A Long-Term Preservation Architecture for Big Data using Cloud Elasticity

Phillip Viana[1,2], Sam Fineberg[3], Simona Rabinovici-Cohen[2], and Liria M. Sato[1]

[1]University of Sao Paulo
[2]IBM
[3]Akamai Technologies

Abstract

Structured and unstructured data (Big Data) have been growing exponentially in enterprise storage systems. At the same time, the need for preservation of such data is increasing due to the demands of regulations and audits. These factors are driving the need for long term digital preservation of Big Data, and more specifically, the ability to extend the life of data beyond that of the systems on which it is stored. Recent research projects have proposed architectures for short term archiving of Big Data, but they lack specific support for long term preservation. In addition, they lack the elasticity needed to scale to very large data sets and to support the variety of potential ways in which data may need to be accessed. In this paper, we propose an architecture for the archiving, long-term preservation and retrieval of Big Data with elasticity. We also present our implementation of a concrete architecture containing an elasticity layer using Vagrant and Chef.

Introduction

With the increasing trend towards the collection of data of both structured and unstructured nature (Big Data) (Beath, Becerra-Fernandez, Ross, & Short, 2012) (Tallon, 2013), there comes the problem of storing the data as efficiently as possible: finding the proper media, calculating costs and ensuring that the storage containers can grow along with the data. Recent innovative technologies such as the internet of things tend to exacerbate the problem, since the volume of data tends to be even bigger.

(Herbst, Kounev, & Reussner, 2013) define the concept of elasticity as "the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible". Such adaptation to changes is sometimes

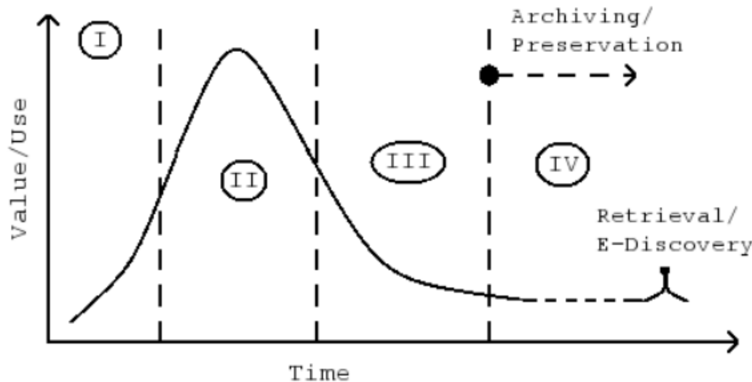

Figure 1. Information life-cycle curve, adapted from (Tallon, 2013)

necessary in a Big Data architecture if one wishes to preserve the data coming from a data stream or being ingested. When preserved data is unstructured in nature, it is common for the increase in data volume to occur so rapidly that it is necessary to increase the system's capacity at runtime to accommodate the new demand, and that's where elasticity comes into play.

We propose a cloud architecture for the archival, long-term preservation and retrieval of structured and unstructured data (Big Data) that uses an elasticity layer to provide scalability during runtime. The Big Data field presents its own set of challenges when it comes to preservation, given the heterogeneous formats that are used and the fact that unstructured data can typically grow rapidly. In addition, Big Data is often distributed across a number of different servers, which poses challenges in the search and retrieval of preserved data.

Related Work

(Tallon, 2013) addressed in depth the governance and financial aspects of Big Data, such as cost, value, return on investment, opportunity costs for data preservation and/or disposal, as well as the information life cycle curve. The information life cycle curve identifies four different regions when data is preserved, as per figure 1.

Region (I) is the point in time when the data is being created, and therefore its use and value are increasing. Region (II) corresponds to the time when the data is being consumed frequently and is usually stored in a hot medium. Region (III) represents the time when the use starts decreasing rapidly and therefore also the value of the data decreases. Finally, in region (IV) the data is archived in a cooler medium and preserved for future use. In region (IV) occasional retrievals of the preserved data occur, causing temporary blips in its use and value.

Our work focuses particularly on what happens in region (IV), that is, after the data has been preserved and is retrieved for analytic purposes.

(Factor et al., 2007) provide the definition of data preservation and differentiate between physical preservation, which is the preservation of the physical bits of data (non-corruption and prevention of data loss) and logical preservation, which is the preservation of the meaning and readability of data throughout the years.

(Pease & Rabinovici-Cohen, 2012) are the first to explicitly cover the problem of Big Data preservation, identifying scalability as one of the challenges to preservation. Since then, several researches have focused on preserving Big Data.

(Rabinovici-Cohen, Henis, Marberg, & Nagin, 2014) describe an engine to execute computational modules - storlets - within the storage container in secure sandboxes. This complements our proposed architecture in which the storlets can be used to enable efficient internal big data preservation processes such as fixity checks, data transformation to sustainable formats, provenance tracking close to where the changes happen.

(Yan, Yao, Cao, Xie, & Jiang, 2017) point emerging Big Data analysis technology as a way to preserve data that would be discarded daily. (Clua & Feldgen, 2017) propose the digitization and preservation of manuscripts as a tool for historic research. Such digitization requires the addition of metadata in order to make the information retrievable. (Bruns & Weller, 2016) identify Twitter as source of historical information about society and its thoughts, and the preservation of tweets as an artifact to be used by future historians. (Khan, Rahman, Awan, & Alam, 2016) follow a similar line of thought by focusing on the long term preservation of online newspapers and magazines and the migration to a normalized format after analytical processes are applied to the content. (McGovern, 2017) focused on data that may go extinct if not preserved, such as information about climate change that might be destroyed due to political motivations.

(Huhnlein, Korte, Langer, & Wiesmaier, 2009) and (Butler et al., 2009) present reference architectures for the long term preservation of structured data. Whilst the objective of our work is not to propose a reference architecture, the frameworks used to generate such architectures were studied and followed in order to create our proposal, particularly the method described by (Angelov, Grefen, & Greefhorst, 2012).

Self-contained Information Retention Format (SIRF)

The Self-contained Information Retention Format, or SIRF, was created by SNIA as a self-contained format for long-term preservation of data. According to (Storage Network Industry Association, 2014), the use of storage containers, such as SIRF containers, greatly increases data sustainability with a small additional cost, since the effort and financial cost to maintain data integrity is significant.

Since SNIA is an association of different storage manufacturers, SIRF was designed to be manufacturer-independent, media-independent and free. Because of these characteristics, a SIRF object will be easily interpreted by preservation systems in the future, thus reducing the costs associated with digital preservation. A SIRF object can be in any format, but its metadata is preserved in the SIRF catalog which currently is in XML or JSON formats.

Let a *storage container* be a device, partition, logical volume, directory, file system, or cloud storage device, that is, a piece or the whole of a physical storage medium that is available to store data. A *SIRF container* is a storage container dedicated exclusively to storing SIRF elements.

The SIRF catalog is the central element of a SIRF container. Every catalog contains metadata about preservation objects or *POs*, as well as metadata for the entire container. According to (Rabinovici-Cohen, Baker, Cummings, Fineberg, & Marberg, 2011), a preservation object is "a digital information object that includes the raw data to be preserved plus additional embedded or linked metadata needed to enable sustainability of the information

encoded in the raw data for decades to come". Both types of metadata (container and PO) are organized into categories.

The container metadata includes the following categories:

- Specification: this category contains information about the SIRF specification used in the container (typically the SIRF version being used).
- Container ID: the unique identifier of the container, which may be associated with the unique identifier of the storage container (for example, tape ID if using LTFS or container ID if using a cloud container).
- State: the state of the container indicates the progress of activities. If a container is currently being initialized, migrated or ready to receive new POs, the state metadata must indicate that.
- Provenance: this is a set of metadata describing the history of a SIRF container e.g., its origins and chain of custody. This metadata includes reference to one or more provenance POs in a well-documented format such as the W3C-PROV format.
- Audit Log: this category contains reference to a log of container access and modification. The content of the audit logs are by themselves POs and depend on the domain being preserved as distinct domains have different audit logs regulations e.g. FDA for the pharmaceutical domain, GDPR for personal data.

The metadata for each PO includes the following categories:

- Object IDs: each PO has one or more identifiers, which can be of four different types and are described below. Identifiers are used to identify a PO and reference other POs.
- Related objects: allows you to reference other POs (internal and external) by their UUIDs.
- Dates: used to keep information about the date and time of the PO creation, last modification and optionally last access.
- Packaging Format: is used to specify the PO packaging format. Typically POs are packaged in archival formats, such as the OAIS AIP.
- Fixity: stores the checksum of the file to ensure that media problems did not cause the PO to go corrupt.
- Audit Log: similar to the container audit log, this category stores information about how the PO was accessed and modified.
- Extension: reserved for domain-specific information. Depending on the use case and the organization using SIRF, this can be used to store extra information belonging to a specific format of the field of knowledge being preserved.

The object IDs referenced above are unique universal identifiers (UUIDs) which must be unique not only in the container, but in any context. As a general rule, the use of domain names is suggested for UUIDs. The types of identifiers are:

- Object name: the name (or one of the names) of the preservation object. This identifier is complementary, not serving as a unique or universal identifier. It can be, for example, the name of the file or an abstract, real-world name for the object. It is an optional field and can be repeated.
- Logical identifier of the object: is the common identifier to be used by different versions of an object. Different versions of the same object must have the same logical identifier. It is a required field and can not be repeated.
- Object version identifier: is the unique identifier (or identifiers) of the object version. Each object must have one or more version identifiers, so this field is required. Typically the version identifier is composed of the logical identifier plus a number that identifies the version of the object.
- Object parent identifier: used when the object is derived from another PO. This identifier identifies the object from which it was derived. This element is optional, but if it exists it must be unique.

OpenSIRF: a reference implementation of SIRF

OpenSIRF is the reference implementation of the SIRF format in Java (Viana & Sato, 2014). Two main components provide the functionality necessary to manipulate a SIRF container:

- The Core component, which contains the Java classes that model the SIRF catalog and its elements.
- The REST component, which exposes a REST API to manipulate the SIRF catalog and its objects.

The OpenSIRF Core component matches the SIRF specification exactly, meaning it contains the Java beans for the SIRF container, magic object, catalog, PO, and all categories. The Java classes in the Core component are JAXB-annotated, allowing easy marshalling to and unmarshalling from JSON and XML.

The OpenSIRF Server component defines an API for manipulating SIRF elements. Since OpenSIRF Server is deployed through a web application server (for example, Apache Tomcat or WebSphere Application Server), other software components can easily integrate with the OpenSIRF Server and therefore interact with a SIRF container.

The OpenSIRF Server uses the Core classes to model the elements being read from and written to by the API. The calls sent back to the client that contain SIRF elements always have a content type header of either XML or JSON. The calls received from clients also always have content type of XML or JSON.

OpenSIRF supports the use of multiple storage containers. Each type of storage container has a driver implemented in OpenSIRF, and the Strategy design pattern is used to

define which driver is selected at runtime based on the target storage container that will be used. The Java code has an `IStorageContainerStrategy` interface that defines the operations that a strategy must be able to perform. These operations are abstract operations on a SIRF container, such as container creation, magic object retrieval, PO submission, among others. The `ISirfDriver` interface is responsible for providing support for several types of media. The interface defines the signature of low-level operations that must be performed at the storage layer. The implementation of different storage containers such as file system or OpenStack Swift is done by `implement`ing the interface in the Java code. Currently OpenSIRF supports file systems including LTFS (Linear Tape File System), magnetic disks and OpenStack Swift, and in order to support other types of media a driver must be implemented. The `ISirfDriver` interface is defined as follows:

```
public void createContainerAndMagicObject(String containerName);

public MagicObject containerMetadata(String storageContainer, String
    sirfContainer);

public InputStream getFileInputStream(String container, String filename)
    throws IOException;

public void uploadObjectFromString(String containerName, String fileName,
    String content);

public void uploadObjectFromByteArray(String containerName, String fileName
    , byte[] b);

public void deleteContainer(String containerName);

public void deleteObject(String containerName, String objectName);

public Set<Container> listContainers();

public void close() throws IOException;
```

The `createContainerAndMagicObject()` method is responsible for bootstrapping the SIRF container on the medium. The magic object is created along with the SIRF container.

The `containerMetadata()` method returns the magic object in JSON or XML format through the marshalling of a `MagicObject` object of OpenSIRF Core. The object is created depending on where the metadata is stored in the physical medium: for OpenStack Swift, the magic object is stored as container metadata; for all other formats, it is stored as a file with name magic.json.

The `getFileInputStream()` method opens an octet stream from the storage container to the HTTP client. This is used to retrieve the binary contents of the POs.

The `uploadObjectFromString()` method transforms a string parameter into a byte array and invokes the `uploadObjectFromByteArray()` method.

The `uploadObjectFromByteArray()` method uploads a byte array of data to a storage container, thus storing the PO in the target medium.

The `deleteContainer()` method removes the SIRF container from a storage container.

The `deleteObject()` method removes a PO from the storage container.

OpenSIRF supports the storage of POs across multiple heterogeneous storage media. This is an important requirement for data preservation since a preservation architecture running for decades will likely require support for different kinds of media that may not exist today. For instance, a data preservation system bootstrapped 20 years ago didn't support solid-state drives since they didn't exist at the time, but the requirement for SSD support may exist today.

The support for multiple heterogeneous types of storage in OpenSIRF is done using the Strategy design pattern and is called `MultiContainerStrategy`. The multi-container strategy is responsible for deciding the target read or write medium at runtime. Once the medium is decided, a driver for that type of media is instantiated and thus the low-level implementation of the container can be accessed. The decision of which medium will be used for the operation depends on a so-called distribution policy. A distribution policy defines how the data is distributed across multiple containers of different types. For instance, the `EVENLY_FREE` policy will store data so that all containers in the architecture will tend to have the same amount of free space at any point in time, while the `SERIAL` policy will use up all the space of a container before storing data into a new container.

POs are found in a multi-container configuration through the use of the `MultiContainerIndex` class. This is a special type of index that always resides in the catalog node, and associates a PO's UUID with the storage container ID. This association happens during write operations. Note that a single PO cannot be spanned across multiple storage containers.

An Architecture for the Long-term Preservation of Big Data in the Cloud

Figure 2 shows our proposed architecture. It contains five layers: the data layer, the REST layer, the archival layer, the storage layer and the elasticity layer which contains the elasticity mechanism.

The bottom layer (storage layer) is where the physical contents of the POs and their metadata are stored. Some types of media are exemplified in the picture, however, this layer is abstract. Currently OpenSIRF supports file systems, magnetic disks and OpenStack Swift, but in the future new types of media will emerge and the storage layer must be ready to support the new formats through the implementation of new drivers.

The archival layer contains components necessary for the SIRF data processing. The logical preservation component has the responsibility of ensuring that the physical data is readable by software, while the physical preservation component ensures that the data is not lost or corrupted. The retrieval and archival components are responsible for reading from and writing to the storage layer, respectively. This layer transforms physical data stored in a medium into logical information (POs with metadata) and returns the information to the REST layer marshalled as XML or JSON (plus the binary contents).

The REST layer is the point of contact between a SIRF container and clients in the external world. It exposes a RESTful interface through GET, POST, PUT, and DELETE commands. According to what is received by this layer, the data is packaged in a preservation object format and sent to the archival layer (including new data), or the archival layer is directly accessed (data deletion or data retrieval). The packaging component sits in this layer, and is responsible for receiving metadata from the other layers and encapsulating

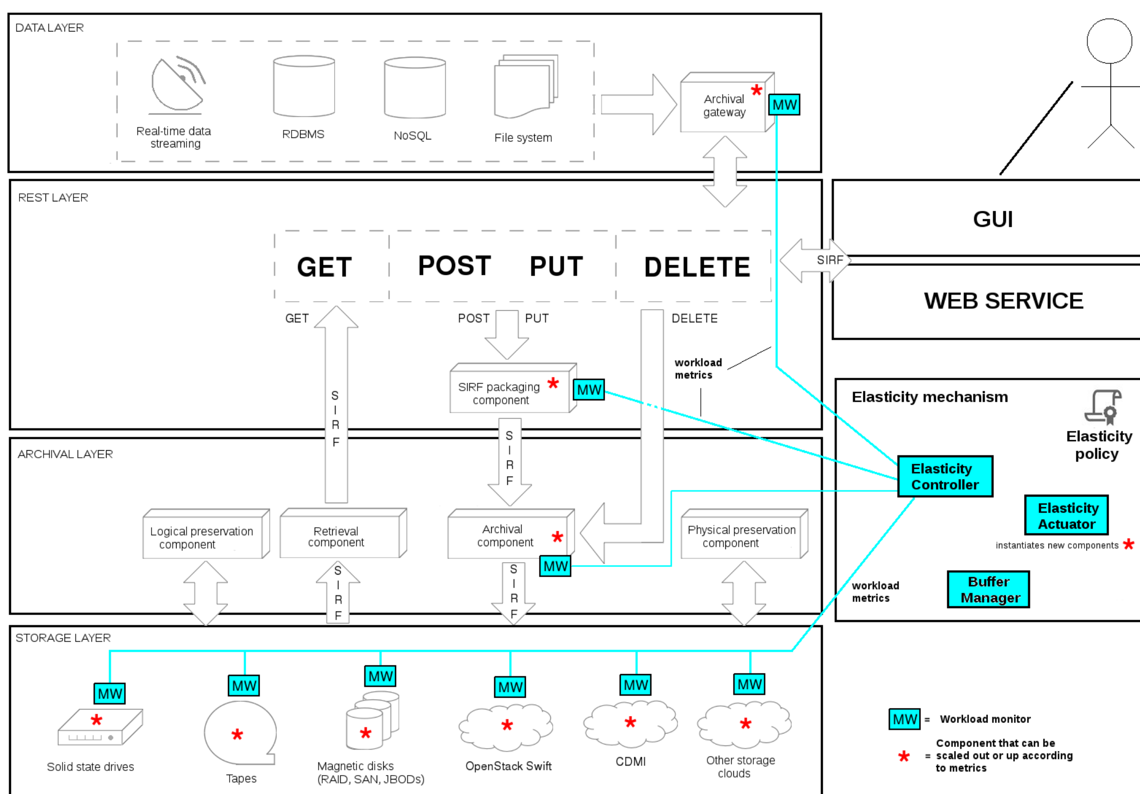

Figure 2. Diagram of the proposed architecture

them in the preservation object format for the SIRF container. The REST layer can be contacted by other systems via a web service or via a graphical user interface.

The Big Data systems reside in the data layer. In this layer are data considered "hot", that is, with high utilization and value, therefore data coming from relational databases, analytical processing and data mining tools, data streams, unstructured data processed by MapReduce workloads, NoSQL, files present in file systems, among many other possible data sources. The data of this layer are part of the areas (I), (II) and (III) of figure 1 given its high utilization and its high value. The data layer also contains the archiving gateway, a component responsible for pulling information from the various Big Data systems and sending the data to the lower layers for archival and preservation. The archiving gateway pre-processes the Big Data from the various systems, generating metadata accordingly and sending the information to the REST layer. Note that the specification of the data layer is abstract: there is no concrete recommendation of which systems the data layer must be comprised of. The implementation of the data layer will depend on features of the environment, such as the application domain, the organization that is using the preservation system, and the user group.

The elasticity layer

(Toeroe, Pawar, & Khendek, 2014) propose a mechanism of elasticity, responsible for reacting to variations of workload. Such approach has inspired the elasticity management

that will be described in the sections that follow.

In the lower right corner of figure 2 is the elasticity layer which contains the elasticity mechanism, responsible for tracking the workload of the various components in the architecture and deciding whether new components need to be instantiated or existing components need to be extended. The monitoring of the various components is done via the workload monitors, which are microservices running on the various OpenSIRF components that frequently track the resource usage of the component. In the case of storage devices, the capacity of the medium is monitored; in the case of the archival gateway, packaging component and archival component, runtime information such as CPU usage, heap and stack are monitored, so that new components can be instantiated to provide load balancing. The allocation and deallocation of resources happen according to an elasticity policy so that the architecture can accommodate and integrate new resources at runtime.

Details about the various elasticity components are given below.

Elasticity policy. The elasticity policy is a machine-readable policy document that defines limits of workload metrics and elasticity actions for the different possible elasticity states: underprovisioned, overprovisioned and optimal. The policy is read by the elasticity controller every time the metrics from the workload monitors are evaluated in order to make a decision of allocating or deallocating resources. In practice, the elasticity policy defines what the underprovisioned and overprovisioned states mean in a given preservation architecture. The capacity limit of a resource is defined in terms of the percentage of usage. For example, the storage layer may be considered underprovisioned if it reaches 90% or 95% of its capacity (this threshold is configurable in the policy). Similarly, the optimal state is also defined in the elasticity policy and is typically a range of additional resources necessary to expedite future bootstrappings. One or more active resources will be pre-allocated in the buffer waiting to be bootstrapped.

Elasticity controller. In the proposed architecture, elasticity actions are orchestrated by the elasticity controller.

The elasticity controller has the function of receiving the metrics from the monitors, interpreting the elasticity policy and requiring the allocation or deallocation of resources. It is the central piece of the elasticity mechanism, and is responsible for defining, during runtime, whether the architecture is underprovisioned, overprovisioned or in optimal state. This is done by consulting the elasticity policy. If the architecture is underprovisioned or overprovisioned, the controller creates requests for the allocation or deallocation of resources in order to accommodate the workload at the lowest possible cost.

If a new resource must be provisioned, there may be one instance of the resource already available in the buffer. If the resource is already allocated in the buffer, it is then selected. If not, the elasticity controller requests the allocation of a new resource through a call to the elasticity actuator and then selects it. By default, a workload monitor is installed alongside the new resource (if a monitor workload is applicable for that type of component). Once the new resource is selected, regardless of whether it came from the buffer or a new allocation, it must be bootstrapped in the preservation architecture. This means the new resource will be visible as a storage container, archiving gateway, or archival component. If the newly allocated resource is a storage medium, the data may need to be redistributed according to the distribution policy.

Workload monitor. A workload monitor is a component able to detect fluctuations in the workload of instances of the monitored service (Toeroe et al., 2014). For storage components, a workload monitor captures metrics about the usage of the media. In our proposal the workload monitors were implemented as microservices which expose a REST interface. The service runs continuously listening on a port. The elasticity controller sends requests via the REST interface and at that point the workload monitor will collect the metrics by accessing the operating system directly (e.g., will capture CPU or RAM usage information, free disk space, etc.).

Elasticity Actuator. The elasticity actuator has the function of allocating and deallocating resources according to the decisions made by the elasticity controller. When an allocation happens, the elasticity actuator also bootstraps the new server in the architecture. The list of tasks performed by the actuator is as follows:

- Determine the internal components (software installed on the operating system) of the resource to be allocated.
- Provision a new server (usually a virtual machine).
- Install the required software on the new resource.
- Configure network connections for the new resource: network interface, domain names, synchronization with the clocks of other servers in the architecture, firewall and operating system services (e.g. ssh, HTTP servers for communication with other architecture components, infrastructure as code)

After the allocation or deallocation of resources by the elasticity actuator, the elasticity controller is responsible for configuring the new feature of the preservation architecture (configuration of the resource as part of storage, data distribution, among other tasks).

Buffer Manager. The buffer manager allocates additional nodes in advance of an elasticity operation, so that ideally there is always a margin of additional resources readily available to be bootstrapped when a scale-out operation occurs. The buffer manager allocates the new servers, without however bootstrapping them. No data should be preserved or processed in the new resource until it is configured as part of the preservation architecture, even if the resource is available and online. When a minimum threshold of buffer resources is reached due to servers being bootstrapped, the buffer manager must contact the elasticity controller so that a new resource is created and made available as part of the buffer.

Due the use of the buffer manager, the "optimal state" is a range of resources that can be configured according to the elasticity policy, and not an exact number.

Implementation and Results

We instantiated the proposed architecture through the use of Chef and Vagrant technologies.

Implementation of an elasticity manager

In the case of this experiment, several virtual machine images were created and uploaded to the OpenSIRF Vagrant organization. These images come with pre-installed components

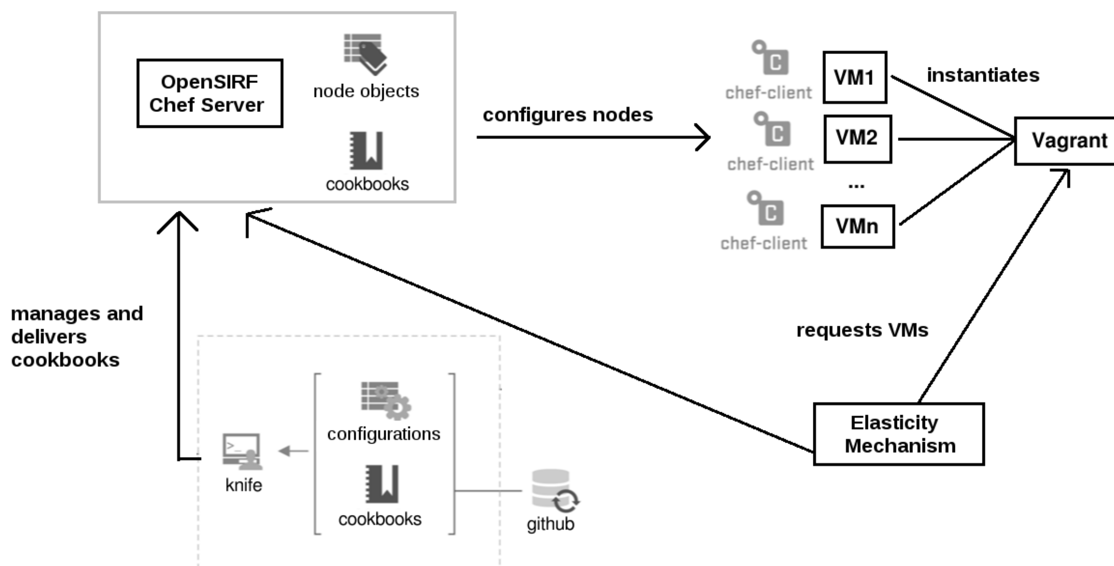

Figure 3. Chef architecture

such as OpenStack Identity, OpenStack Swift, workload monitors, a web server and the OpenSIRF Server War file. A component of the architecture can therefore be created and made available to be bootstrapped by passing a configuration file as input to Vagrant, specifying the image and attributes of the target virtual machine.

Chef is used as the infrastructure-as-code framework for the many configuration actions that take place in OpenSIRF. A central Chef server holds the cookbooks and converges the cookbooks on the servers when an action must be taken on them.

Figure 3 shows a diagram of how Chef and Vagrant are used in practice and how the combination was used to implement elasticity.

The OpenSIRF Chef Server holds all the node objects and cookbooks, which contain recipes. Recipes are specifications of files and actions a server must take in order to be configured or to be running a certain way. These cookbooks are frequently updated by the knife tool, which compiles the source of the cookbooks present in GitHub and updates the Chef Server with the latest versions.

Upon a request for scaling out or scaling up, the elasticity mechanism requests a new VM to Vagrant. The VMs come with the Chef client installed by default and can therefore be viewed by the Chef server. Once the new servers are available, the elasticity mechanism contacts the Chef server and asks for one or more cookbooks to run on the new server. These cookbooks are responsible for generic bootstrapping, such as connectivity, storage setup and data redistribution.

Steps in resource allocations. When a request is sent to the REST layer to submit a new PO (HTTP PUT or POST), the archival component will consult the distribution policy of the architecture and then map the PO to a target storage container. When this happens, the metrics related to storage capacity are by default sent to the elasticity controller, which then evaluates the metrics and the elasticity policy. If there is the need to provision new resources, the buffer manager information is retrieved in case a server is readily available to be bootstrapped. If it is, the resource is bootstrapped. If not, the elasticity actuator

instantiates a new server with a storage container.

Once the new storage container is provisioned, the elasticity actuator invokes the knife command, which accesses the Chef server and sends a request for a Chef recipe to be executed on the new server. The Chef server invokes the client in the new container, which executes the recipe for configuration, therefore making the new storage container visible to the architecture (bootstrapping process). The elasticity controller then redistributes the data, if necessary, according to the elasticity policy.

Steps in resource deallocations. When a new DELETE request is sent to the REST layer, the archival component first locates the PO in the media by verifying the index. The PO is then removed through the driver of the respective storage container. Like with the allocation, the workload monitor sends metrics by default to the elasticity controller at each request. If there are underutilized resources that can be released, the data is redistributed among the remaining storage containers according to the elasticity policy. A Chef recipe is also used to execute the steps to decommission the server.

Once the server is deallocated, the buffer manager is accessed, so that the resource that has just been released is added back to the buffer for future bootstrapping (rather than being completely removed). If, per the elasticity policy, there are sufficient resources in the buffer, the resource is completely deallocated.

Evaluation

Experiment with Twitter

The experiment revolved around capturing tweets from Twitter as a source of unstructured text, analyzing them using Apache Spark and then feeding both the pre-analytic, pure unstructured text and the analytic output of Apache Spark to different elements of the data layer. This approach led to an experiment containing both unstructured and structured data (Big Data).

The Twitter4s Scala API was used to provide the data streaming component of the data layer. It is capable of retrieving data from Twitter through both a simple REST API and a live streaming API. Virtual machines running Scala with Twitter4s were set up, continuously receiving Twitter data according to chosen keywords and hashtags (these keywords were chosen based on the most common hashtags being used at the time). Once the pure text (unstructured) data was received from the Twitter4s APIs, it was also stored in a known location on the file system as pure, unstructured text. Apache Spark was then used to analyze the data by performing map, reduce, filter and zip functions, which are typical of Big Data analytical tools.

Some of the original data coming from tweets was stored in its original form in file systems residing in the data layer. The analyzed data from Spark was consolidated into JSON objects and then stored in mongoDB and relational databases (DB2). These systems served as the data layer of the experiment, and the scenarios consisted of the archival gateway pulling data from the data layer and passing it forward to the REST layer for archival and preservation. Once the requests for preservation were received in the REST layer by the OpenSIRF Server component, it then invoked the multi-container strategy in order to write the preservation objects to the storage layer, instantiating the drivers for

each specific type of medium according to the target storage containers. The storage layer contained SSDs, hard disks and OpenStack Swift containers.

Experimental Setup

We ran the experiment using a Dell PowerEdge T30 server with Intel Xeon E3-1225 v5 (Quad-core) processor against a Linux 3.10 operating system (CentOS 7) with 32 GB of RAM. VirtualBox 5.2 was used for virtualization and Vagrant 2.0.1 was used as a virtualization manager and orchestrator. OpenSIRF Server ran on Apache Tomcat 8.

Coverage

A set of test cases were designed with the objective of validating three aspects of the research:

- The behavior of the elasticity mechanism across different elasticity states and changes between different elasticity states (underprovisioned, optimal and overprovisioned).
- Heterogeneous container (media) support.
- Real-time preservation of analytic Big Data.

A high variation of scenarios was also an important focus of the experiment:

- The four possible operations on preservation objects were executed in order to exercise the architecture code: insertion, removal, updating and retrieval of archived preservation objects.
- Statistics on the elasticity and distribution policies were collected, thus verifying that the different combinations of media and elasticity states were exercised in several opportunities.
- Tests were performed in scenarios with different combinations of media types: hard disk, SSD, Swift, hard disks+SSD, and hard disks+SSD+Swift.

Results

Several scenarios were tested according to media combination: hard disks, SSDs, OpenStack Swift and the combinations HD+SSD and HD+SSD+Swift. Every scenario typically included more than one instance of each type (i.e. two or more hard disks, two or more SSDs etc.). Figure 4 shows the number of scenarios of the experiment by media type and number of requests. "Allocation" and "Deallocation" refer to the instantiation of new resources in a scale-out manner, while "Capacity" refers to the increase of resource capacity (scale-up).

The data distribution policy was also taken into consideration in the experiment. 36.3% of the scenarios used serial distribution (data was allocated to the medium that was closer to being full at the time of allocation), while the remaining 63.7% of the scenarios used the equally free distribution policy (data was allocated with the intent of all containers having same amount of free space).

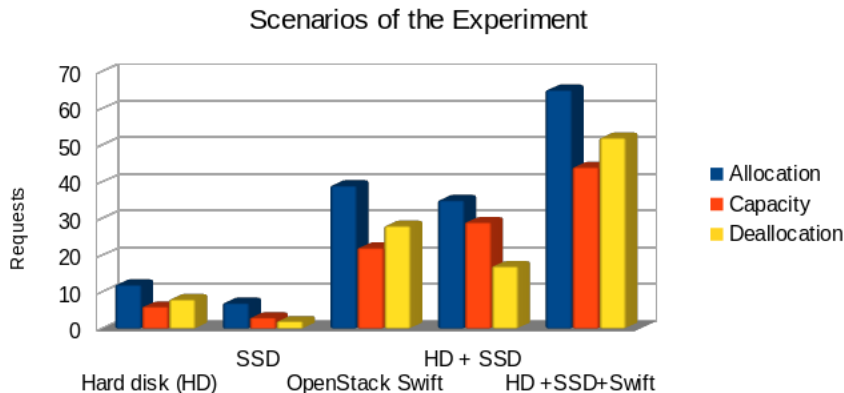

Figure 4. Scenarios of the experiment, by media combination

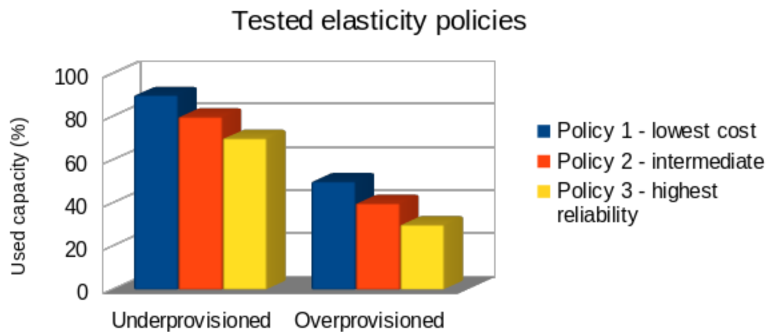

Figure 5. Number of scenarios for each elasticity policy by elasticity state

Let the underprovisioning limit be the minimum resource utilization (in %) in order for an architecture to be considered underprovisioned. Likewise, let the overprovisioning limit be the minimum resource inutilization (in %) in order for an architecture to be considered overprovisioned. Figure 5 shows the elasticity policies that were tested in the experiment. Three sample policies were defined:

- Lowest cost: avoids having resources sitting in the buffer by having a 90% underprovisioning limit and a 10% overprovisioning limit.
- Highest reliability: allocates more resources to the buffer, therefore being more readily available for scale-out operations by having a 70% underprovisioning limit and a 30% overprovisioning limit.
- Intermediate: intermediate scenario with 80% underprovisioning limit and 40% overprovisioning limit.

Figure 6 shows the number of operations on preservation objects by each type of media container (including multi-container configurations). Due to the likelihood that long-term preservation containers will have heterogeneous types of media, we focused on the execution

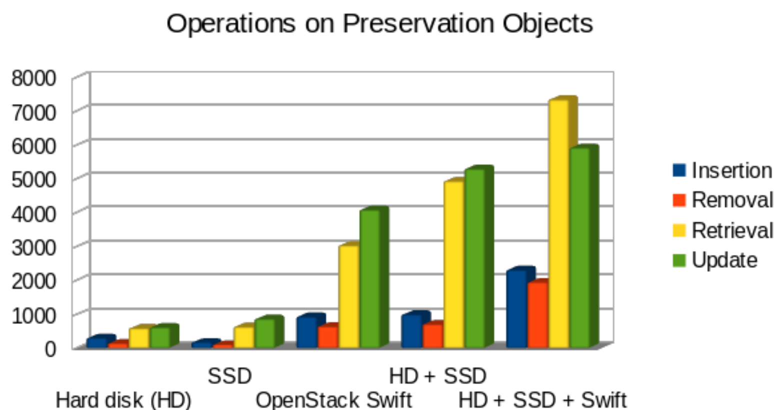

Figure 6. Number of operations on preservation objects by media combination

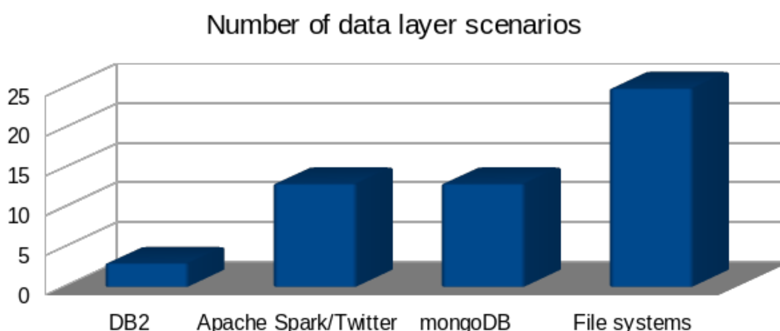

Figure 7. Number of data layer scenarios by technology

of multi-container scenarios. Also, the update and retrieval of objects was the focus of the tests since they are the most common operations on a preservation object.

Figure 7 shows the different type of scenarios for each data layer technology. Since the focus of our work was on Big Data preservation, we focused on more scenarios with unstructured text (such as files in a file system) and analytic data collected from Twitter/Apache Spark, either in its original form or condensed and recorded into a NoSQL database (mongoDB). Other scenarios were executed against a DB2 instance using the TPC-C benchmark.

Conclusion

We created a long-term preservation architecture for Big Data using elasticity in the cloud. The architecture uses elasticity to automatically adapt to Big Data workloads in real time. The SIRF format was followed to guarantee the use of metadata in the preservation domain, therefore having a full preservation system, beyond the simple archival of data.

Performance analysis was not a focus of this work, and we identify this as a topic for future work. Different performance aspects should be analyzed such as the overhead of the archival gateway and workload monitors. Another possible future work item is the spanning of preservation objects across multiple media. Currently the architecture only supports the preservation object living in one storage medium at a time, however, for bigger preservation

objects it may be necessary to span the object across multiple storage containers, indexing the object contents in a known location.

References

Angelov, S., Grefen, P., & Greefhorst, D. (2012, April). A framework for analysis and design of software reference architectures. *Inf. Softw. Technol.*, *54*(4), 417–431. Retrieved from http://dx.doi.org/10.1016/j.infsof.2011.11.009 doi: 10.1016/j.infsof.2011.11.009

Beath, C., Becerra-Fernandez, I., Ross, J., & Short, J. (2012, June). *Finding value in the information explosion.* Retrieved from https://sloanreview.mit.edu/article/finding-value-in-the-information-explosion

Bruns, A., & Weller, K. (2016). Twitter as a first draft of the present: And the challenges of preserving it for the future. In *Proceedings of the 8th acm conference on web science* (pp. 183–189). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2908131.2908174 doi: 10.1145/2908131.2908174

Butler, M., Reynolds, D., Dickinson, I., McBride, B., Grosvenor, D., & Seaborne, A. (2009, Sept). Semantic middleware for e-discovery. In *2009 ieee international conference on semantic computing* (p. 275-280). doi: 10.1109/ICSC.2009.6

Clua, O., & Feldgen, M. (2017, March). Using a research project as classroom support the case of digital preservation of degraded historic manuscripts at the university of buenos aires school of engineering. In *2017 ieee world engineering education conference (edunine)* (p. 33-37). doi: 10.1109/EDUNINE.2017.7918176

Factor, M., Naor, D., Rabinovici-Cohen, S., Ramati, L., Reshef, P., & Satran, J. (2007, January). The need for preservation aware storage: A position paper. *SIGOPS Oper. Syst. Rev.*, *41*(1), 19–23. Retrieved from http://doi.acm.org/10.1145/1228291.1228298 doi: 10.1145/1228291.1228298

Herbst, N. R., Kounev, S., & Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th international conference on autonomic computing (ICAC 13)* (pp. 23–27). San Jose, CA: USENIX.

Huhnlein, D., Korte, U., Langer, L., & Wiesmaier, A. (2009, Dec). A comprehensive reference architecture for trustworthy long-term archiving of sensitive data. In *2009 3rd international conference on new technologies, mobility and security* (p. 1-5). doi: 10.1109/NTMS.2009.5384830

Khan, M., Rahman, A. U., Awan, M. D., & Alam, S. M. (2016, Sept). Normalizing digital news-stories for preservation. In *2016 eleventh international conference on digital information management (icdim)* (p. 85-90). doi: 10.1109/ICDIM.2016.7829785

McGovern, N. Y. (2017, June). Data rescue: Observations from an archivist. *SIGCAS Comput. Soc.*, *47*(2), 19–26. Retrieved from http://doi.acm.org/10.1145/3112644.3112648 doi: 10.1145/3112644.3112648

Pease, D., & Rabinovici-Cohen, S. (2012). *Long-term retention of big data.* Event lecture.

Rabinovici-Cohen, S., Baker, M. G., Cummings, R., Fineberg, S., & Marberg, J. (2011). Towards sirf: Self-contained information retention format. In *Proceedings of the 4th annual international conference on systems and storage* (pp. 15:1–15:10). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1987816.1987836 doi: 10.1145/1987816.1987836

Rabinovici-Cohen, S., Henis, E. A., Marberg, J., & Nagin, K. (2014). Storlet engine for executing biomedical processes within the storage system. In *Business process management workshops.*

Storage Network Industry Association. (2014). *Self-contained information retention format (sirf).* Technical Position.

Tallon, P. P. (2013, June). Corporate governance of big data: Perspectives on value, risk, and cost. *Computer*, *46*(6), 32-38. doi: 10.1109/MC.2013.155

Toeroe, M., Pawar, N., & Khendek, F. (2014, Nov). Managing application level elasticity and availability. In *10th international conference on network and service management (cnsm) and workshop* (p. 348-351). doi: 10.1109/CNSM.2014.7014191

Viana, P., & Sato, L. (2014). A proposal for a reference architecture for long-term archiving, preservation, and retrieval of big data. In *Proceedings of the 2014 ieee 13th international conference on trust, security and privacy in computing and communications* (pp. 622–629). Washington, DC, USA: IEEE Computer Society. Retrieved from `http://dx.doi.org/10.1109/TrustCom.2014.80` doi: 10.1109/TrustCom.2014.80

Yan, W., Yao, J., Cao, Q., Xie, C., & Jiang, H. (2017). Ros: A rack-based optical storage system with inline accessibility for long-term data preservation. In *Proceedings of the twelfth european conference on computer systems* (pp. 161–174). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/3064176.3064207` doi: 10.1145/3064176.3064207