Towards Transparent Integration of Heterogeneous Cloud Storage Platforms

Ilja Livenson
PDC, KTH Royal Institute of Technology
SE-100 44
Stockholm, Sweden
livenson@kth.se

Erwin Laure
PDC, KTH Royal Institute of Technology
SE-100 44
Stockholm, Sweden
erwinl@pdc.kth.se

ABSTRACT
The recent increase in popularity of Storage-as-a-Service cloud solutions influences the way scientists are managing their data. More and more academic groups store and work on their data outside the university domain. Hence, data aggregation and metadata management is becoming increasingly important.

While most cloud storage systems offer convenient access APIs, they are rarely fully compatible with each other. This makes migration between backends, or integration of data located in various places, more complicated. A common answer to this vendor lock-in problem lies in adopting common standards. Although there are no commonly adopted standards for cloud storage, yet, the potentially most promising direction at the moment is SNIA’s Cloud Data Management Interface (CDMI), which defines the semantics of handling data objects and the required metadata.

In this paper we present our prototype implementation of CDMI-compatible proxy, which offers a transparent integration layer on top of both cloud systems and local storage infrastructures. The work is being conducted as part of the FP7 project VENUS-C and released under an open-source license.

Categories and Subject Descriptors
D.2.12 [Interoperability]: Data mapping; E.2 [Data]: Data storage representations

General Terms
Cloud computing, storage systems, integration, metadata storage

1. INTRODUCTION
Data has become a focal element in a number of scientific areas. The black-box analysis model, where processing of a certain data item is being parametrized by the associated metadata, has become a standard approach in many areas, especially those involving machine learning and data mining. The side-effect is that the data required for processing is often located in a number of different locations, for example, internal storage, public data sets or is being generated on-demand. The same applies also to the output data.

Working with heterogeneous sources of data is both error-prone and inflicts additional work overhead on the application developer as she must cater for specifics of various data access protocols. A common solution is to provide a proxy service that includes adapters for the required data source types and manages meta-information related to the stored data. In this paper we present our approach to creating such a service with a focus on integrating Storage-as-a-Service cloud computing offerings. An important distinction from the traditional storage systems (like SAN/NAS or simple FTP servers) is a richer choice of the data abstractions. The simple file abstraction, or blob, is still the most used one, however message queues and table storage become increasingly popular.

There are several projects aiming at providing a similar offering of transparent access to multiple storage systems. Roughly, they can be divided into client libraries with connections to multiple clouds and services that encapsulate storage services on the backend and provide a common logical namespace for the client. There are pros and cons in both approaches1. An example of more widely known and adopted client-side library is Simple Cloud API [14], which provides a PHP library for working with three abstractions – blobs, message queues and tables – on top of several cloud backends. The main drawback comes from the implementation language – PHP is mostly used for web applications. Another popular library is Dasein Cloud API [6]. However, its support of data storage items is limited to blobs.

From the service side perspective, the most visible part is the interface. There are multiple projects proposing a generic API for data management services. The most known is SNIA’s Cloud Data Management Interface (CDMI) [15]. A CDMI-compliant server exposes a certain dialect of HTTP that allows to perform standard operations on data items (containers, blobs, message queues) similar to those that major cloud providers expose, for example AWS with its S3 and SQS or Windows Azure Blob and Queue. CDMI also defines associated metadata and its semantics. Moreover, this interface may also be used for managing domains, security access, and monitoring and billing information. CDMI

1The main arguments are very similar to those used in “thin” vs “fat” client case.
was chosen as the main interface for our solution. It has drawbacks, for example, table storage support is not yet available, but so far it is the most mature non-proprietary data management interface.

Other API examples include Cloud Storage Abstraction Layer [19], which defines operations for common cloud storage data services. Simple API for Grid Applications (SAGA) [13] – an abstraction of the grid computing system, which includes also a data management part, and SRM protocol [16] – a protocol commonly used in Grid environments for exposing distributed storage systems. In a way, FUSE\(^2\)-approach can also be seen as an integration solution with multiple existing implementation for various storage systems, for example, AWS S3 or Rackspace Cloud Files [9, 10]. Mounting remote storage locations to a laptop provides a convenient way for managing data resources, though limited to blobs.

This paper contributes by presenting an architecture of the data management solution that satisfies requirements of a wide class of scientific applications as well as addresses technical challenges resulting from the heterogeneity of data sources. We believe that aggregating metadata of various storage objects, especially in multi-tenant deployments, can generate added value in a number of ways. For example, optimizing the monetary costs of the cloud storage by moving files to a cheaper storage, or creating file replicas for faster and more reliable access. As such, metadata handling is one of the corner stones of our solution. The solution is being developed inside the VENUS-C project [7] and has a high priority of satisfying the needs of the project application scenarios. Nevertheless, the solution is generic and is not limited to any specific domain.

The rest of the paper is structured in the following way: we first describe the two main design decision modifiers – the data abstractions we are planning to support and possible data access methods to heterogeneous storage platforms (as well as their suitability for various application types) – and then go into more details about the architecture of the solution, including some of the implementation and deployment details. We end by describing the plans with respect to the implementation and additional functionality based on the aggregated metadata.

### 2. DATA ABSTRACTIONS

Talking about storage integration on a high level is often not so useful, because for different people the term “data” has often fundamentally different meanings. Data often refers to structured data, for example, XML of some sort, binary data (e.g. images or genome sequences) or tabular data like SQL database tables. In the scope of cloud computing, four different data storage and access mechanisms became commonplace across the vendor landscape:

- **BLOB aka blob storage (binary large object).** Blob storage is useful for storing files such as images or other binary structures. This is often used to serve as (distributed and remotely accessible) file storage. Examples are Amazon S3 [2] and Azure Blob Storage [18]. Usually, such service can handle “files” of very large sizes in the Gigabyte and Terabyte range. The term “files” is in quotes because it serves more as convenient abstraction for a large object. Commercial cloud offerings sometimes also allow to “tick a checkbox” for specific stored items to enable global distribution and caching by exposing these files in the content distribution network (CDN) of the cloud vendor.

- **Table storage.** We refer to the intersection of the large family of non-relational databases nowadays often called “NoSQL” and tables for large data items. Examples of this type include Amazon SimpleDB [3], Apache Cassandra [4] or Windows Azure Table storage [18]. In contrast to a common SQL-based database, these services lack relational query capabilities and strong transactional semantics, often providing only weak consistency. However, the typically provide massively scalable storage and high availability for huge numbers of table entries.

- **Queue storage.** Cloud storage would not be complete without queueing semantics, which are often used for guaranteed item delivery that is at-least-once delivery of work items. Examples include Amazon SQS and Azure Queue Storage. These queues are often used to decouple customer-facing interactive services (such as web sites), and long-running services, such as background workers.

- **Relational table storage (SQL).** Despite the availability of cloud table storage, some scenarios might still need the safety net provided by traditional SQL databases, such as the rich query semantics and transactional guarantees. To deal with such cases, cloud vendors usually also offer some SQL database service, often a modified version to provide the scale needed for cloud applications.

Most of the libraries that allow to use more than one cloud provider typically offer support for at least common blob and queue semantics, less frequently for table storage and SQL, because of the highly different feature sets of the corresponding implementations.

### 3. DATA ACCESS METHODS

To better understand the implications of using integrated storage solution, let’s look at the possible data access mechanisms. Figure 1 provides an example of two cloud storage systems (“A” and “B” respectively) and call chains that the user code could use for interacting with the storage. In the first case, the user code of the application “A” interacts with the cloud storages directly, using corresponding cloud SDKs. Another option is for an application developer to use a certain persistency layer for wrapping two SDKs of the cloud into a common API (the “wrapper” in the application “B”, for example, Simple Cloud API library [14] fits this call chain). The last case is to use a proxy service (e.g. a CDMI Service), which in turn dispatches the call to the concrete storage instance in interaction.

To give a real-life example, consider a bioinformatics genome alignment application with a workflow consisting of acquiring genomes of several organisms, running an alignment algorithm on them and publishing the results. Genomes are published in a number of way, for example, as files accessible on internal network service, on the FTP site of EBI or as Amazon Public Data Sets [1]. The user application

\(^2\)Filesystem in Userspace (FUSE) allows implementing a full filesystem in userspace.
could then either be capable of accessing all of the sources or use a certain generic database that aggregates information on a regular basis. A separate issue is publishing of the results: it could be uploaded to a custom location (thus creating another data source) or linked back to the aggregation database. In case the latter is local, for example, you are syncing required genomes to your local storage, this corresponds to the “wrapper” scenario in Figure 1. A remote aggregation database corresponds to the CDMI scenario.

4. ARCHITECTURE AND IMPLEMENTATION OVERVIEW

For a design of the integration platform for storage systems, as well as the follow-up implementation, to be successful, one must carefully analyse its planned usage. The following are the main aspects that were taken into account when designing the system:

- Ease of access for the client. It means that both the learning curve and the daily usage must be very easy. In practice, we try to achieve that by exposing more than one access interface. The following subsection goes into more details on the planned frontends of a service.

- Ease and flexibility of deployment for the service provider. We are accommodating for three main deployment scenarios:
  - Deployment of a service on a single machine for development purposes or for an “army-of-one” scenario when a user is not affiliated with any larger infrastructure provider, for example, university or research institute.
  - Deployment of a service inside a university for internal communities.
  - Deployment as part of the commercial or academic cloud offering.

In all of these cases abstract components could have various backends, for example, local message queue engine or insourced commercial cloud storage. We discuss deployment options in more details in Section 4.3.

At the current moment we are not aiming at supporting all of the abstractions listed above in Section 2, but rather follow the development of the CDMI specification. It means that in the near future our main goals include integration of blobs and message queues, with table storage and SQL DBs being in the pipeline.

Supporting blobs and message queues means, in short, providing well-defined interfaces for CRUD operations on blobs, and CRUD and enqueue/dequeue operations on message queues.

Supporting CDMI among other things means implementing mechanisms for security, logging, serialization/deserialization of the data, retention and hold management of the storage items.

4.1 Components

Our solution is a component-based one. Figure 2 gives an overview of the main building blocks. The first group of the components are interfaces that a service exposes. Although the main access and management interface is CDMI (described below in more details), we also expose “traditional” HTTP and FTP interfaces to allow publishing of certain blob data items to a wider community. The FUSE interface is there for convenience of the Linux and MacOS users. More custom interfaces could be added, for example, JSON-RPC [11] for a better integration with web and mobile applications.

The second group of components includes core services of the solution. These services include:

- Networking. This includes the ability of attaching to a certain socket, both local and remote, and managing network connections.

- Authentication and Authorization. This includes a simplistic implementation of the identity provider for deployments when none are available and hooks to more established authentication mechanisms, for example, LDAP or Kerberos based.

3CRUD – Create, Read, Update, Delete

4Unfortunately, so far there are no implementations of FUSE for Windows OS.
- Logging and Accounting. Requests and operations of our solution are logged. These logs are processed to create accounting information and present a user with an overview of the activities. Although not often viewed as a core component – it is not directly related to main functions of the service – in cases when storing a data can inflect direct financial cost this becomes a very important component. We see the ability of tracing down user’s operations and exposing that in a user-friendly way as a crucial for the success of our platform.

- Metainformation Datastore. It contains all of the non-content related data. For example, for blobs this is size, filename, ownership and ACL rules, retention information, as additional capabilities, for example, list of replicas or additional access protocols. For message queues metainformation includes durability of a queue and exchange type. The exact list of the metainformation items is not fixed and is implementation and deployment specific. Metainformation is used both for CRUD operations on the storage items and analysis of the aggregated information.

- Business Logic. Contains implementations of various methods relying on the metadata. These methods include, for example, calculating the size of the data indexed in the system, calculating the costs for storing the data, methods for balancing stored items according to a specified metric (e.g. achieving a high-availability by replicating data into many backends), support for 3rd party transfers and so on.

The third and the last group are actual data abstractions that follow common semantics for blobs, message queues and, in the future, table and SQL storages. Behind each of the component from this group are adapters to the services that actually implement the semantics. Table 1 summarizes the current selection of the backends for the components. Based on the adoption of the solution, we are planning to add support for more backends, for example, SRM [16] for blob backend that enabling data access to a number of distributed storage systems from the Grid environment.

<table>
<thead>
<tr>
<th>Component</th>
<th>Possible Backends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metainformation Datastore</td>
<td>CouchDB, AWS SimpleDB, MS Azure Table</td>
</tr>
<tr>
<td>Blob</td>
<td>POSIX filesystem, AWS S3, MS Azure Blob</td>
</tr>
<tr>
<td>Message Queue</td>
<td>AMQP-compliant, AWS SQS, MS Azure Queue</td>
</tr>
</tbody>
</table>

4.2 Cloud Data Management Interface

CDMI is an important part of our solution defining semantics and functionality of many components. For a better understanding, we give a brief overview of what CDMI actually is. More information and the latest specification can be accessed at http://cdmi.sniacloud.com/.

CDMI provides an interface description for performing a set of operations on the data elements from the cloud. Data elements are divided into six main object classes (as in version 1.0 of the specification):

- **Data object** – serving as an abstraction of a file or blob.
- **Queue object** – serving as an abstraction of the message queue.
- **Container object** – can be used for encapsulating other objects. Corresponds to the folder in a filesystem.
- **Domain object** – abstraction of the administrative domain inside a storage system.
```
PUT /MyContainer/MyDataObject.txt HTTP/1.1
Host: cloud.example.com
Accept: application/vnd.org.snia.cdmi.dataobject+json
Content-Type: application/vnd.org.snia.cdmi.dataobject+json
X-CDMI-Specification-Version: 1.0
{
    "mimetype" : "text/plain",
    "metadata" : {
    },
    "value" : "This is the Value of this Data Object"
}

HTTP/1.1 201 Created
Content-Type: application/vnd.org.snia.cdmi.dataobject+json
X-CDMI-Specification-Version: 1.0
{
    "objectURI" : "/MyContainer/MyDataObject.txt",
    "objectID" : "AAAAFAAo7EFMb3JlbSBpcHN1bSBkb2xvc1BzaXQgYW1ldCBhWV0g==",
    "parentURI" : "/MyContainer/",
    "domainURI" : "/cdmi_domains/MyDomain/",
    "capabilitiesURI" : "/cdmi_capabilities/DataObject",
    "completionStatus" : "Complete",
    "mimetype" : "text/plain",
    "metadata" : {
    }
}
```

Figure 3: Sample CDMI request and response for a successful creation of a new blob item.

- **Capability object** – abstraction of the interactions that a CDMI service is capable of performing for a certain object.

Using these objects it is possible to both operate on the data elements, as well as manage security access, monitoring and billing information and expose certain data items through a non-CDMI interface, for example, iSCSI or FTP. Figure 3 provides an example of CDMI request and response for creating a new blob item.

CDMI specification tries to adhere to RESTful [20] principles whenever possible to ease implementation of the interface. A CDMI-compliant solution doesn’t need not to implement the full specification, only a core set of the requirements is mandatory. Information about the limitations of the implementation is accessible through the capabilities part of the interface.

There are several limitations of CDMI:

- Small number of available implementations. Although CDMI has a good support group and is gaining momentum as an industry standard, there are still quite a few implementations available.

- Missing support of a table storage in the current version of the specification.

- Missing support of SQL DB. Most probably, it will never end in CDMI because of complications in mapping rich SQL semantics to less generic HTTP-like spirit of CDMI.

- Much smaller adoption on the client side than of more traditional interfaces. It means that exposing more than a single CDMI interface is needed to allow a lower overhead adoption of the solution for existing applications relying on HTTP/FTP-like interfaces.

4.3 Implementation details

We are implementing our solution in Python using Twisted library [17] for creating networking applications. Separation of the access interfaces (e.g. CDMI) from the actual business logic as suggested by the best practices of using Twisted was a nice fit for our architecture. We also rely on the services provided by Twisted for implementing access controls (Twisted Guard) and logging.

For connecting to the components’ backends, we either rely on the existing SDKs, for example, boto [5] for Amazon Web Services and py-amqplib [12] for talking to AMQP-compliant queue engines, or implement the missing link ourselves.

We differentiate three main component layers (example request processing is shown in Figure 4):

- **Frontend layer**. Deals with accepting and processing incoming requests and constructing valid responses. For example, CDMI-frontend would parse HTTP request, make sure that it has all the valid fields, decode the values, call corresponding business object and construct a CDMI HTTP response using environment and results of the business object operation.

- **Business object layer**, or ADT\(^5\) layer. Responsible for implementing object specific operations, for example.

\(^5\)Abstract Data Type
Table 2: Comparison of the data access methods.

<table>
<thead>
<tr>
<th>Type</th>
<th>Impl. Costs</th>
<th>Portability costs</th>
<th>Runtime Performance</th>
<th>Runtime Flexibility</th>
<th>Feature Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native SDK</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
<td>Advanced</td>
</tr>
<tr>
<td>Wrapped SDKs</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>As needed</td>
</tr>
<tr>
<td>CDMI Proxy</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Commodity only</td>
</tr>
</tbody>
</table>

![Diagram](image.png)

Figure 4: Data and metadata flow in CDMI-proxy.

Managing object metadata in the datastore, evaluating ACLs and calling selected backend for actual data storage.

- Backend layer. Executes actual communication with a specified backend. For example, an Azure Queue backend would enqueue a new value into the specified queue.

A slightly separate part of the solution is a metainformation datastore. Although most of the cloud storage systems allow to attach metainformation to the stored objects, the exact metadata and granularity can vary a lot. Moreover, we are also supporting export of a local filesystem via CDMI-proxy, which means that a certain metainformation storage system should have been implemented anyway. We decided to go with a standalone metainformation datastore, the best fit was a document-oriented database (aka table storage). The benefit of that is that it could be easily shared among deployments for higher efficiency. For example, consider the case case when application is running in AWS and the main CDMI-proxy instance is deployed in-house. We could synchronise metainformation datastore (in a slave mode) to AWS installation using standard methods of the document-oriented database. In this case, application could access metadata information in a much more efficient way.

5. DISCUSSION

Integration of multiple cloud storage systems behind a proxy service has both benefits and drawbacks. Moreover, often it is not the optimal way for accessing the data. Table 2 compares the previously mentioned data access methods with each other\(^6\). The concrete selection of a mechanism depends on various factors, specifically on available development resources, deployment agility (the necessity to run in different clouds) or the concrete performance demands of the e-Science application.

- Native SDK. If there is an SDK library supporting the programming language of choice, using that SDK is relatively cheap option, because the API is usually optimized to make data access as frictionless as possible. On the other hand, supporting another cloud provider might require adaptation of some data access code. Dynamically accessing a different cloud storage system is not foreseen in this approach. Runtime performance (throughput and latency) are best-in-class, only bound by the physical network connection. The application can use the full feature-set of the underlying cloud, which improves performance and efficiency even more.

- Wrapped SDKs. Integrating multiple SDKs into an application introduces higher development costs in the beginning, which are saved when moving to a different cloud (support for the 2nd cloud is already implemented). The SDKs exhibit the same runtime performance as the native SDK (just mapping function calls, i.e. throughput and latency are only constrained by the network connectivity). The application can dynamically use both clouds. The developer of the storage wrapper can design the wrapper API to provide a

\(^6\)“Runtime Performance” refers to throughput and latency when accessing data. “Runtime Flexibility” refers to the capability to easily access data in a different cloud storage system.
development experience which is tailored towards the specific e-Science application, i.e. the wrapper developer can optimize the API for her application code.

- **CDMI Proxy.** The implementation costs for using a CDMI proxy are relatively cheap (assuming there is a CDMI library supporting the programming language of choice). Supporting multiple cloud storage back ends is possible out of the box, because the specifics of interacting with the different cloud storage services is wrapped in the CDMI service. The data access throughput and latency will be worse compared to the vendor SDK, because the introduction of an additional intermediary means lower throughput and higher latency. Exposing multiple clouds behind a common interface, which is defined without specific requirements of the e-Science application, means that CDMI service can only expose a commodity baseline set of capabilities.

Whenever we are using external storage services, we are dealing with the issue of crossing identity domains. Using CDMI proxy approach can potentially lead to an increase of the identity domain crossings, as shown in Figure 6. This has a drawback of an increased security complexity, but a strong benefit of a much higher deployment flexibility as shown in Figure 5. For example, CDMI provider could offer internal messaging queue, but proxy requests to AWS S3 and/or Windows Azure Blob for blob specific operations. This creates a certain marketplace for possible CDMI service deployments, which is a strong vote for using an integrated storage solution. In many cases analysis of the metadata could provide insights into the optimal data storage loca-

---

**Figure 5:** Example deployment models of CDMI-proxy. Client application can talk in a homogeneous way to CDMI-proxy deployments that could either forward requests to remote backends, or use local disk. Compliance to an open standard allows a client to use other CDMI-compliant solutions – for example, CDMI-Serve developed at SARA (NL).

**Figure 6:** Authentication when crossing identity domains.
bile devices, using in-house CDMI-proxy is a much faster solution than contacting remote storage clouds directly.

- Easier release cycle. It is much easier to update a central CDMI-proxy service than a set of deployed libraries.

- Optimization effect from optimizing data of multiple users can be higher than if optimized individually (“The whole is greater than the sum of its parts”).

6. CONCLUSIONS
The system presented in this paper is currently being implemented as part of the VENUS-C project. Its validation will be done through both project scenarios [8] and applications attracted via the open call aimed at extending user community by funding 10-20 new pilot projects. The code of the CDMI-proxy is open-source and is accessible at https://github.com/livenson/vcdm.

7. ACKNOWLEDGMENTS
The authors would like to thank Christian Geuer-Pollmann for fruitful discussions and technical insights. The work is supported by EU FP7 project VENUS-C.

8. REFERENCES
[14] Simple Cloud API. simplecloud.org