cost. We first examine and present the design, development, and evaluation of a cloud-based software infrastructure that leverages some grid computing services and dedicated to the storage, processing, and delivery of multidimensional environmental data. In our design we used the Amazon EC2/S3 cloud computing APIs in order to provide an elastic hosting and processing facilities for data, and the Globus Toolkit v4 (GT4) to federate data elements in a wider grid application context. The scalability is ensured by a hybrid virtual/real aggregation of computing resources.

The rest of this chapter is as follows. In section 2, we introduce the application context, describing important characteristics of environmental data and what scientists need to accelerate research. In section 3, we describe the current status of elastic allocated storage and the tools we developed in order to realize our goals, stressing on the new capabilities produced benefitting the computer scientist community. A GT4 [10] web service providing environmental multidimensional datasets using a grid/cloud hybrid approach is described in section 4, while section 5 presents experiments that characterize the performance and cost of our approach; these results can help identify the best deployment scenario in a real world operational applications. Finally, in section 6 we present some conclusions and outline future work.

2. Distributing multidimensional environmental data

ESG is built upon the Globus Toolkit and other related technologies. ESG continues to expand its hosted data and data processing services, leveraging an environment that addresses authentication, authorization for data access, large-scale data transport and management, services and abstractions for high-performance remote data access, mechanisms for scalable data replication, cataloging with rich semantic and syntactic information, data discovery, distributed monitoring, and Web-based portals for using the system. Current work aims to expand the scope of ESG to address the need for federation of many data sources, as will be required for the next phase of the Intergovernmental Panel on Climate Change (IPCC) assessment process.

In the world of environmental computational science, data catalogues are implemented, managed, and stored using community developed file standards such as Network Common Data File (NetCDF), mainly used for storage and (parallel) high performance retrieval, and the Gridded Binary format (GriB), usually used for data transfer. The most widely used data transfer protocol is OpenDAP (the

Open source Project for a Network Data Access Protocol), formerly DODS (Distributed Oceanographic Data System). OpenDAP supports a set of standard features for requesting and transporting data across the web [11]. The current OpenDAP Data Access Protocol (DAP) uses HTTP for requests and responses. The Grid Analysis and Display System (GrADS) [6] is a free and open source interactive desktop tool used for easy access, manipulation, and visualization of earth science data stored in various formats such as binary, GRIB, NetCDF, and HDF-SDS. The GrADS-DODS Server (GDS) combines GrADS and OPeNDAP to create an open-source solution for serving distributed scientific data [21].

In previous work, Montella et al. developed the GrADS Data Distribution Service (GDDS) [14] service, a GT4-based web service for publishing and serving environmental data. The use of GT4 mechanisms enables the integration of advanced authentication and authorization protocols and the convenient publication of service metadata, which is published automatically to a GT4 index service. This latter feature enables resource brokering involving both data and other grid resources such as CPUs, storage, and instruments—useful, for example, when seeking data sources that also support data processing.

Hyrax is a data server that combines the efforts at UCAR/HAO to build a high performance DAP-compliant data server based on software developed by OpenDAP [12]. A servlet frontend formulates a query to a second backend server that reads data from the data stores and returns DAP-compliant responses to the frontend. The frontend may then either pass responses back to the requestor, perhaps with modifications, or it may use them to build more complex responses.

Montella et al. turned to a Hyrax-based software architecture when they developed the Five Dimensional Data Distribution Service (FDDDS), leveraging the Hyrax OpenDAP server to achieve a better integration within a web/grid service ecosystem. To implement FDDDS, they extended the client OpenDAP class APIs to separate them from the frontend and to provide the needed interfaces in order to provide services to the grid. FDDDS inherits most GDDS features included the automatic index service advertisement of available metadata. This GT4-based environmental data delivery service operates on local data storage. Thus, cloud deployment is possible in a fully virtualized context with no kind of cloud-specific optimization [9].

3. Environmental data storage on elastic resources

Our goal in this work is to explore whether it is feasible to leverage Amazon cloud services to host environmental data. In other words, we want to determine the difficulty, performance, and economic cost of operating a service like FDDDS with data (and perhaps processing as well) hosted not on local resources but on cloud resources provided by Amazon. This service, like FDDDS, should allow remote users to request both entire datasets and subsets of datasets, and ultimately also to

perform analysis on datasets. Whether the service is deployed in a native grid environment or in a grid on cloud fashion should be transparent to the consumer. Ideally, the service will inherit the security and standard connection interface from grid computing and achieve scalability and availability thanks to the elastic power of the cloud.

In conducting this study, we focus in particular on performance issues. The use of dynamically allocated cloud resources has the potential for poor performance, due to the virtualized environment, internal details of cloud storage behavior, and extra cloud/intra-cloud network communication. We anticipate that it will be desirable to move as much processing work (subsetting and data analysis) as possible into the cloud, so as to minimize the need for cloud-to-outside world data transfer. This approach can also help to reduce costs, given that Amazon charges for the movement of data between its cloud storage and the outside world.

3.1 Amazon cloud services

We summarize important characteristics of the Amazon EC2, S3, and EBS services that we use in this work.

The *Elastic Compute Cloud* (EC2) service allows clients to request the creation of one or more virtual machine (VM) instances, each configured to run a VM image supplied by the client. The user is charged only for the time (rounded up to the nearest full hour) a EC2 instance is up and running. Different instance types are supported with different configurations (number of virtual cores, amount of memory, etc.) and costing different amounts per hour. An EC2 user can configure multiple VM images and run multiple instances of each to instantiate complex distributed scenarios that incorporate different functional blocks such as web servers, application servers, and database servers. EC2 provides tools, web user interface and APIs in many languages that make it straightforward to create and manage images and instances. A global image library offers a starting point from which to begin image setup and configuration.

The *Simple Storage Service* (S3) provides a simple web service interface that can be used to store and retrieve data objects (up to five Gbytes in size) at any time and from anywhere on the web. Only write, read, and delete operations are allowed. The number of objects that can be created is effectively unlimited. The object name space is flat (there is no hierarchical file system): each data object is stored in a bucket and is retrieved via a unique key assigned by the developer. The S3 service replicates each object to enhance availability and reliability. The physical location of objects is invisible to the user, except that the user can choose the geographical *zone* in which to create the object (currently, US West, US East, and Europe). Unless objects are explicitly transferred, they never leave the region in which they are created.

S3 users can control who can access data or alternatively can make objects available to all. Data is accessed via REST and SOAP interfaces designed to work with any Internet development toolkit. S3 users are charged for storage and for transfers between S3 and the outside world. The default download protocol is HTTP; a BitTorrent protocol interface is provided to lower costs for large-scale distribution. Large quantities of data (e.g., a large environmental dataset) can be moved into S3 by using an import/export service based on physical delivery of portable storage units, which is more rapid and less expensive than Internet upload.

The *Elastic Block Store* (EBS) provides block-level storage volumes that can be attached to a EC2 instance as a device, but that persist independently from the life of any particular instance. This service is useful for applications that require a database, file system, or access to raw block-level storage, as in the case of NetCDF file storage. EBS volumes can range in size from one to 1,000 GB. Multiple volumes can be mounted to the same instance, allowing for data striping. Storage volumes behave like raw, unformatted block devices, with user-supplied device names and a block device interface. Instances and volumes must be located in the same zone. Volumes are automatically replicated. EBS uses S3 to store volume snapshots in order to protect data for long-term durability and to instantiate as many volumes as needed by the user. EBS performance can vary because the needed network access and the S3 snapshot interfacing and are deeply related to the specific application, so benchmark are needed for each case. The user is charged only for the data stored in the volume plus how much S3 consumed for snapshots (storage space and I/O operations).

From the developer's perspective, the use of EBS is completely transparent because volumes are seen as block devices attached to EC2 instances. In contrast, S3 features require the use of web service APIs. In order to interface S3 with NetCDF we choose the Java implementation freely available in source code.

3.2 Multidimensional environmental data standard file format

As we deal here with data in NetCDF format, we describe that data format briefly. NetCDF is a data format and abstraction, implemented by a software library, for storing and retrieving environmental multidimensional data. Developed by UCAR in the early 90s for meteorological data management, NetCDF has become a widely used data format for a wide range of environmental computing science applications. The NetCDF implementation provides a self-describing and machine-independent format for representing multidimensional scientific data: the abstraction, the access library, and the data format support the creation, access, and sharing of scientific information.

The NetCDF data abstraction models a scientific data set as a collection of named multidimensional variables (scalars and arrays of bytes, characters, integers, and floating-point numbers) along with their coordinate systems and some of their named auxiliary properties. Each variable has a type, a shape specified by a list of

named dimensions, and a set of other properties described by attribute pairs. The NetCDF interface allows data to be accessed by providing a variable name and a specification for what part of the data associated with that variable is to be read or written, rather than by sequential access and individual reads and writes. A dimension is a named integer used to specify the shape of one or more variables, and usually represents a real physical dimension, such as time, latitude, longitude, or atmospheric level. A variable is an array of values of the same type, and is characterized by a name, a data type, and a shape described by a list of dimensions. Attributes may be a single value or a vector of values. One dimension may be unbounded. A variable with a shape that includes an unbounded dimension can grow to any length along that dimension. The unbounded dimension is like a record number in conventional files; it allows us to append data to variables.

NetCDF software interface implementations are available in C, Fortran, Java, and MatLab, among others. We work here with the NetCDF-Java library, a 100% Java framework for reading NetCDF and other file formats into the Common Data Model (CDM), a generalization of the NetCDF, OpenDAP and HDF5 data models, and for writing to the NetCDF file format. The NetCDF-Java library also implements NcML, which allows the developer to add metadata to CDM datasets, as well as to create virtual datasets through aggregation. This library implementation permits access to NetCDF files via network protocols such as HTTP and, via a plug in architecture, enables the development of different data reader.

3.3 Enhancing the S3 APIs

S3 has two important limitations that complicate its use for large environmental multidimensional data sets. The first is the five gigabyte maximum object size, which is too small for environmental applications. For example, a real time weather forecasting application developed at DSA/uniParthenope in 2003 and still running today produces each day an 11 gigabyte NetCDF dataset just from runs performed with the Weather Research and Forecast (WRF) model [2]. The second is the requirement that an object must be read or written in its entirety. The most basic operation performed on multidimensional environmental datasets is subsetting: extraction of scalars, arrays, and matrices along one or more dimensions. This operation requires random access to stored objects. Thus, simply storing each NetCDF file as an S3 object is inefficient and (for larger files) also infeasible.

On the positive side, S3 uses a highly reliable replica and location service completely transparent to the user: each object is named by a unique resource identifier and accessed by an URL that can be made public. Multiple concurrent accesses to different objects belonging to the same bucket are possible without an evident loss in performance.

The design of our S3-enhanced Java API seeks to overcome S3's limitations while preserving S3's high performance and availability (Figure 1, 2). As noted earlier, each S3 data object is identified via a URL. While S3 does not support a hierar-

chical file system, an S3 URL string can include most printable chars including the slash, which is commonly used to structure folder and file names. Internal Amazon services that use S3 for storage commonly partition large datasets (e.g., VM images) across multiple S3objects, with the set of objects that make up the dataset listed in a *manifest file*.

We adopt a similar approach for our NetCDF files. Specifically, we implement a self-managed "framed object," a virtual, large (perhaps more than five gigabyte) object, identified by its name (no manifest file is needed) and partitioned across a set of physical frames stored one per S3 object, with names chosen to describe a subfolder-like organization. Each frame is identified by a "sub-name" coding the number of the frame, the total number of frames, and the frame size. Because object names are limited in size in 255 characters, we encode numbers in base 62, using all printable characters compatible with internet URLs in the following orders: 10 digits, 26 lower case characters, and 26 upper case characters. In order to increase the reliability of the service, each frame is digitally signed by MD5. Each time a frame is retrieved it is checked against the signature and re-requested if an error is detected.

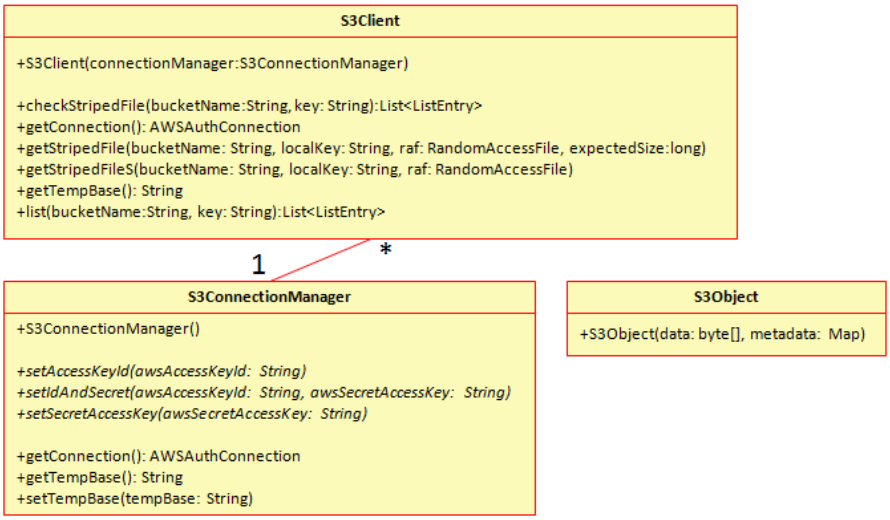


Figure 1. Part of the enhanced S3 Java interface class diagram.

Because saving space in S3 can both reduce costs (for storage and data transfer) and improve performance, we compress each frame using the zip algorithm. S3 supports high performance concurrent access, so we implement all writing and reading operations using a shared blocking queue. Our API manages framed objects in both write and read operations. A developer sees each read and write operation as atomic. When an object is to be stored on S3 using the framed approach, it is divided into frames each of a size previously evaluated for best performance,

plus a last smaller frame for any remaining data. Then, each MD5-signed and compressed frame is added to the queue to be written to S3. The queue is consumed by a pool of worker threads. The number of threads depends on the deployment conditions and can be tuned for best performance. Each worker thread can perform both write and read operations. The framed object writing operation can be blocking or nonblocking. In the blocking case the caller program waits until each frame is correctly stored on S3; in the nonblocking case, it is notified when the operation terminates.

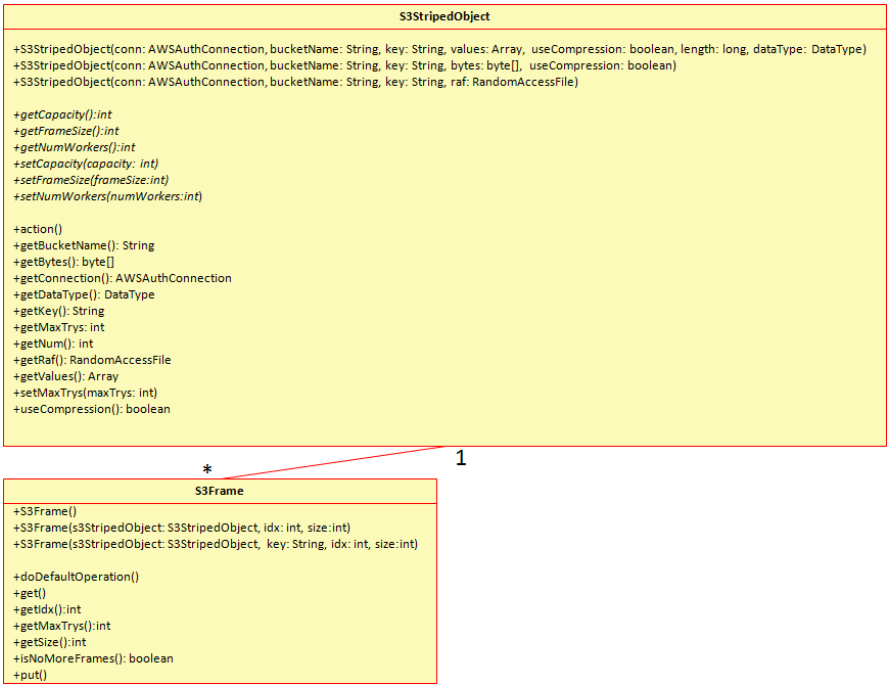


Figure 2. S3StripedObject class diagram

The developer also sees the framed object read operation as an atomic operation. This operation first extracts object features (size, total amount of frames, size of each frame plus the rest frame size) from the name of a stored frame. Then the object is allocated in client memory and the required read operations are submitted to the queue. The worker threads concurrently check for frame integrity using the MD5 signature, uncompress the frame, and place it in the right position in memory. The framed object read operation, like the write operation, can be either blocking or nonblocking (Figure 2).

The S3-enhanced Java API that we developed can be used as a middleware interface to S3, on which we can then build higher-level software such as our S3-enabled NetCDF Java interface.

3.4 Enabling the NetCDF Java interface to S3

The NetCDF-Java library supports access to environmental datasets stored both locally, via file system operations, and remotely, via protocols such as HTTP and OpenDAP. Data access is performed via an input/output service provider (IOSP) interface. The developer can build custom IOSPs that implement methods for file validity checking, file opening and closing, and data reading, in each case specifying the variable name and the section to read. An IOSP generates requests to a low-level random access file (RAF) component, a buffered drop-in replacement for the homonymous component present in the Java input/output package. The use of RAF provides substantial speed increases through the use of buffering. Unfortunately, RAFs are not structured in the NetCDF-Java library as a customizable service provider, so developers cannot build and register their own random access file component.

The NetCDF file format is self describing: data contain embedded metadata. The NetCDF Markup Language (NcML) is an XML representation of netCDF metadata. NcML is similar to the netCDF network Common data form Description Language (CDL), but uses XML syntax. A NetCDF file can be seen as a folder in which each variable can be considered a file. Thus, a NcML file description can be considered as an aggregator for variables, dimensions and attributes.

In our S3-enabled NetCDF Java Interface we use the NcML file representation as a manifest file, the NetCDF file name as a folder name, and each variable as a framed object: basically a subfolder in which each variable is stored in frames. Thus, if an S3-stored NetCDF file is named s3://bucketname/path/filename.ncml, then its data is stored in s3://bucketname/path/filename/varname/framename objects.

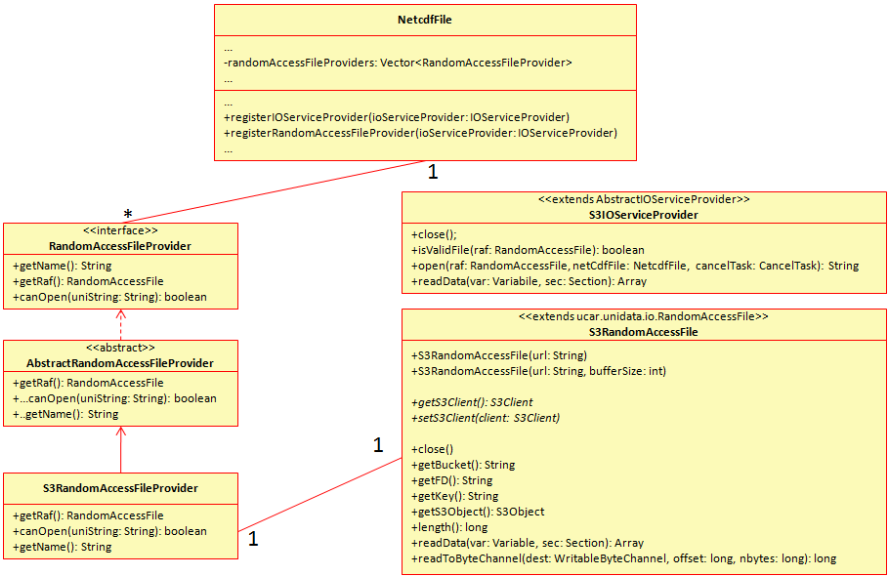


Figure 3. The S3-enabled NetCDF Java interface class diagram.

Our S3IOServiceProvider *open method* interacts with our S3-enhanced Java package to retrieve the NcML file representation and create an empty local image of the remote netCDF file using the NcMLReader component. Thus, metadata are available locally, but no variable data are actually downloaded from S3. In order to implement this behavior we create a custom RAF, which defines three methods: canOpen returns true if the passed filename can be considered as a valid file; getName returns the name of the RAF; and getRaf returns the underlying random access file component. We also provide an abstract RAF provider component that defines the RAF reference and implements the getRaf method. Finally, we implemented the S3RandomAccessFileProvider component, which returns the string "S3RAFProvider" as name and verifies that the filename begins with the "s3:" string. The canOpen method creates an instance of the S3RandomAccessFile and performs the needed initialization.

S3RandomAccessFile (S3RAF) is the key component in our S3-enabled NetCDF service. It implements the low-level interaction with our S3 Java API. When a variable section is read, S3RAF retrieves only the needed data frames stored on S3 and writes them to a temporary NetCDF local file. If more than one frame is needed, our S3 Java interface uses the framed object component to read them concurrently. A caching mechanism ensures that we do not download the same frame twice. This feature minimizes download time and reduces the number of S3 get operations and consequently the overall cost (Figure 3).

The following code reads a variable section from a NetCDF file stored on S3:

```
1: String fileName = "s3://12WREKPN1ZEX2RN4SZG2/wrfout_d01_2009-02-10_00-00-
00.nc_1_test";

2: S3ConnectionManager.setIdAndSecret("12WREKPN1ZEX2RN4SZG2", "…");

3: NetcdfFile.registerIOProvider("ucar.unidata.io.s3.S3IOServiceProvider");
4:NetcdfFile.registerRAFProvider("ucar.unidata.io.s3.S3RandomAccessFileProvide
r");

5: NetcdfFile ncfile = NetcdfFile.open(testFileName);
6: String section = "1:1:1,1:19:,1:190:1,1:254:1";
7: Array arrayResult = ncfile.findVariable("U").read(range);
```

In line 1, we define the file name. The string "s3://" identifies the protocol, allowing the S3RAFProvider component to recognize that the location can be accessed by the S3RAF and that it must create an instance of this object. The string immediately following the protocol name is the bucket name. (Because bucket names must be unique, a good strategy is to use the S3 user id.) The last part of the string is the actual file name.

In line 2, we use the S3ConnectionManager component to set the user id and password required by the S3 infrastructure. The S3ConnectionManager component also allows the developer to configure deployment and performance details, such as the temporary file path, the size of the read and write queues, and the number of worker threads.

In line 3, we register the S3IOServiceProvider using the standard NetCDF Java interface, while in line 4 we register the S3RAFProvider, which as stated above improves the standard NetCDF-Java interface to implement completely transparent access to S3-stored datasets.

In line 5, we open the file, using the same syntax as for a local operation. Line 6 specifies that we wish to read just one time step of the whole two-dimensional variable. We read variable in line 7, retrieving an Array object reference to data in the same manner as for a local data access. Observe that the underlying cloud complexity is completely hidden.

4. Cloud and grid hybridization: The NetCDF service

The NetCDF service developed by Montella et al. [16] is a GT4-based web service. It leverages useful GT4 features and captures much previous experience in environmental data delivery using grid tools. The service integrates multiple data sources and data server interaction modes, interfaces to an index service to permit discovery, and supports embedded data processing. Last but not least, it is designed to work in a hybrid cloud/grid environment.

4.1 The NetCDF service architecture

The NetCDF service provides its clients with access to *resources*: an abstracted representations of data objects that are completely disjoint from the underlying data storage associated with the data objects. A *connector* links the NetCDF service resource to a specific underlying data storage system. Available connectors include the *NetCDF file connector*, which using our S3-enhanced NetCDF Java interface can serve local files, DODS-served files, HTTP-served files and S3-stored files; the *GDS connector* that can serve Grib and GrADS files served by a Grads Data Server; and the *Hyrax connector* for OpenDAP Hyrax-based servers. We are also developing an *instrument connector* as a direct interface to data acquisition instruments [16] based on our Abstract Instrument Framework [15].

The primary purpose of a connector is to dispatch requests to different data servers and to convert all responses into NetCDF datasets (Figure 4).

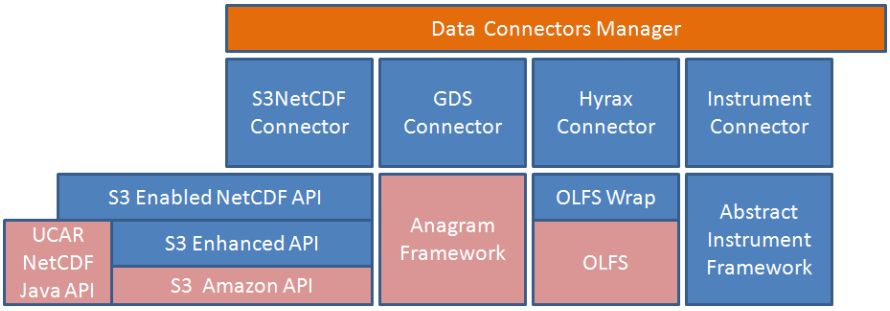


Figure 4. The NetCDF service data connector architecture.

Once a requested subset of a dataset is delivered by the data connector and stored locally, the user can process that subset using local software. This feature is implemented using another full customizable plug in. Thanks to the factory/instance approach, each web service consumer deals with its own data in a temporary private storage area physically close to the web service. The processor connector component mission is to interface different out of the process NetCDF dataset processors caring out a standard way to perform data input, processing job submission and data output (Figure 5).

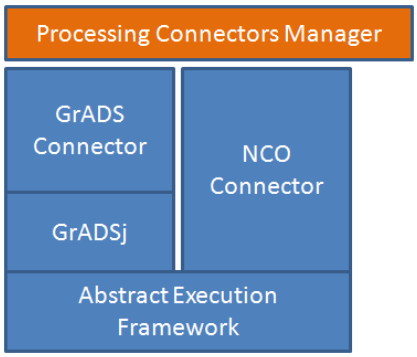


Figure 5. The NetCDF service processing connector architecture.

Two processor connectors are available. The *GrADS processor connector* provides access to NetCDF data as gridded data. A consumer can send to the processor connector complex command sequences using a common Java interface or directly with GrADS scripts. The GrADS processor connector is build on top the GrADSj Java interface we developed in previous work [17].

The *NetCDF Operator connector* is a Java interface to the homonymous software suite. The netCDF Operators, or NCO, are a suite of standalone, command-line programs that each take netCDF files as input, operate on those files (e.g., derive new data, compute averages, extract hyperslabs, manipulate metadata), and produce a netCDF output file. NCO primarily aids manipulation and analysis of gridded scientific data. The single-command style of NCO allows users to manipulate and analyze files interactively, with simple scripts that avoid some overhead (and power) of higher level programming environments. As in the case of the GrADS processor connector, the web service consumer interacts with the NCO processor connector using a simple Java interface or directly with shell-like scripts. As internally GrADSj, the NCO processor connector leverages on the AbstractExecutionFramework (AEF) we developed in order to manage the execution of out of the process software from the Java environment is a standard high level fashion [18].

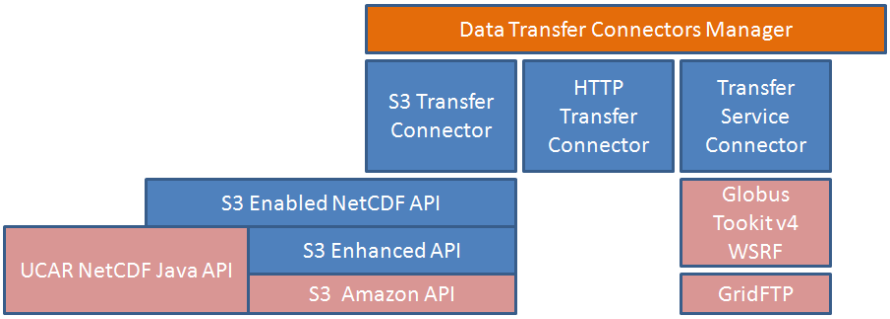


Figure 6. The NetCDF service transfer connector architecture.

Once a data subset is extracted and processed, the main actor in the data pipeline is the data transfer connector. This plug-in-like component permits the developer to customize how the selected data is made available to the user. All transfer connectors share the cache management system and the automatic publishing of data descriptions into an index service. (These features are applied automatically only when there are no privacy issues.) In general, subset data from a public available dataset are still public, but the user can set a subsetting result as private; in contrast, a processing result is private by default, but the user can declare it as public (Figure 6).

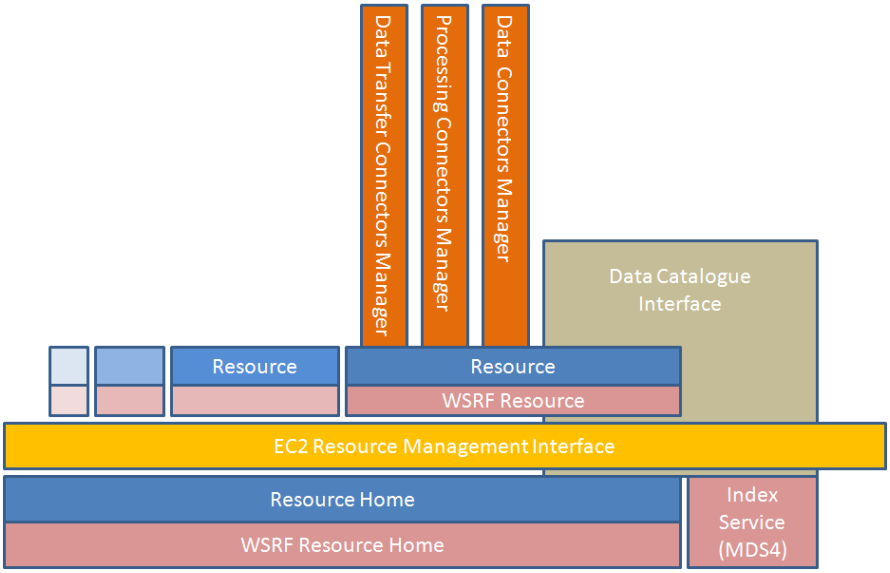


Figure 7. NetCDF service architecture: the big picture.

The caching and auto publication processes work on the two kinds of datasets in the same way. Because each NetCDF file is completely described by its metadata, each dataset can be uniquely identified by a MD5 hash of that metadata. Thus, we sign each dataset generated by a subsetting or processing operation and copy it to a caching area. Then, each time a NetCDF file is requested, we first evaluate the MD5 metadata signature and check the cache. In the case of a cache miss, the dataset is requested for subsetting or submitted for processing. Following a cache hit the dataset is copied from the cache area to the requestor's temporary storage area, and in addition the NetCDF dataset cache manager component increase the usage index of the selected dataset. If this index overcomes a chosen threshold, the dataset is promoted to be a stored dataset.

The web service resource manager explores resources and advertises them automatically on the index service. The NetCDF dataset cache manager component periodically explores the cache, decreasing the usage count of each cached NetCDF dataset. A dataset is deleted if its count reaches zero, in order to save local storage space. This usage checking process is performed even on promoted datasets. The actual storage device priority sequence, ordered from the most frequently accessed to the least is: local file system, EBS, S3 and then deleted. The policy that is used to manage data movement of these storage devices is fully configurable and based on the number of requests per unit of time multiplied by the space needs for the specific device.

The transfer connector act as the main data delivery component implementing how the web service result can be accessed by the user. The default transfer con-

nector is the TransferServiceConnector. In this way we represent the result by an End Point Reference (EPR) to a temporary resource managed by a TransferService. This service is a wrapper over the GridFTP service. The TransferService uses the Grid Security Infrastructure and works in an efficient and effective way for secure data transfer. Other available transfer connectors include the HTTPTransferConnector and the DODSTransferConnetor suitable for publically available result datasets. Finally the S3TransferConnector stores the results on S3 and then the user can access them directly (Figure 7).

4.1 NetCDF service deployment scenarios

The NetCDF Service represents the grid aggregator component for cloud hosted multidimensional environmental data resources. We explore three different deployment scenarios, in which the NetCDF service is deployed variously (see Figure 8):

1. on a computer outside the cloud (*Grid+S3*);
2. on an EC2 instance (*Cloud*); or
3. in a proxy-like configuration in which the service runs on a computer outside the cloud and automatically runs one or more EC2 instances to manage operations on datasets (*Hybrid*).

In the *Grid+S3* deployment scenario, a standalone NetCDF service runs on a server external to the cloud. This server must be powerful enough and have enough storage space to support subsetting and processing operations. Data can be hosted locally, by DODS servers on secured private networks, and/or on S3-based cloud services. Thanks to the AbstractExecutionFramework component, the processing (GrADS and NCO) software can work as jobs submitted to a local queue and execute on a high performance computing cluster. In this scenario, we use the cloud provider for data storage as in the case of locally stored large datasets and S3 stored automatically promoted cached datasets.

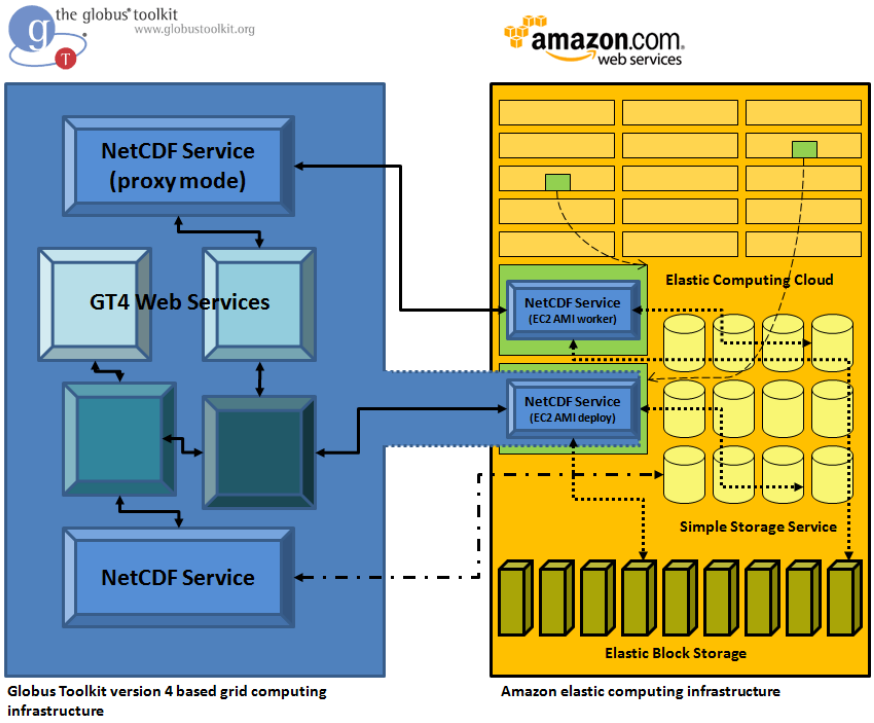


Figure 8. The three NetCDF service deployment scenarios: Grid+S3 (bottom left), Cloud (the lower NetCDF service on the right), and Hybrid (top left)

In the *Cloud* scenario, the NetCDF service is deployed on an EC2 instance, directly accessible from outside the cloud using Amazon's Elastic IP service. Data can be stored on EBS and accessed locally by the instance, while automatically promoted cached datasets can be stored on EBS and/or S3, depending on frequency of use. The EC2 instance has to provide enough computing power to satisfy dataset processor needs. The advantage of this kind of deployment is high-speed access to elastically stored datasets using EBS (faster) and S3 (slower, but accessible from outside the cloud). The main drawback to this approach is that the user is charged for the cost of a continuously running EC2 instance and the need for an assigned Elastic IP.

The *Hybrid* scenario has the highest level of grid/cloud hybridization. Here, we deploy the NetCDF service on a real computer outside the cloud infrastructure. This service acts as a proxy for another instance of the service running (as an EC2 instance) in the cloud infrastructure. When the NetCDF service receives a request, it checks to see if that EC2 instance is already running. If there is not, or if the running one is too busy (the virtual CPU's load exceeds a threshold), a new instance is created. If the data to be accessed is hosted by EBS, and an active EC2 instance is connected to the EBS where the data is hosted. Because an EBS vo-

lume can be connected to only one EC2 instance a time, the request must be directed to that

In this scenario, a consumer interacts directly only with the NetCDF service running on the real machine; the interaction with the NetCDF service running on the elastically allocated computing resource is completely transparent. The main advantages of this approach are that (a) the user is charged for running AMIs only when processing datasets and (b) there is no need for assigned Elastic IP. Moreover, the amount of data that must be transferred from the cloud to the grid is limited because most data processing is done inside the cloud. In addition, the elasticity provides for scalability and the cloud infrastructure provides for predictable quality of service, data availability, backups, and disaster recovery. Finally, the grid machine serving the NetCDF service does not need huge computing power or storage space because it acts only as a proxy providing a transparent interface to the cloud infrastructure. The main drawback is the complexity of the deployment and the increased number of interface layers.

5. Performance evaluation

All software components developed in this work are high-quality prototypes ready for real world application test beds and even production use. To evaluate the performance of the different deployment methods, we measured performance in three different scenarios: reading and writing data from S3 using the framed object implemented in our Java interface; reading NetCDF data from S3 comparing intra-cloud and extra-cloud performance; and NetCDF dataset serving comparing EBS and S3 storages.

5.1 Parameter selection for the S3-enhanced Java interface

We must evaluate the I/O performance of the S3-enhanced Java interface in order to identify an optimal framework configuration. The two developer-controllable parameters are the frame size and number of concurrent threads used as workers by the framed object components. (A third parameter, the size of the blocking queue, was empirically evaluated as four times the number of worker threads.) The choice of frame size is critical because, once set, the stored framed object must be re-uploaded in order to change it. In contrast, the number of worker threads can vary even while the application is running and can potentially be adjusted automatically based on observed network congestion and machine CPU usage.

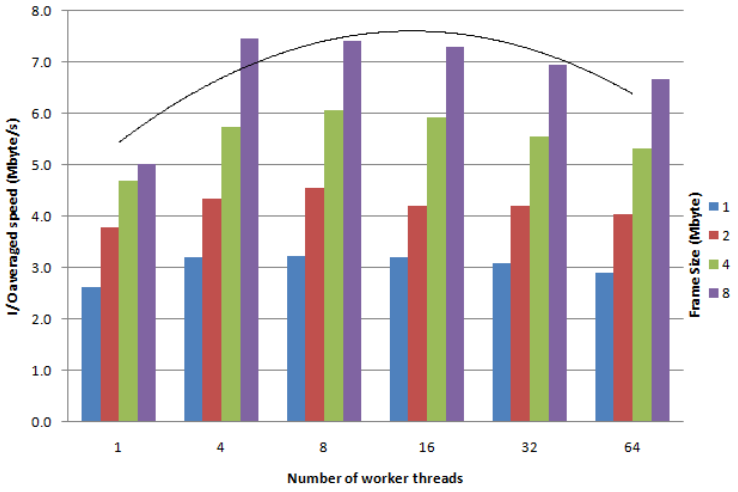


Figure 9. S3-enhanced Java interface tuning.

We used a 130 Mbyte NetCDF file to evaluate upload and download performance for a range of frame sizes (1, 2, 4, and 8 Mbytes) and thread counts (1 to 64 threads). In each case, we ran the experiment 100 times and averaged the results. We used a worst-case approach, storing the file in a US zone and accessing it from Europe (Figure 9).

The best performance for both reading and writing is achieved when using 8 megabyte frames. The best number of concurrent threads vary depends on the operation: for writing (uploading) the best performance is achieved 8 to 16 worker threads, while for reading (downloading) the maximum number of tested worker threads, 64, was the best. We have applied these lessons in an improved version of our S3-enhanced Java interface that uses a live performance evaluator to vary the number of worker threads over time.

5.2 Evaluation of S3- and EBS-enabled NetCDF Java interfaces

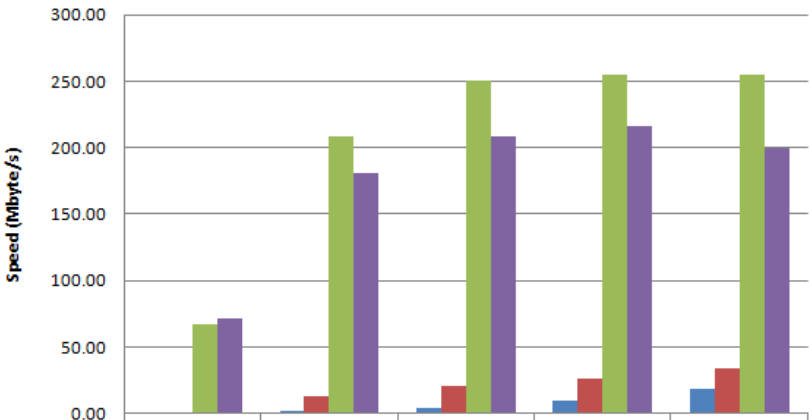
Our second set of experiments compare the performance of our S3-enabled NetCDF Java interface with analogous operations using EBS. We use a four-dimensional NetCDF file of size ~11 Gbytes produced by the WRF model on a grid with 256 x 192 cells and 28 vertical levels. This file contains six days (144 hours) of 111 variables: 14 one-dimensional, varying in time; 10 two-dimensional, varying in time and level; 72 three-dimensional, varying in time, latitude, and longitude; and 15 four-dimensional, varying in time, level, latitude, and longitude.

We define five separate tests, corresponding to reading the first one (5.5 Mbytes), 24 (132 Mbytes), 48 (264 Mbytes), 72 (396 Mbytes), and 144 hours (792 Mbytes), respectively, of a single four-dimensional variable. The subset variable is the west

to east component of the wind speed U(X, Y, Z, T), where X, Y, and Z are the spatial dimensions and T is the temporal dimension, in the order T, Z, Y, X, meaning that the disk file created for that variable contains data for all time periods for (X=0, Y=0, Z=0), then data for all time periods for (X=0, Y=0, Z=1), and so on. Thus, a request for 144 hours of the variable involves a single contiguous sequence of bytes, while a request for 1 hour of the variable involves multiple small reads. As before, we run each test 100 times and average the results.

We ran the test suite in four configurations, in which the subsetting operation is performed variously on:

1. a computer in Europe, outside the cloud, with the data on S3 in the US;
2. an EC2 virtual machine with the data on S3 in the same zone;
3. an EC2 virtual machine with data on an attached EBS volume; and
4. a server outside the cloud (a quad-core Xeon Linux based server) with the data on that computer's local disk.



	1h	24h	48h	72h	144h
local<-S3	0.10	2.11	3.97	10.21	18.57
EC2<-S3	1.19	13.22	20.92	25.96	34.23
EC2<-EBS	67.54	208.21	250.28	254.59	255.31
local<-local	71.03	181.04	208.11	216.68	199.72

Figure 10. Performance in Mbyte/sec of the S3-enabled NetCDF Java interface for different subset time periods and different data transfer scenarios.

As shown in Figure 10, performance increases in all cases with subset time period and thus read size. We also see that performance varies considerably across the four different configurations. Accessing S3 from EC2 (EC2←S3) is faster than when accessing S3 from outside the cloud (local←S3), with the difference being proportionally larger for smaller reads. This result suggests that there may be advantages to performing subsetting operations in the cloud and then moving only

the subset data to a remote client. We also see that EBS performance (EC2←EBS) is far superior to S3 performance—indeed, superior to accessing local disk (local←local). We might conclude from this result that data should always be located on EBS. However, there are other subtle issues to consider when choosing between S3 and EBS storage. If data are to be accessed only from an EC2 virtual machine, EBS is the best choice. On the other hand, S3 permits data to be accessed directly (for example, using our S3-enhanced Java NetCDF interface) from outside the cloud, and permits multiple accesses to proceed concurrently. Cost must also be considered: running instances on EC2 machines may or may not be cost effective, depending on workload. Creating virtual machine instances dynamically, only when needed for data subsetting, analysis, and transfer purposes, could be a winning choice. We return to these issues below.

5.3 Evaluation of NetCDF service performance

Our final experiments are designed to evaluate our NetCDF service from the perspectives of both cost and performance. For these studies, we developed an Amazon Web Service Simulator (AWSS), a Java framework that simulates the behavior of the Amazon EC2, EBS, and S3 components. Given specified data access and pricing profiles, this simulator generates a performance and cost forecast. To evaluate the simulator's accuracy, we compared its predicted cost against results obtained in which we ran data hosting and virtual machines instances without any kind of cloud application. We obtained cost profiles comparable with the Amazon Simple Monthly Simulator.

We evaluated the three scenarios described in Section 4.1. In *Grid+S3*, the NetCDF service is hosted on a computer outside the cloud and accesses data hosted on S3 via the Amazon HTTP-based S3 protocol.

In *Cloud*, the NetCDF service is hosted on a "standard.extralarge" EC2 instance (15 GB of memory, 8 EC2 Compute Units, 1690 GB of local instance storage, 64-bit platform) that runs continuously for the entire month. Data are hosted on 15 one terabyte EBS volumes attached to this EC2 instance.

In *Hybrid*, the NetCDF service is hosted on an EC2 instance with the same configuration as in *Cloud*. However, this instance is not run continuously but instead is created when a request arrives and then shut down after two hours unless an additional request arrives during that period.

Because *Cloud* and *Hybrid* generate subset data on EC2 instances, that data must be transferred from the cloud to the outside world. We measured performance for three different protocols—HTTP (~2.5 Mbyte/sec), gridFTP (~16 Mbyte/sec), and scp (~0.56 Mbyte/sec)—and use those numbers in our simulations.

We present results for the following configuration and workload. We assume 15 Tbyte of environmental data, already stored in the cloud; thus no data upload costs are incurred. We consider a 30-day duration (during which time data storage costs are incurred), with requests for varying subsets of the same four-dimensional data-set arriving as follows:

- Days 0-2: No requests.
- Days 3-5: 144 requests, each for a one hour subset (i.e., 5.5 Mbytes).
- Days 6-8: 144 requests, each for a 24 hour subset (132 Mbytes).
- Days 9-11: 144 requests, each for a 48 hour subset (264 Mbytes).
- Days 12-14: 144 requests, each for a 72 hour subset (396 Mbytes).
- Days 15-17: 144 requests, each for a 144 hour subset (792 Mbytes).
- Days 18-29: No requests.

Requests arrive according to a normal distribution with a mean of 0.5 hours; they total 280 Gbytes over the 30 days. We assume a worst-case situation with a machine in Europe accessing AWS resources hosted in the US West zone. We use Amazon costs as of March 2010, as summarized in Table 1.

Table 1: Amazon Web Services costs as of March 2010

Component	Cost ($US)
EC2 standard.extralarge, US East, per hour	0.68
Data transfer from outside to EC2, per Gbyte	0.15
S3 storage, per Gbyte per month	0.15
Data movement out of S3, per Gbyte	0.15
S3 10.000 get operations	0.01
EBS storage, per Gbyte per month	0.10
EBS million of I/O operations	0.10

Table 2 shows the cost predicted by our simulator for the three scenarios over the simulated month (the Hybrid has been evaluated using EBS or S3 storages), while Figure 11 shows predicted time *per request* for the different combinations of scenario, subset size, and transfer protocol.

Table 2: Simulated monthly costs for each approach

Approach	Storage ($US)	Computing ($US)	Networking ($US)	Total ($US)
Grid+S3	2211	0	50	2261
Cloud EBS	1472	482	50	2004
Hybrid EBS	1474	231	49	1754
Hybrid S3	2216	233	49	2498

Grid+S3 is the most expensive, due to its use of S3 and the associated storage and data transfer costs. Its performance is good, especially for larger subsets, because the S3 Enhanced Java interface that is used to move data outside the cloud performs well with large data selections.

Cloud is cheaper than *Grid+S3* and has better performance than *Grid+S3* (when using GridFTP for external transfers), except in the 144 hour (no subsetting) case.

Hybrid is the most cost effective because of its use of EBS to host data and its creation of EC2 instances only when required. When using GridFTP to transfer data outside the cloud, is more economical as problem size (storing, subsetting and transfer data) increases. *Grid+S3* is expensive and slow for small subsets, but becomes competitive when the subset data grow.

One factor not addressed in these results is what happens when multiple remote clients request data at the same time. In that situation, *Grid+S3* may become more competitive, as the fact that an EBS volume can be attached to only one EC2 instance at a time limits the performance of *Hybrid*.

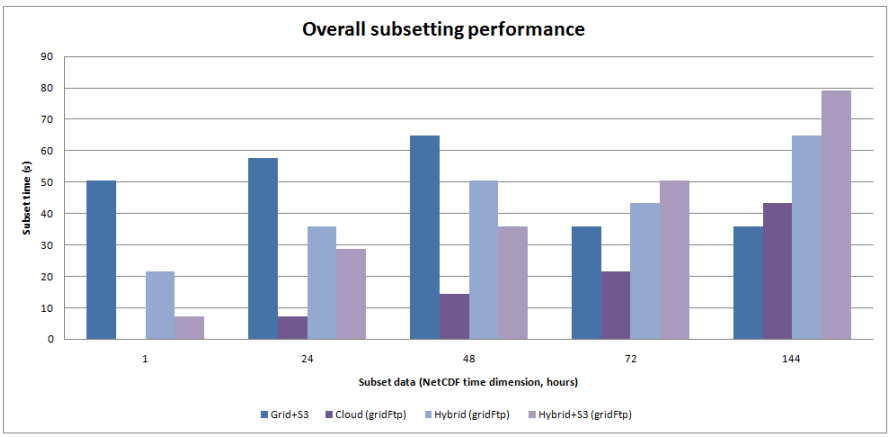


Figure 11. Modelled overall subsetting performance for each deployment scenario.

It is also instructive to compare these results with what we know of ESG. As of early 2010, ESG systems hold around 150 Terabytes of climate simulation data. During 2009, one of the two major servers operating at that time (at NCAR) delivered a total of 112 Tbytes in response to 170,886 requests, for an average of 20 requests per hour and 654 Mbytes/file. Our experimental configuration had 15 Tbytes (~1/10th of the total size) and returned a total of 280 Gbytes in response to 720 requests, for an average of 1 request per hour and 388 Mbytes/file. We need to repeat our simulations with an actual ESG workload, but an estimate based on Figure 11 suggests a cost of roughly $20,000 per month to host ESG on Amazon.

6. Conclusions and Future Directions

We have sought to answer in this chapter the question of whether it is feasible, cost-effective, and efficient to use Amazon cloud services to host large environ-

mental datasets. We believe that the answer to this question is "yes," albeit with some caveats.

To evaluate feasibility, we have developed a NetCDF service that uses a combination of external and cloud-based services to enable remote access to data hosted on Amazon S3 storage. Underpinning this service is an S3-enabled NetCDF Java API that uses a framed object abstraction to enable near random access read and write access to NetCDF datasets stored in S3 objects. MD5 signatures and compression enhance reliability and performance. The S3-enabled NetCDF-Java interface permits the developer to access both local files and S3-stored (or EBS-stored) NetCDF data in an identical manner. The use of Amazon storage is transparent with the only difference being the need to provide access credentials. Overall, this experience leads us to conclude that an Amazon-based ESG is feasible.

The NetCDF service is the product of our considerable previous work on (GT4-based) environmental data provider web services. This service is highly modular and based on a plug in architecture. Multidimensional environmental datasets are exposed as resources that furthermore are advertised automatically via an index service. The web service consumer interacts with a private, dynamically allocated instance of the service resource. The operation provider permits data selection, subsetting, processing, and transfer. Each feature is implemented by a provider component that allows for improvements, expansion, and customization. The service can be deployed in three main ways: stand alone on a machine belonging to a computing grid to distribute data hosted locally, on secured custom servers, or using the S3 service; stand alone on a virtual machine instance running in the EC2 ecosystem providing data stored on EBS or S3; and in a proxy mode running on an external computer that acts as a proxy to service instances managing resources on dynamically allocated EC2 instances.

To evaluate performance, we have conducted detailed experimental studies of data access performance for our NetCDF service in these different configurations. To evaluate cost, we developed and applied a simple cloud simulator. Based on these results, we believe that the hybrid solution represented by the third NetCDF Service deployment scenario, *Hybrid*, provides the best balance between external and cloud hosted resources. This strategy minimizes costs because EC2 instances are run only when required. Another cost reduction is achieved because there is no need for Elastic IP: the service instance running on the virtual machine connects to the one on the real machine identified by a fully qualified domain name. Two other sources of costs are the put and get data operations on S3 and the data transferred over the cloud infrastructure boundary. In *Hybrid*, most data processing is performed inside the cloud, and thus a consumer who requests subsets or analysis results retrieves less data. Less data transfer means also better performance.

The use of EBS rather than S3 provides increased data access performance. While this approach permits to use the S3 as an effective way to carry out the cloud large datasets in an effective and efficient way leveraging on concurrent S3 object

access with the Java API we developed. Finally, as the need for storage is scaled by the elastic computing approach typical for the cloud infrastructure, even the processing computing power scales with the needs thanks to the possibility of instancing as much EC2 instances hosting NetCDF Services as are needed.

A more detailed comparison of cost and performance for ESG awaits the availability of detailed ESG access logs. However, it seems that the cost of hosting the current ESG holdings on Amazon, and responding to current workloads, might be ~$20,000 month. Determining ESG's current data hosting costs would be difficult, but when all costs are considered, it may not be too different.

ESG is now preparing for the next generation of climate models that will generate petabytes of output. A detailed performance vs. costs comparison will be needed to evaluate the suitability of commercial cloud providers such as Amazon for such datasets—a task for which our simulator is well prepared. Unfortunately, this comparison is difficult or impossible to perform at present because of lack of data on future cloud configurations and costs, and on future client workloads. However, we do note that server-side analysis is expected to become increasingly important as data sizes grow, and infrastructures such as Amazon are well designed for compute-intensive analysis of large quantities of data.

This work is the result of a year of prototyping and experiments in the use of hybrid grid and cloud computing technologies for environmental data elastic storing, processing, and delivery. The NetCDF service integrates multiple software layers to identify a convenient hybrid deployment scenario. In an immediate next step, we will apply this technology to more realistic weather forecast and climate simulation scenarios. In the process, we will improve the stability of the individual components and develop further features identified by experience.

References

1. Armbrust M.,A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin. I. Stoica, M. Zaharia: Above the Clouds: A Berkeley View of Cloud Computing. Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report No. UCB/EECS-2009-28, http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html, February 10, 2009

2. Ascione I., G. Giunta, R. Montella, P. Mariani, A. Riccio: A Grid Computing Based Virtual Laboratory for Environmental Simulations. Euro-Par 2006 Parallel Processing, (W.E. Nagel, W.V. Nagel, W. Lehner, eds.) LNCS 4128, Springer (2006) 1085–1094.

3. B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, S. Tuecke: Data Management and Trans-

fer in High Performance Computational Grid Environments. Parallel Computing Journal, Vol. 28 (5), May 2002, pp. 749-771.

4. Bernholdt et al.: The Earth System Grid: Supporting the next generation of climate modeling research. Proceedings of the IEEE, vol. 93, No. 3, March 2005

5. Buyya, R. Chee Shin Yeo Venugopal, S.: Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. Proceedings the of 10th IEEE International Conference on High Performance Computing and Communications, 2008. HPCC '08.

6. Doty, B.E. and J.L. Kinter III: Geophysical Data Analysis and Visualization using GrADS. Visualization Techniques in Space and Atmospheric Sciences, eds. E.P. Szuszczewicz and J.H. Bredekamp, 1995, NASA,Washington, D.C., 209-219.

7. Foster, Ian and Zhao, Yong and Raicu, Ioan and Lu, Shiyong: Cloud Computing and Grid Computing 360-Degree Compared. Proceedings of Grid Computing Environments Workshop, 2008. GCE '08

8. Foster, Ian: What is the Grid? A Three Point Checklist. July 2002.

9. G. Giunta, G. Laccetti, R. Montella: Five Dimension Environmental Data Resource Brokering on Computational Grids and Scientific Clouds. APSCC, pp.81-88, 2008 IEEE Asia-Pacific Services Computing Conference, 2008

10. Foster I.: Globus Toolkit Version 4: Software for Service-Oriented Systems. Journal of Computational Science and Technology 21 (2006) 513–520.

11. Gallagher J., N. Potter, T. Sgouros: DAP Data Model Specification DRAFT. www.opendap.org. November 6, 2004 Rev.: 1.68

12. Gallagher J., N. Potter, P.West, J. Garcia and P. Fox: OPeNDAP's Server4: Building a High Performance Data Server for the DAP Using Existing Software. AGU meeting in San Francisco. 2006.

13. Mell P., Tim Grance. The NIST Definition of Cloud Computing, National Institute of Standards and Technology. Version 15. Information Technology Laboratory, July 2009

14. Montella R., G. Giunta, and A. Riccio. Using grid computing based components in on demand environmental data delivery. Proceedings of Upgrade Content Workshop HPDC2007. Monterey Bay. June 2007.

15. Montella R., G. Agrillo, R. Di Lauro. Abstract Instrument Framework: Java interface for instrument abstraction. DSA Technical Report. Napoli. April 2008

16. Montella R., G. Agrillo, D. Mastrangelo, M. Menna: A Globus Toolkit 4 Based Instrument Service For Environmental Data Acquisition And Distribution. Proceedings of Upgrade Content Workshop HPDC2008. Boston. June 2008

17. Montella R., G. Agrillo. GrADSj: a GrADS Java Interface. DSA Technical Report. Napoli. April 2009

18. Montella R., G. Agrillo. Abstract Execution Framework: Java interface for out of the process execution. DSA Technical Report. Napoli. June 2009

19. Rew R. K., G. P. Davis. NetCDF: An Interface for Scientific Data Access, IEEE Computer Graphics and Applications, Vol. 10, No. 4, pp. 76-82, July 1990.

20. Sotomayor B., K. Keahey, I. Foster: Combining Batch Execution and Leasing Using Virtual Machines. ACM/IEEE International Symposium on High Performance Distributed Computing 2008 (HPDC 2008), Boston, MA. June 2008.

21. Wielgosz J. and J. A. B. Doty: The grads-dods server: an open-source tool for distributed data access and analysis. 19th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology (2003).

22. Wielgosz J.: Anagram - A modular java framework for high-performance scientific data servers. 20th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology (2004).

Index terms (alphabetically):

Amazon Web Services
Cloud computing
Grid computing
Environmental datasets
NetCDF