Hera Object Storage: A Seamless, Automated Multi-Tiering Solution on Top of OpenStack Swift

Remo Höppli, Thomas Michael Bohnert, Leonardo Militano
ICCLab, InIT, Zürich University of Applied Sciences, Winterthur, Switzerland
e-mail: remo.hoeppli@gmail.com, thomas.bohnert@zhaw.ch, leonardo.militano@zhaw.ch

***Abstract*—Over the last couple of decades, the demand for storage in the Cloud has grown exponentially. Distributed Cloud storage and object storage for the increasing share of unstructured data, are in high focus in both academic and industrial research activities. At the same time, efficient storage and the corresponding costs are often contrasting parameters raising a trade-off problem for any proposed solution. To this aim, classifying the data in terms of access probability became a hot topic. This paper introduces Hera Object Storage, a storage system built on top of OpenStack Swift that aims at selecting the most appropriate storage tier for any object to be stored. The goal of the multi-tiering storage we propose is to be *automated* and *seamless*, guaranteeing the required storage performance at the lowest possible cost. The paper discusses the design challenges, the proposed algorithmic solutions to the scope and, based on a prototype implementation it presents a basic proof-of-concept validation.**

Keywords—Cloud Computing, Object Storage, OpenStack, Multi-tiering.

I. Introduction

Distributed Cloud storage solutions have gained high momentum over the recent years where we witnessed an exponentially increasing demand for data storage. With the widespread deployment of modern high computational empowered devices in a context of an always-to-Internet-connected world, the ability to create huge amounts of data caused this trend. The result of this is a massive volume of heterogeneous and rapidly changing data, also known as Big Data [1]. Indeed, modern applications are making deep use of the available technology, but at the same time they raise huge concerns about how to handle, and in particular, how to store all the data being produced and shared. To give some exemplifying numbers: Facebook requires each week extra 60TB of storage just for new photos [2], YouTube requires 1 Petabyte of new storage every day [3] and globally the digital universe is about doubling every two years, growing 50-fold from 2010 to 2020. The analysis presented in [4] reports also that the digital universe is expected to reach the impressive number of 40,000 exabytes in 2020. About 40% of the digital data will be stored or processed in a Cloud somewhere in its journey from originator to disposal. With a more cautious look at the data being produced, one can observe that the lion's share is taken by unstructured data (e.g., video, audio, pictures/illustration, messaging, text files and similar). This is the reason for the high interest we are witnessing in object storage for distributed storage systems in the Cloud. This form of storage foresees that a file and its respective metadata are encapsulated as an object. The metadata hosts information about the application the object is associated, level of protection, number of replicas, and geographic location.

Noteworthy, not all data is to be treated in the same way and not all storage has the same features and cost. Therefore, an efficient, cost-effective use of resources is definitely of interest. To this scope, a differentiation in the data management can be made based on how often data are accessed, how performing the storage devices are and the costs. A data temperature model classifies data according to the frequency of accessing it, with *hot* and *cold* data being very frequently and rarely accessed data respectively. Recent studies reported that the relation of the cold and hot data is in the order of 10-20% for hot data and the remaining for cold data 80%, with this latter growing with the fastest pace [6]. Optimizing the data storage allocations based on the temperature model is the objective of a so-called *multi-tiered storage*. This assigns the different categories of data to different types of storage media to optimize the cost. Tiered storage policies place the most frequently accessed data on the best performing storage and rarely accessed data on low-performing cheaper storage.

This paper is presented in the aforementioned context and introduces Hera Object Storage (HOS in short hereafter) as an agile *automated* and *seamless* multi-tiering storage system. The proposed HOS is built on top of OpenStack Swift, exploits its features and extends it in two main aspects:

- Stored objects can be moved between different storage tiers in a *seamless* way, so that availability, access performance and storage costs are optimized;
- Stored objects move to the best suited storage tier in an *automatic* way.

The remainder of the paper will discuss these aspects in details and is organized as follows. Section II presents the current state-of-the-art discussing some of the most relevant related projects. The architecture for HOS is presented in Section III, whereas Section IV introduces the proposed solutions for a seamless multi-tiering storage and object lookup. Section V presents an algorithm for the automated multi-tiering. Section VI presents a validation analysis based on a prototype implementation, whereas Section VII concludes the paper with future works and possible extensions.

II. Related Work

Several commercial products for object storage have seen the light over the last years. Among them are Microsoft Azure, Amazon S3, IBM Cloud Object Storage, NetApp StorageGRID just to mention some of the main market actors. Several solutions have been driven in parallel as open source initiatives such as Lustre, MogileFS, and OpenIO; in [7] a detailed overview on all object storage projects is reported. Among the open-source solutions, OpenStack Swift has been proven to work in large distributed environments. Objects and files in Swift are written to multiple disk drives spread throughout servers in the data center and the OpenStack software is responsible for ensuring data replication and integrity. Its success is witnessed by its availability in many OpenStack distros from big vendors such as IBM, Rackspace, Cisco, Oracle, and VMware. The popularity and wide adoption of OpenStack Swift is further supported by software defined storage vendors such as Ceph and Nexenta which provide Swift-compatible APIs to their projects.

An alternative to OpenStack Swift that attracted very much of interest from industries and academia is Ceph [8]. Ceph is as a unified, distributed storage system implementing block, file, and object storage. The RBD, Radosgw, and Ceph FS service interfaces decide, whether a block, file, or object storage is accessed. Even if several Ceph-based solutions appeared on the market in recent years (e.g., the software defined storage solution from SUSE), the architecture has a monolithic structure on the functional level, where Rados is the single service handling any kind of storage. This makes the effective construct and operation of Ceph complex to understand. Additionally, the only way to use Ceph as an object storage is to use the complete stack with no possibility of using another storage system as backend or as an additional storage [8].

A main difference between Ceph and Swift is that Ceph promotes consistency and partition tolerance over availability, whereas Swift promotes availability and partition tolerance but is eventually consistent. This practically means that in Ceph, differently than in Swift, for every data write operation an acknowledgment is sent to the client only after *all* the replicas are correctly written. To determine where and how to store the data on the storage nodes, Ceph uses the CRUSH algorithm, whereas Swift uses an approach based on hash rings (see [5] and [8] for more details). Both mechanisms have the advantage that no lookup tables have to be maintained. However, Ceph has the advantage that it already provides a sort of storage-tiering, whereas Swift only provides mechanisms to implement a storage-tiering-like solution.

In the context of multi-tiered storage, one main concern is placing an object to be stored into the right tier to satisfy the latency requirements while minimizing the storage cost. The solution to this problem is not trivial as different data types with various access patterns exist. Consider for example collected meteorology data that needs to be elaborated as fast as possible to obtain timely weather forecast. After processing the data it can be archived. Another example are large streaming platforms, where a new movie or music will be accessed extensively only immediately after creation while other objects will be popular for a long time. Also online map services, satellite imagery of popular cities may be accessed thousands of times per day, whereas rural areas might be viewed only a few times a week. Creating a system that *automatically* determines the right storage level for the data objects is not trivial and should be highly customizable.

OpenStack Swift offers the possibility to define storage policies (e.g. replication or erasure coding) through definition of different object rings so that different devices can belong to different rings [5]. The policies are implemented at the container level when it is created and for the whole lifetime. Once a container has been created with a specific policy, all objects stored in it will be stored in accordance with that policy. Even if this option actually provides a basic storage multi-tiering, unfortunately, it introduces complexity and overhead. For instance, multiple containers need to be created, at least one container for each storage policy. Then the decision where to put an object is to be taken and, information about the exact ring to address is needed when accessing the object. Nonetheless, the availability of storage policies in Swift supported our decision to design HOS on top of OpenStack Swift to inherit its features and extend it with others. To reduce the intrinsic complexity and overhead of Swift, an *automatic* and *seamless* decision on the data tier is set as the main objective of HOS. The system autonomously needs to decide whether an object should be moved to a slower cold storage or to an expensive hot storage tier. At the same time HOS will allow to access the stored object by simply accessing a single initial container, independently from the storage tier it belongs to at the moment of access.

III. Hera Object Storage Architecture Design

The HOS architecture is defined within a wider software defined storage system for cloud environments, named Hera as part of the Solidna project[1]. The general goal of Hera is to build the storage foundation for cloud infrastructures by using latest open source technologies. A first part of the project is Hera block storage (HBS), which provides highly available and scalable iSCSI block storages. HBS is based on ZFS [9] and uses erasure coding for high resilience. The block storage can be used by hypervisors and servers, as well as containers. HOS is built on the foundation of HBS. Although, it is out of the scope of this paper to give an in-depth description of HBS,

[1] https://blog.zhaw.ch/icclab/category/transfer-2/solidna/

a design requirement for HOS coming from the underlying HBS is that it should be able to use iSCSI block storage in its backend. Indeed, storage volumes presented by HBS are available as iSCSI LUNs. HBS' distributed architecture ensures that there is no single point of failure and no performance bottleneck. Therefore, it can perfectly satisfy the requirements of an object storage system like Swift. For the next higher layer, the iSCSI LUNs, provided by HBS, do not differ from any normal iSCSI LUNs. This offers the opportunity to replace this layer with other open source or proprietary block storage systems. This has the advantage that even legacy storage systems could be used with HOS.

Further design constraints for HOS derive from OpenStack Swift on which it is built. To support authentication, authorization, and accounting in HOS, we adopt *OpenStack Keystone* [10] as it is natively supported by Swift and it is well supported by other OpenStack products. Obviously, other proprietary authentication systems, such as SwiftStack Auth, SwiftStack LDAP, or SwiftStack Active Directory, could be used to integrate HOS with other already existing authentication systems.

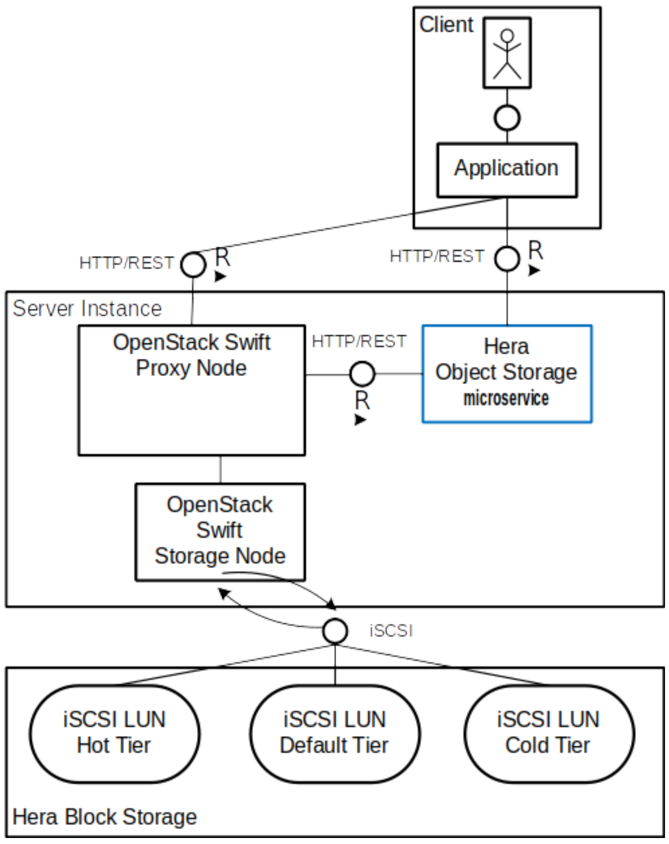


Fig. 1 - HOS architecture design (in blue the novel proposed microservice).

A key architecture design choice for HOS is the logical level of interaction with OpenStack Swift. In particular, three options are evaluated and compared:

- *Integrating HOS below Swift's logical level:* this would mean that for standard object access, the normal Swift API is used, whereas for extended functions, such as moving objects between different tiers, Swift would be bypassed with HOS. The major benefit would be that HOS does not depend on the Swift implementation, as long as the object placement algorithms in Swift do not change. The main drawback of this solution is that the whole object placement logic of Swift and the proxy logic in distributed systems need to be implemented also in HOS. Moreover, it would also require a solution to ensure that the data within Swift remains consistent when dealing with objects below Swift's logic, so that neither data nor state are corrupted.
- *Integrating HOS as a middleware in Swift:* this option is a supported feature in Swift and would allow to use Swift internal libraries, leveraging the already existing functionalities. Nevertheless, being Swift under active development, with frequent changes to its internal libraries, it would lead to updates needed in HOS for compatibility with Swift.
- *Deploying HOS as a microservice on top of Swift:* the main advantage of this loosely coupled solution is that there is no dependency on Swift-internal libraries although it exploits its functionalities. In the unlikely case that Swift REST APIs are changed, only the HOS backend would need to be redesigned. Furthermore, the whole set of features provided by Swift can be used within HOS with as little effort as doing the specific REST call. In terms of the system architecture, this option has the advantage that Swift and HOS could even be used side by side in the same environment running on independent VMs. Alternative object storage backends could also be adopted with moderate effort, as long as they provide the necessary basic functionality (e.g., Amazon S3). Moreover, the microservice design enables the system for better *scalability* if demand increases.

Considering the pros and cons of the three possible HOS-Swift integration design options, the solution best fitting to our scope is to deploy HOS as a microservice on top of Swift as sketched in Fig. 1.

The design and implementation of the HOS microservice can be divided in three main sub-problems: a) *seamless multi-tiering storage and object lookup;* b) *automated storage tiering;* c) *REST APIs and command line implementation* for programmatic interaction with HOS. To ensure that changes to the external components of the system do not impact the whole application, the design is created on a modular base, see Fig. 2. Each interface connecting HOS with the outside world is implemented as an adapter providing a defined set of functions to the core controller. If there are changes to one of the external components, only the specific adapter will need to be adjusted.

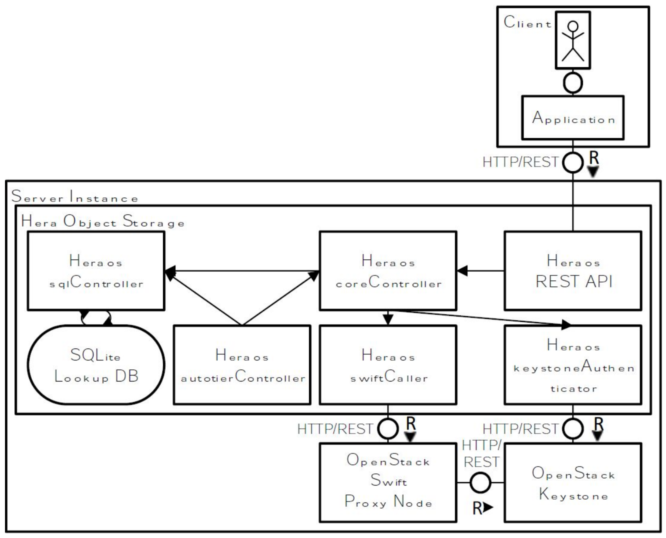


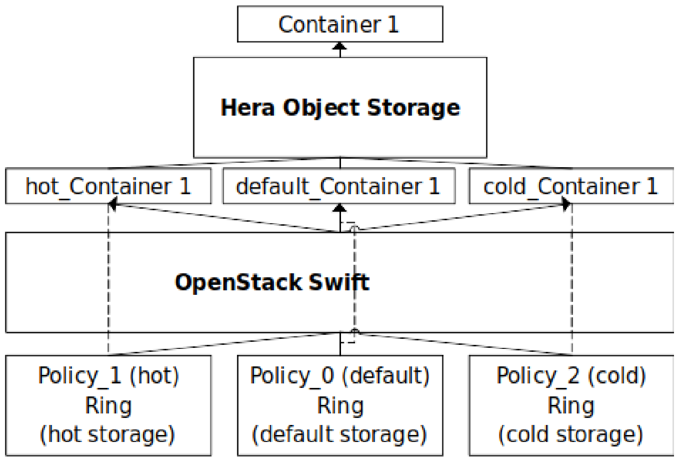
Fig. 2 - HOS modular structure.

IV. Seamless Multi-Tiering Storage and Object Lookup

For the multi-tiering functionality, we define three storage tiers which are supported by the the HBS backend: *hot storage, default storage* and *cold storage* (in decreasing order of disk performance). These storage tiers are made available to HOS exploiting the *storage policies* feature defined by OpenStack Swift. More specifically, adopting the storage policies feature we are able to define multiple object rings in Swift. Each of the storage rings can be configured to use different storage volumes and different replication levels. Additionally, storage policies allow to configure erasure coding for storing passive data that is less frequently accessed. The policies are defined at the container level which can be placed on a particular ring by applying that ring's storage policy to it. Every object stored in that specific container is then stored on the volumes associated to the specific ring [5].

To allow a seamless multi-tiering storage, we introduce an abstraction on top of the container defined in Swift. This abstraction requires that every container created on HOS needs to be created on each of the three storage tiers in Swift. More specifically, OpenStack Swift defines three storage volumes with different performance and redundancy characteristics. On each of the volumes, an object ring, configured with a specific storage policy, places its data. The three containers created in Swift are labeled with the storage policy specific of the ring it resides on. These three containers are then combined and abstracted by HOS providing a single container to the end-user as shown in Fig 3. An end-user can have multiple objects stored in a container while spreading them across all three storage tiers in the backend.

When objects have been stored into a specific tier, the HOS should be able to retrieve it again from a container when needed without necessarily providing information of the tier it resides on. A first "naive" implementation we tested out is to make HOS look for the desired object in every tier, starting with the hot storage, then checking the default tier and at least the cold tier, until the object is found. This solution is referred to as *Naive REST Lookup (NRL)* in the remaining of the paper. The drawback of this algorithm is that retrieving an object in the worst case where the object is not present requires up to four REST calls made to Swift. In fact, besides the three REST calls for to the three storage tiers, a fourth call is always needed at the beginning to authenticate the user at the AAA system and retrieve a session token. In the best case, where the requested object is on the hot tier two REST calls are needed.



Container 1
Hera Object Storage
hot_Container 1
default_Container 1
cold_Container 1
OpenStack Swift
Policy_1 (hot) Ring (hot storage)
Policy_0 (default) Ring (default storage)
Policy_2 (cold) Ring (cold storage)

Fig. 3 - Container abstraction in HOS exploiting Swift storage policies.

As an advanced efficient alternative to the *NRL* solution, we propose an SQLite-based solution. The SQLite database will provide a lookup directory for the stored objects. After the first REST call for authentication, the database is queried to get information about the Swift container storing the requested object. If the object is found in the database, the core controller obtains the information about the object's location and directly queries the correct Swift container (see Fig. 2 for the HOS modular structure). This solution is referred to as *Advanced SQLite Lookup (ASL)* in the remaining of the paper. *ASL* reduces the REST calls needed to retrieve an object to at most two in all cases. In fact, the number of REST calls for objects that are not present reduces from four to one, for cold tier objects from four to two, for default tier objects from three to two, whereas the number of REST calls for objects in the hot storage remains two. If the object retrieval was not successful, the lookup database is most likely in an inconsistent state and a repair task will be invoked.

A. Need for atomic operations in ASL

When adopting the *ASL* solution, database updates need to be atomic operations to keep the state of the database consistent with the state in Swift. Updating the objects in Swift and updating the record in the database are two separate transactions that cannot happen simultaneously. This could cause inconsistency issues. To face the problem, we introduced an object lock that only applies to update operations. The implementation of this update lock is done at the database level. Every request updating an object, reads and sets the update lock in an atomic transaction. If the update lock is already set, an exception will be thrown, indicating that this object cannot be altered at the given time. Otherwise, the update lock is set and the current process is the only one allowed to change the object state until the lock is released. Simple requests, which only retrieve information or data, will not be affected in any way by these update locks. Therefore, the read availability of the locked objects is assured.

V. Automated Storage Tiering

HOS should provide functionalities to move objects along the storage tiers according to the current access behavior. This allows for often accessed objects to reside on hot storage and rarely accessed objects to be moved on cold storage *dynamically*. At the same time, changes in an object's access behavior over time should lead to an adapted object tiering. In HOS, whenever a tier change is prompted, a Swift REST API call will be made to copy an object from one container to another within the Swift layer. After copying the object, the object should be removed from the old container. To support such an automated storage tiering, the access to the stored objects has to be monitored in order to dynamically evaluate the access characteristics and implement a wished tiering policy. As concerns the proposed HOS modular architecture, this can be supported by the SQLite database, where requests to create, update, delete or retrieve an object can be recorded along with the respective instant in time. Based on the collected information the smart automated storage tiering we have in mind has the following features:

1. The proposed algorithm should be *dynamic* so that an object can be moved between storage tiers as soon as the given rule settings dictates that;
2. The proposed algorithm should *not overreact* to prevent unnecessary object moving operations. This could be the case of an object in the hot storage tier having a short time frame where it is rarely accessed;
3. The proposed algorithm should be able to identify cases where an object is to be moved *promptly* to the hot storage tier, i.e., objects which suddenly become popular and are accessed extensively.

A first simple and straightforward solution for the automated tiering is to define for each storage tier boundary access rate values. The number of accesses to an object within an *observation period (OP)* will give this access rate value. At the end of each *OP* the objects will then be associated to one of the three available tiers. Unfortunately, although *dynamic*, such a simple approach does not meet all the wished features listed above. When the *OP* is short, objects might be moved between the tiers quite often. Moreover, it might also happen that the algorithm *overreacts*. An object that should basically reside in hot storage could have a single *OP* with only very few access attempts and thus, being moved to cold storage. Every further access attempt will then be very slow until the *OP* ends and the object is moved back to the hot storage tier. One could try to smartly tune the length of the *OP* to cope with this issue. But a longer timeframe which results in a more stable average in the access attempts, reduces the *dynamicity* of the multi-tiering storage, since objects can only be moved at the end of the given *OP*. However, in no case the third requirement listed above is addressed. For all these reasons, an advanced ad-hoc algorithmic solution has been designed as described in the remaining of this Section.

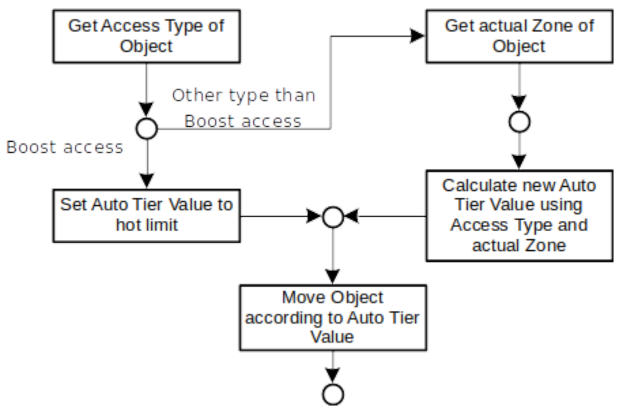
A. The object auto-tier value for multi-tier storage

Let us define the *auto-tier value* av_i associated to each object i as the parameter that determines the new tier for object i at the end of each *OP*. The value for av_i is increased or decreased in every *OP*, as a function of the last access rate to object i and the current storage tier the object is stored in. The value for *OP* is determining how often the *auto-tier value* is updated. This can be tuned according to design preferences about how quick the system should react to object access variations. The shorter *OP* is the more agile the algorithm becomes, but the more computational power is used to check and move objects. The decision about the new tier for an object is based on the schematic tier division and the respective boundaries set to the different tiers, i.e., *cold limit, default limit* and *hot limit*. When av_i is lower than the *cold limit* object i will be associated to the cold tier, when av_i is higher than the *cold limit* but lower than the *default limit* object i will be in the default tier, otherwise it will be in the hot tier.

The number of access attempts for object i during an *OP* are classified as either *low access, medium access, high access* or *boost access,* based on predefined policies. In our implementation we classify the objects into the four access classes considering also the overall behavior of all objects. In particular, we define p_b, p_h, p_m and p_l as the fractions of objects associated to boost, high, medium and low access respectively. Our proposed approach foresees the following steps to be implemented:

1. Collect all access rates for the objects within the last *OP*;
2. Order all the collected access rates in a decreasing order;
3. Classify the top p_b objects of the ordered list as *boost access*;
4. Classify the subsequent p_h objects of the ordered list as *high access*;
5. Classify the subsequent p_m objects of the ordered list as *medium access*.
6. Classify the remaining objects as *low access*.

The combination of the current tier and the so-updated access type for the object will dictate the increase/decrease of av_i. In its turn this may trigger a change in the tier for object *i*. A special case is considered for objects showing a *boost access* as the av_i will immediately be set to be equal to the *hot limit*. The diagram of auto-tier value update is reported in Fig.



Get Access Type of Object
Get actual Zone of Object
Other type than Boost access
Boost access
Set Auto Tier Value to hot limit
Calculate new Auto Tier Value using Access Type and actual Zone
Move Object according to Auto Tier Value

4.

Fig. 4 - Auto-tier value update diagram.

To avoid ping-pong effects between neighboring tiers (i.e., cold-default tier, default-hot tier), due to small changes in the access type, the default tier is further split into a *default-* zone and a *default+* zone. Correspondingly a *default-* limit and a *default+* limit are introduced, as plotted in Fig. 5. For the final object placement, objects in the *default-* and *default+* zones will then both lead to the default tier. The arrows in Fig. 5 report conceptually the object transition directions between tiers according to the current access type and the auto-tier value. A dashed arrow indicates that the object is moving into the arrow's direction over multiple *OPs*, whereas the not dashed arrow represents an object being instantly moved to the hot tier.

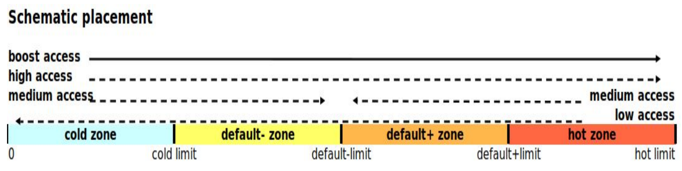


Fig. 5 - Schematic object placement.

TABLE I. GENERIC AUTO-TIER UPDATE SCHEME

	cold zone	default- zone	default+ zone	hot zone
low access	avi = avi - x11	avi = avi - x12	avi = avi - x13	avi = avi - x14
medium access	avi = avi + x21	avi = avi + x22	avi = avi - x23	avi = avi - x24
high access	avi = avi + x31	avi = avi + x32	avi = avi + x33	avi = avi + x34

Still to define is how to update the av_i after every *OP*. Several policies can be defined to this scope. In Table 1 we report a general model we have in mind, where we adopt a 3x4 matrix parameter **x** representing the increment/decrement for the auto-tier value according to the three access modes and the four storage zones.

This general scheme offers the opportunity to implement different policies and fine-tune the values for the elements of **x**. In our specific prototype implementation, the elements of **x** are proportional to the boundaries set for the different tiers and a speed factor. The speed factor represents the speed (measured in number of *OPs*) for an object to move across a given tier if it keeps the current access type. To this scope, we define as "length" of a zone associated the difference between the upper and the lower boundaries for the specific zone. As an example the length of the cold zone corresponds to the cold limit value, the length of the *default-* zone is equal to the difference between the *default- limit* value and the *cold limit* value, and so on. We also define the mentioned speed factors as s_h, s_m and s_l for hot, medium and low access types respectively. The rationale behind the proposed solution is to take into account that a frequently accessed object which is currently in a cold tier has a larger impact on the application requiring the object, than a less frequently accessed object that currently is stored in a hot tier. The proposed algorithm for the auto-tier value update is summarized in Algorithm 1.

Algorithm 1 Pseudocode to update auto-tier value av_i

```
1:  if $boost\,access == true$ then
2:      $av_i = hot\,limit$
3:  else if $high\,access == true$ then
4:      $av_i = av_i + \frac{(hot\,limit)-(default+limit)}{s_h}$
5:  else if $medium\,access == true$ then
6:      if $av_i > default - limit$ then
7:          $av_i = av_i - \frac{(default+limit)-(default-limit)}{s_m}$
8:      else
9:          $av_i = av_i + \frac{(default-limit)-(cold\,limit)}{s_m}$
10:     end if
11: else
12:     $av_i = av_i + \frac{(cold\,limit)}{s_l}$
13: end if
14: if $av_i > hot\,limit$ then
15:     $av_i = hot\,limit$
16: else if $av_i < 0$ then
17:     $av_i = 0$
18: end if
```

VI. PROOF-OF-CONCEPT IMPLEMENTATION AND VALIDATION

For the proof-of-concept implementation of the proposed solutions, HOS has been provided with a set of REST APIs. These include functions to handle object storage operations (e.g., move objects in another container or storage tier), handle the authentication procedure in an efficient and seamless way for the end-user, monitor and maintain the storage system. Also a command line client has been implemented bringing the full functionality of the HOS REST APIs to the terminal. These can be used in a similar way to the OpenStack Swift client and can be installed on any machine running Linux. Adopting the REST APIs, all dependencies on Hera-internal libraries are removed the syntax of the OpenStack Swift client is followed.

The HOS prototype we worked on is a minimal installation that presented the smallest scaling possible. However, since HOS is designed as a microservice system, it does scale in a horizontal, distributed manner. To build a scalable system, a load balancer would have to be introduced for accessing HOS. Furthermore, HOS' persistent lookup database would need to be deployed in a new database server where multiple HOS nodes can access. To avoid the database server resulting in a single point of failure, we suggest to set it up as a high availability cluster, as provided for example MySQL [12]. Considering that OpenStack Swift and HOS are designed to run as scalable systems, no further adjustments are necessary.

The proposed automated tiering policy was validated on the implemented HOS prototype in a set of simulated scenarios. To this aim, system parameters are set as reported in Table II. The reasoning behind the listed settings is that frequently accessed objects would move quickly to better tiers and less frequently objects sink slowly to lower tiers. Several object access behaviors have been simulated to ensure that the designed concept works in a practical scenario. The system worked as expected and this gave us confidence that our concept works also in a real-life scenario. Nevertheless, further validation and improvement is needed before the automated storage tiering can be used in a productive environment. On the other hand, alternative policies and parameters tuning can be proposed in our HOS prototype for an optimized multi-tiering solution.

TABLE II. NOTATIONS AND VALUES

Parameter	Notation	Value
observation period	*OP*	20 s
low fraction of objects	p_l	60%
medium fraction of objects	p_m	20%
hot fraction of objects	p_h	15%
boost fraction of objects	p_b	5%
cold limit		50
default- limit		100
default+ limit		150
hot limit		200
trans. speed hot access	s_h	2
trans. speed medium access	s_m	5
trans. speed low access	s_l	10

B. Object Lookup performance evaluation

In this Section we report on the evaluation of object lookup solutions we introduced in Section IV. To this aim, two different analysis are presented which focus on the time to retrieve objects from the storage system. Noteworthy, the results reported in this paper strictly depend on factors such as memory, CPU and NIC of the HOS node, of the Swift backend and the HBS, network connections between the parts and the disks attached to the HBS. Therefore, the measured results yield as an example and are valid for the specific system we considered in our prototype implementation. However, we don't lose generality in our analysis since we compare the two object lookup solutions against each other on the same system configuration.

In the analysis we present, each scenario is repeated 20 times and the results are then averaged. To this scope, the *unittest* Python module was adopted [10]. In our first analysis we wish to compare the *NRL* and *ASL* solutions in terms of time to retrieve an object from a container when this is stored on the different available tiers. The results are reported in Fig. 6 and include also the case where the queried object is not present in any of the tiers.

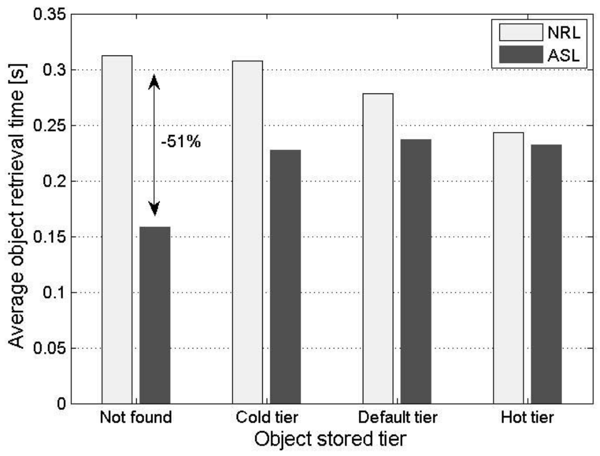


Fig. 6 - Average object lookup time for *NRL* and *ASL* solutions in retrieving an object from the different storage tiers.

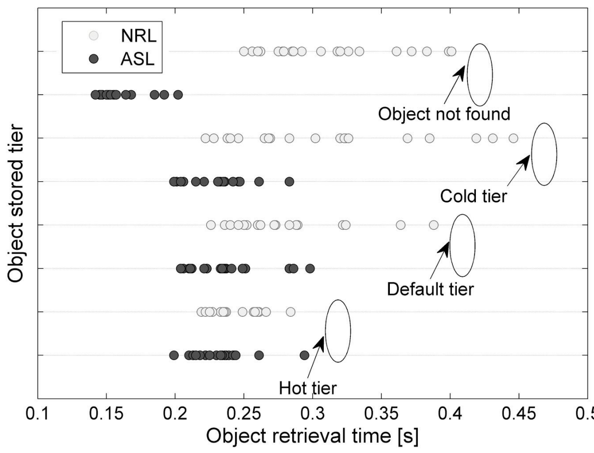


Fig. 7 - Distribution object lookup time tests for *NRL* and *ASL*.

As we clearly can see from the plots, the *ASL* always outperforms the *NRL* solution. The introduced gain ranges between a minimum of 5% when the requested object is on the hot tier, to a maximum of 51% when the object is not present. A further interesting observation can be made when looking at the distribution of the performed test results as reported in Fig. 7. In particular, we can observe that the results are more distributed for the *NRL* solution. We see a connection for this behavior to the number of REST calls made for the different tested cases, as the cases where four REST calls are needed are more sparse (see for instance the *NRL* tests for the object not found and cold tier cases).

The second analysis we present has the focus on three different scenarios where 20 containers are populated with a given number of objects. The tests consists in retrieving the headers for all the objects' using REST calls. The collected data is then processed to build a tree-like structure of containers and objects to give a human a readable overview of the storage. The three test scenarios we considered are:

- Retrieving the information from 20 containers populated with 0 objects, labeled with 20/0 in the plot;
- Retrieving the information from 20 containers, each having 1 object on a random tier, labeled with 20/20 in the plot;
- Retrieving the information from 20 containers, each having 1 object on each of the three tiers, labeled with 20/60 in the plot.

The results are reported in Fig. 8. As we can observe, the *ASL* solution is in all cases outperforming the *NRL,* which is what we expected. This is particularly evident in scenarios 20/0 and 20/20 where the introduced gain reaches respectively a 84% and a 60% value. Noteworthy, these two scenarios can be completed with the *ASL* algorithm using one and twenty one REST calls, respectively. This shows how the final time is directly proportional to the number of REST calls to perform. Moreover, as expected pretty close results are observed for the *NRL* and *ASL* algorithms in the 20/60 scenario as in both cases sixty-one REST calls are required to finish.

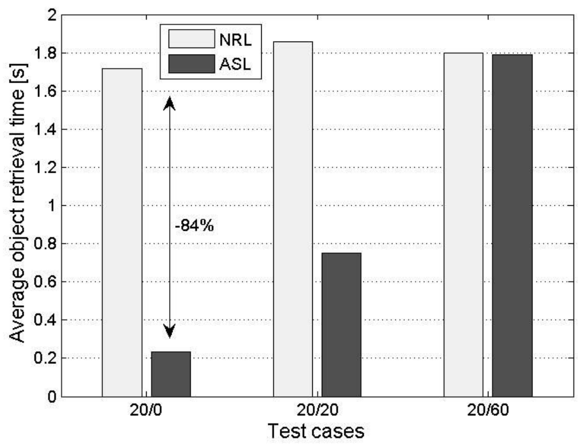


Fig. 8 - Average object lookup time for *NRL* and *ASL* in different test cases.

VII. Conclusions and Future Work

In this paper we introduced HOS as a storage system based on OpenStack Swift which implements additional features for object lookup and multi-tiering object storage. The implemented solution simplifies the usage of the storage policies implemented as different rings in Swift and additional functionalities have been designed following the microservice design principles. The proposed solution has been implemented in a prototype used for basic analysis and performance evaluation. The obtained results validated the proposed solution in a sample system setting and in real scenarios where multi-tiering object storage is adopted.

In our future work we plan to further investigate the proposed policies to further enhance scalability and efficiency in the multi-tiering solution. The automated storage-tiering algorithms could be, for instance, extended using machine-based learning technologies to automatically train HOS to move objects to the best-fitting storage tier. Adopting NoSQL distributed databases as for instance MongoDB would be a direction of investigation to follow to better support distributed cloud storage. A further research direction is the implementation of on-storage computation, to enable HOS to run operations directly on the stored objects instead of downloading the data for later computation. This could significantly reduce the time for the computation of large archived objects.

References

[1] R. Nachiappan, B. Javadi, R.N. Calheiros, and K. M. Matawie, *Cloud storage reliability for Big Data applications: A state of the art survey.* Journal of Network and Computer Applications, 97, 35-47, 2017.

[2] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, *Finding a Needle in Haystack: Facebook's Photo Storage*. In OSDI, vol. 10, pp. 1-8, 2010.

[3] B. Baesens, *Analytics in a big data world: The essential guide to data science and its applications*. John Wiley & Sons, 2014.

[4] J. Gantz, and D. Reinsel, *The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east.* IDC iView: IDC Analyze the future, 1-16, 2012.

[5] OpenStack Swift. URL: http://docs.openstack.org/developer/swift/

[6] R. Borovica-Gajić, R. Appuswamy, and A. Ailamaki, *Cheap data analytics using cold storage devices.* Proceedings of the VLDB Endowment, 9(12), 1029-1040, 2016.

[7] P. Nicolas *The History Boys: Object storage… from the beginning*. URL: http://www.theregister.co.uk/2016/07/15/the_history_boys_cas_and_object_storage_map/

[8] Ceph. URL: https://ceph.com/

[9] Oracle Solaris ZFS–What Is ZFS. URL: http://docs.oracle.com/cd/E23823_01/html/819-5461/zfsover-2.html

[10] OpenStack Keystone. URL: http://docs.openstack.org/developer/keystone/

[11] Python Documentation – Unittest. URL: https://docs.python.org/2.7/library/unittest.html

[12] M. Rys, "*scalable SQL*" Communications of the ACM 54.6: 48-53, 2011.