Towards a Community Cloud Storage

Ying Liu
KTH Royal Institute of Technology
Stockholm, Sweden
yinliu@kth.se

Vladimir Vlassov
KTH Royal Institute of Technology
Stockholm, Sweden
vladv@kth.se

Leandro Navarro
Universitat Politècnica de Catalunya
Barcelona, Spain
leandro@ac.upc.edu

***Abstract*—Community Clouds, usually built upon community networks, operate in a more disperse environment compared to a data center Cloud, with lower capacity and less reliable servers separated by a more heterogeneous and less predictable network interconnection. These differences raise challenges when deploying Cloud applications in a community Cloud.**

OpenStack Swift is an open source distributed storage system, which provides stand alone highly available and scalable storage from OpenStack Cloud computing components. Swift is initially designed as a backend storage system operating in a data center Cloud environment.

In this work, we illustrate the performance and sensitivity of OpenStack Swift in a typical community Cloud setup. The evaluation of Swift is conducted in a simulated environment, using the most essential environment parameters that distinguish a community Cloud environment from a data center Cloud environment.

***Keywords*-OpenStack Swift, Community Cloud, Evaluation**

I. Introduction

Many of Web2.0 applications, such as Wikis and social networks as well as Cloud-based applications may benefit from scalable and highly available storage systems. There are many distributed storage systems emerged to meet this goal, such as Google Spanner [1], Amazon Simple Storage Service [2], Facebook Cassandra [3],Yahoo! PNUTS [4], and OpenStack Swift [5]. However, all of these state-of-the-art systems are designed to operate in a data center environment, where under a normal workload, there are no hardware bottlenecks in the server capabilities as well as the network. We investigate the possibility to adapt these storage systems in an open area network environment. A realistic use case is the community Cloud [6], [7].

We consider a community Cloud built upon a community network. Specifically, community networks are large-scale, self-organized, distributed and decentralized systems constructed with a large number of heterogeneous nodes, links, content and services. Some resources, including nodes, links, content, and services, participating in a community network are dynamic and diverse as they are built in a decentralized manner, mixing wireless and wired links with diverse routing schemes with a diverse range of services and applications [8]. Examples of such community networks are Guifi.net in Spain [9] and AWMN in Greece [10]. A Cloud infrastructure in a community network can be used primarily to isolate and provide services, such as a storage service.

In order to provide Infrastructure as a Service Cloud in a community network and enable Cloud-based services and applications, a proper backend storage system is needed for various purposes, such as maintaining user information, storing Virtual Machine (VM) images as well as handling intermediate experiment results possibly processed with big data platforms [11], [12]. The first approach towards such a storage system is to evaluate the existing open source storage systems that may match our scenario. One of the widely-used open source Cloud software is OpenStack, which includes the storage system called Swift [5]. Studies have been conducted regarding the performance of Swift in a general Cloud environment, such as in the Amazon EC2, and in some private clouds [13], [14]. In this paper, we have provided the methodology and conducted a thorough evaluation of OpenStack Swift using a simulated environment regarding its properties to operate in a community Cloud environment. The evaluating results may also be extended to be used in multi-site Clouds and multi-Cloud federations with undedicated networks using best effort Internet, which has no guarantees in network latency and bandwidth.

Since Swift is originally designed to operate in a data center, we start our investigation by the identification of the differences between a data center environment and a community Cloud environment. In general, a data center is constructed with a large number of powerful dedicated servers connected with the exclusive, high bandwidth, and low latency network switches with regular maintenance. In contrast, a community Cloud upon a community network is established by connecting heterogeneous computing and networking resources in a wide geographic area. The three main differences between these two environments are:

- The types of computing resources: powerful dedicated servers vs. heterogeneous computing components;
- The network features: exclusive high speed networks vs. shared ISP broadband and wireless connections;
- The maintenance: regular vs. self-organized.

Based on the identification above, we have conducted the evaluations on a Swift cluster in the later sections.

The contributions of this paper are:

- We demonstrate a methodology to evaluate the performance of a distributed storage system in a simulated community Cloud environment;

- We quantify Swift performance under different hardware specifications and network features;
- We design and evaluate a self-healing algorithm on a Swift cluster to handle server failures;
- We investigate the feasibility to apply OpenStack Swift in a community Cloud environment.

The rest of this paper is organized as follows. Section II provides the background on the community Cloud and OpenStack Swift. Section III presents the experimental environment that we used for the Swift evaluation. Section IV describes our evaluation plan. Section V and Section VI present results of the evaluation of the Swift performance under different hardware and network configurations. Section VII presents an implementation of a self-healing mechanism for Swift. We conclude in Section VIII.

II. Background

A. Community Network and Community Cloud

Community networking, also known as bottom-up networking, is an emerging model for the Future Internet across Europe and beyond, where communities of citizens can build, operate and own open IP-based networks, a key infrastructure for individual and collective digital participation [15], [16]. Many services, including Internet access, cable TV, radio and telephone, can be provided upon the infrastructure of the community networks. An example is given in [17]. Furthermore, community networks can also be employed and extended to provide Cloud-based services, namely the *community Cloud*, to the community, researchers, and participating SMEs. Community Cloud provides alternative choices for its participants in the community to manipulate their own Cloud computing environment without worrying about the restrictions from a public Cloud provider.

However, building a community Cloud is very challenging. One of the major challenges is to achieve the self-management, high performance, and low cost services based on the overlay of the community networks, which is usually organized as the aggregation of a large number of widespread low-cost unreliable networking, storage and home computing resources. In this work, we provide an evaluation of an existing open source software, OpenStack Swift, in a simulated community Cloud environment.

B. OpenStack Swift

OpenStack Swift is one of the storage services of the OpenStack Cloud platform [18]. It consists of several different components, providing functionalities such as highly available and scalable storage, lookup service, and failure recovery. Specifically, the highly available storage service is achieved by data replication in multiple *storage servers*. Its scalability is provided with the aggregated storage from many storage servers.

The lookup service is performed through a Swift component called *the proxy server*. The proxy servers are the only access entries for the storage service. The main responsibility of a proxy server is to process the mapping of the names of the requested files to their locations in the storage servers. This namespace mapping is provided in a static file called *the Ring file*. Thus, the proxy server itself is stateless, which ensures the scalability of the entry points in Swift. The Ring file is distributed and stored on all the storage and proxy servers. When a client accesses a Swift cluster, the proxy checks the Ring file, loaded in its memory, and forwards client requests to the responsible storage servers.

The high availability and failure recovery are achieved by processes called *the replicators*, which run on every storage server. Each replicator uses the Linux rsync utility to push the data from the local storage server to the other storage servers, which should maintain the same replicated data, based on the information provided in the Ring file. By doing so, the under-replicated data are recovered.

C. Related Work

One of the leading companies for cloud backups called Zmanda has measured the performance of Swift on Amazon EC2 [13]. They have evaluated Swift performance with different proxy server and storage server capabilities by employing different flavors of Amazon instances. The result provides premium deployment strategy to satisfy the throughput requirements at a possible low cost. However, the evaluations only focus on the hardware setup of a Swift cluster. In addition, our work provides the evaluation of Swift's performance under different network environments, which is an important aspect in a community Cloud.

FutureGrid, the high performance experimental testbed for computer scientists, has evaluated several existing distributed storage systems towards a potential usage as VM repositories [19]. The evaluations are conducted by the comparison of MongoDB [20], Cumulus [21], and OpenStack Swift. They have concluded that all these three systems are not perfect to be used as an image repository in the community Cloud. However, Swift and Cumulus have the potential to be improved to match the requirements. In comparison, instead of comparing the performance of several open source Cloud storage systems, we analyze Swift performance under identified metrics described in Section IV.

Another scientific group has investigated the performance of Swift for CERN-specific data analysis [14]. The study investigated the performance of Swift with the normal use case of the ROOT software [22]. However, their work mainly focuses on the evaluation of Swift under the CERN-specific data usage pattern. Our work evaluates Swift performance under a general usage pattern but with different resource constrains, namely, server capacities and network features.

III. Experiment Environment

A. The Platform

In order to simulate a community Cloud environment, we start by constructing a private Cloud and tune some of its parameters according to the aspects that we want to evaluate. These parameters are explained at the beginning of every evaluation section. For the private Cloud environment, we have configured the OpenStack Compute (Nova) [23] and the Dashboard (Horizon) on a large number of interconnected high-end server computers. On top of the OpenStack Cloud platform, we have configured a Swift cluster with different VM flavors for our experiments. Specifically, the following four different VM flavors, which decide the capabilities of the servers in terms of CPU and memory, are constructed.

- XLarge Instance: 8 Virtual Core, 16384 MB Memory;
- Large Instance: 4 Virtual Core, 8192 MB Memory;
- Medium Instance: 2 Virtual Core, 4096 MB Memory;
- Small Instance: 1 Virtual Core, 2048 MB Memory.

Each virtual core is 2.8 GHz. The different capabilities of VMs are used in the experiments described in Section V, which identify the bottom-line hardware requirements for Swift to operate according to a specific level of performance. These VMs are connected with 1Gbps sub-networks. Hard drives with 5400 rpm are mounted for the storage servers. In our view, this setup represents a reasonable environment in a community Cloud infrastructure.

B. The Swift Setup

Our Swift cluster is deployed with a ratio of 1 proxy server to 8 storage servers. According to OpenStack Swift Documentation [24], this proxy and storage server ratio achieves efficient usage of the proxy and storage CPU and memory under the speed bounds of the network and disk. Under the assumption of uniform workload, the storage servers are equally loaded. This implies that the Swift cluster can scale linearly by adding more proxy servers and storage servers following the composition ratio of 1 to 8. Due to the linear scalability, our experiments are conducted with 1 proxy server and 8 storage servers.

C. The Workload

We have modified the Yahoo! Cloud Service Benchmark [25] (YCSB) to generate the workloads for a Swift cluster. Our modification allows YCSB to support read, write, and delete operations to a Swift cluster with best effort or a steady workload according to a requested throughput. If the requested throughput is so high that requests cannot be admitted by the system, then the requests are queued for later execution, thus achieving the average target system throughput in the long run. Furthermore, YCSB is given 16 concurrent client threads and generates uniformly random read and write operations to the Swift cluster.

The Swift cluster is populated using randomly generated files with predefined sizes. Our experiment parameters are chosen based on parameters of one of the largest production Swift clusters configured by Wikipedia [26] to store images, texts, and links. The object size is 100KB as a generalization of the Wikipedia scenario.

D. The Network Instrumentation

We apply "tc tools" by NetEm [27] to simulate different network scenarios, and to be able to manage the network latency and bandwidth limitations for every thread or service.

IV. Evaluation Plan

Swift is designed to achieve linear scalability. However, potential performance bounds can be introduced by the lack of computing resources of the proxy servers, disk read/write speed of the storage servers, and the network features of the Swift sub-networks, which connect the proxy servers and the storage servers. By understanding these potential performance bounds of Swift, as well as the major differences between a community Cloud environment and a data center Cloud environment identified in Section I, we have set up the following three sets of experiments, which in our view, cover the major concerns to achieve a community storage Cloud using Swift.

1) Hardware requirement by Swift proxy servers: In this experiment, we focus on the identification of the minimum CPU and memory requirement of a Swift proxy server to avoid bottlenecks and achieve efficient resource usage, given a network and disk setup described in Section III-A.

2) Swift Performance under different networks: In this set of experiments, we correlate Swift performance with different network features given that the proxy servers and the storage servers are not the bottleneck. The evaluations quantify the influences introduced by network latencies and insufficient bandwidth to the performance of a Swift cluster. The results can help a system administrator to predict and guarantee the quality of service of a Swift cluster under concrete network setups.

3) Swift's Self-healing property: According to our experience in a community Cloud, the servers are more prone to fail comparing to the servers in a data center. Thus, the applications deployed in a community Cloud are expected to have self-* properties to survive with server failures. In this experiment, we have examined the failure recovery mechanism in Swift and extended it with a self-healing control system. In our view, this self-healing mechanism is essential when Swift is deployed in a community Cloud.

V. Server Hardware Evaluation

It is intuitively clear and stated in Swift documentation [24] that the proxy servers are compute intensive and the storage servers are disk intensive. The following experiments are focused on the minimum hardware requirements of the proxy servers. The storage servers are configured with our Small VM flavors listed in Section III-A, which are ensured

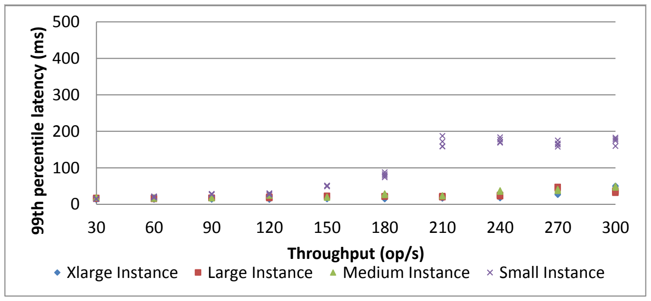

Figure 1. Read latency of Swift using proxies with different VM flavors

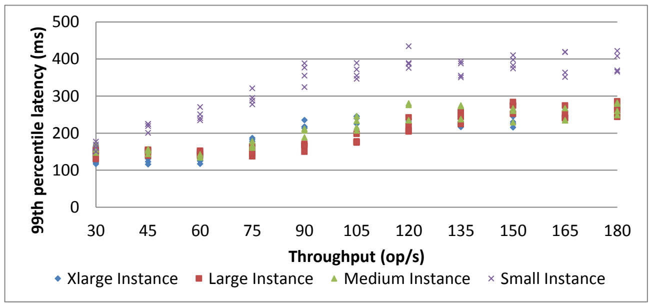

Figure 2. Write latency of Swift using proxies with different VM flavors

not to be the bottleneck. The proxy servers are deployed with different flavors of VM instances.

Figure 1 and Figure 2 illustrate the 99th percentile read and write latencies (y-axis) under a specified steady workload (x-axis) generated from YCSB with the deployment of the proxy servers using different VM flavors. The figure is plotted using the results from five repeated runs in each setup.

Obviously, there are no significant performance differences when using Medium, Large, or even Xlarge VM flavors as Swift proxy servers, for both reads and writes. In contrast, when running the proxy server on a Small VM instance, the latencies of the write requests increase sharply after 30 operations per second (abbreviate as op/s). This phenomenon is also observed in the read experiments after the throughput of 120 op/s.

Discussion: In order to achieve efficient resource usage as well as acceptable performance without potential bottlenecks in Swift, our experiment demonstrates the correlation of the performance of a Swift cluster with the capabilities of the proxy servers with read or write workloads. In particular, the Small instance proxy servers, in our experiment, experience severe performance degradation after some throughput threshold values shown in Figure 1 and Figure 2 for read and write requests. We observe that the CPU of the Small instance proxy servers saturates after the threshold value, which results in the increase of request latency. The balance of request latency (system throughput) and the saturation of the CPU in the small instance proxy servers is reached at a later steady state shown in both Figure 1 and Figure 2.

It is important to know both the threshold value and the steady state value for every different deployment of the Swift proxy servers. This experiment demonstrates the methodology for identifying the potential bottlenecks in the capabilities of the proxy servers in a specific Swift setup. The threshold value defines the state where the latency of service starts to be affected significantly if the workload keeps increasing. The steady state value can be used to calculate the maximum reachable throughput of the system, given the number of concurrent clients. In between the threshold value and the steady state, a linear correlation of the request latency and the system workload with different coefficients for reads and writes is observed.

VI. Network Evaluation

In this section, we focus on the evaluation of a Swift cluster under different network features. Specifically, we examine the effect of the network latency and the bandwidth limit to Swift's performance. We employ a Swift setup with the Medium proxy servers and the Small storage servers, which is demonstrated to have no bottleneck in our operational region in the previous section.

As introduced in Section III-D, we use NetEm "tc tools" to simulate different network features in a Swift cluster. We have run two sets of experiments to evaluate the Swift under two important network factors, namely the network latency and the available bandwidth. In each set of the experiments, our results provide the intuitive understanding of the performance impact of these network factors on Swift. Furthermore, these performance impacts are compared with the results from the experiments conducted separately on storage servers and proxy servers. The storage server and proxy server comparison provides more fine-grained guidelines for the possible deployment of a Swift cluster in a community Cloud to achieve efficient network resource usage and desired performance.

A. *Swift Experiment with Network Latencies*

In this section, we present the evaluation results of Swift's performance under specific network latencies. The network latencies are introduced on the network interfaces of the proxy servers and the storage servers separately. In particular, for the read experiments, we have introduced the latencies on the outgoing links while, for the write experiments, latencies are introduced on the incoming links. Latencies are introduced in an uniform random fashion in a short window with the average values from 10 ms to 400 ms. After 400 ms latency, the Swift cluster might become unavailable because of request timeouts.

Figure 3 and Figure 4 demonstrate the influence of network latencies to the read and write performance of a Swift cluster. In both figures, the x-axis presents the average latencies introduced to either the proxy servers or the storage servers network interfaces. The y-axis shows the corresponding performance, quantified as system throughput

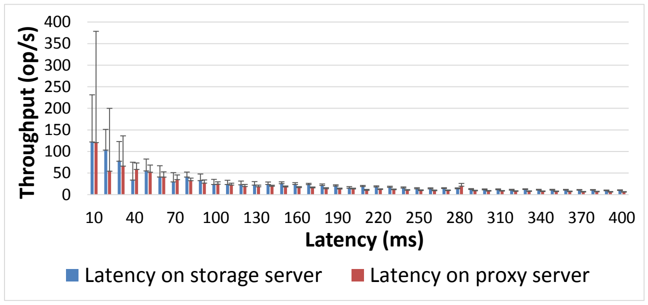

Figure 3. Read Performance of Swift under Network Latencies

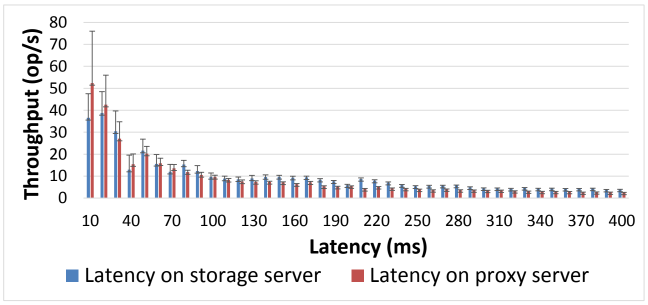

Figure 4. Write Performance of Swift under Network Latencies

in op/s. Experiments on different latency configurations last for 10 minutes. Data are collected every 10 seconds from the system performance feedback in YCSB. The plot shows the mean (the bar) and standard deviation (the error line) of the results from each latency configuration.

It is obvious that Figure 3 and Figure 4 share similar patterns, which indicates that the network latencies on either the storage servers or the proxy servers bound the read and write performance of the cluster. Furthermore, it is shown in both figures that the network latencies on the proxy servers result in further performance degradation. The throughput of Swift becomes more stable (shown as the standard deviation), although decreasing, with the increasing of network latencies. The decreasing of throughput causes less network congestions in the system and results in more stable performance.

Discussion: From the experiment data, we could estimate the bounded performance of a Swift cluster by deriving a mathematical correlation model of the system throughput and the network latencies. For example, a logarithmic approximation may best fit the performance boundaries shown in both figures. The correlation model can be used to predict system throughput under a given network latency and further be used to make guarantees for the quality of service. In particular, by knowing the network latencies on every incoming and outgoing links of a server in a community Cloud, we could estimate the bounded performance (throughput) for a particular requested data unit.

In order to estimate the bounded performance for a requested data unit, we need to understand the consistency

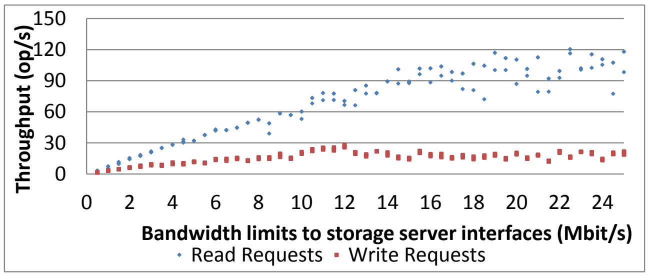

Figure 5. Performance of Swift with Bandwidth Limits on Storage Servers

mechanism implemented in Swift. Every read and write request is sent to several storage servers, depending on the configured replication degree, from the proxy servers. Not all the responses from the storage servers are needed to successfully complete a client request. Specifically, Swift is an eventual consistent system that, for a read request, only one correct response from the storage servers is needed. In contrast, the write operations require stricter scenario where the majority of the storage servers, which store the same replicated data, are needed. By understanding this consistency mechanism in Swift, the read performance is bounded by the fastest (lowest latency) outgoing links of the storage servers that store the requested data. In comparison, the write performance is bounded by slowest incoming links of the storage servers in the majority quorum, which is formed by the faster majority from all the storage servers that should store the target data. By knowing the network latencies of all the links in a community Cloud and the mappings of the namespace, we could use the results in Figure 3 and Figure 4 to estimate a bounded performance for a read or write request on a single data unit.

B. *Swift Experiment with Network Bandwidth*

In this section, we present the evaluation results regarding the influence of the available network bandwidth to the performance of a Swift cluster. First, we illustrate the read and write performance bounds that correspond to the available bandwidth in the storage servers. Then, we present the monitored results of the network bandwidth consumption in the proxy server and come up with deployment suggestions.

1) Storage Server Bandwidth Consumption: In this experiment, the available network bandwidth allocated to each storage server is shown in the x-axis in Figure 5. The y-axis represents the system throughput that corresponds to the limited network bandwidth allocated to the storage servers. The plot shows system throughput under different bandwidth allocations averaged in ten minutes. Data are collected through three individual repeated experiments.

Discussion: As shown in Figure 5, there is a clear linear correlation of the bandwidth consumption and the system throughput for both read and write. Different gains can be calculated for the read and write workloads before reaching the saturation point and entering the plateau. Specifically, with the gradual increase of the available bandwidth allocated to the read and write workloads, the write workloads reach the saturation point earlier than the read workloads. There are mainly two reasons for this difference. One reason is that usually the disk write speed is slower than the read speed. Another reason is that a write request in a Swift cluster need the majority of the responsible storage servers to perform while a read request only require the fastest respond from one of the responsible storage servers.

In order to achieve the efficient usage of the network bandwidth in community Cloud platform, a system maintainer needs to find out the read and write saturation balance among the available bandwidth and the above mentioned two potential bottlenecks. For example, using our hard disk and the Swift setup described in Section III-B, the saturation point for the write requests corresponds to 11 Mbit/s bandwidth allocation shown in the x-axis in Figure 5. Understanding the saturation balance among multiple resources, a system maintainer can deploy the storage nodes of a Swift cluster based on the available outbound and inbound bandwidth that fit the request patterns (read-intensive or write-intensive).

2) Proxy Server Bandwidth Consumption: When evaluating the correlations between the available bandwidth and the Swift performance, we have monitored the bandwidth usage by all the storage and proxy servers. The bandwidth consumption patterns in the proxy servers are of interest.

Figure 6 presents the read and write workloads generated from our customized YCSB. Data are collected every 10 seconds. Since the proxy servers are the only entry points for a Swift cluster, it is intuitively clear that the workload in Figure 6 is equal to the inbound network traffic imposed on the proxy servers when using the setup of only one workload generator and one proxy server. Figure 7 shows the outbound bandwidth consumption on the proxy server for the read and the write requests. The bandwidth consumed by the reads and the writes shows correlations. Specifically, the outbound bandwidth consumed by write requests is X times more than the bandwidth consumed by the read requests, where X corresponds to the replication degree. In our setup, the replication degree is configured to three, thus writes consume three times more outbound bandwidth on the proxy server than the reads.

Discussion: The extra bandwidth consumed by write requests shown in Figure 7, is because of the replication technique employed by Swift. Specifically, the replicated data are propagated from the proxy server in a flat fashion. The proxy servers send the X times replicated data to the X responsible storage servers directly, when the data is written to the cluster. Based on this knowledge, a system administrator can make decisions on the placement of proxy servers and differentiate the read and write workloads considering network bandwidth consumption. Based on the above analysis, it is desired to allocate X times (the replication degree) more outbound network bandwidth to the proxy servers under write-intensive workload because of the data propagation schema employed by Swift.

VII. The Self-healing of Swift

Since a community Cloud is built upon a community network, which is less stable than a data center environment, systems operating in a community Cloud environment have to tackle with more frequent node leaves and network failures. Thus, we have developed a self-healing mechanism in Swift to operate in a community Cloud environment.

Originally, the failure recovery in a Swift cluster is handled by a set of replicator processes introduced in Section II-B. Swift itself cannot automatically recover from server failures because of the Ring file, which records the namespace mapping. Specifically, the replicator processes on all the storage servers periodically check and push their local data to the locations recorded in the Ring files, where the same replicated data should reside. Thus, when the storage servers fail, the Ring files should respond to such changes swiftly in order to facilitate the replicators by providing the location of the substitute replication servers, and thus guarantee the availability of the data. However, this healing process is expected to be monitored and handled manually by a system maintainer in the Swift design. In the following paragraphs, we describe an algorithm to automate the healing process of Swift. The self-healing algorithm is validated under different failure rates introduced by our failure simulator.

A. The Self-healing Algorithm

In this section, we describe a MAPE (Monitor, Analysis, Plan, and Execute) control loop for the self-healing algorithm in Swift for the Ring files shown in Figure 8.

The control cycle illustrated in Figure 8 is implemented on a control server, which can access the local network of the Swift cluster. In the monitor phase, it periodically sends heartbeat messages to all the storage servers registered in the Swift cluster. We assume that the storage servers follow the fail-stop model. If there are no replies from some storage servers, these servers are added to the Failure Suspicious List (FSL) maintained in the control server. The FSL records the number of rounds that a server is suspected failed. Then, in the analysis phase, the control server compares the current round FSL with the previous FSL. Servers in the previous FSL are removed if they do not exist in the current FSL. Servers in both previous FSL and current FSL will be merged by summing up the number of suspicious rounds in previous FSL and current FSL. A configurable threshold value is introduced to manipulate the confidence level of the failure detect mechanism. Next, in the plan phase, if the number of the suspicious rounds associated with the storage servers in the FSL exceeds the threshold value, the control system marks these servers as failed and plans to remove them from the Ring files. If server removals are needed, the control system checks the available servers from the

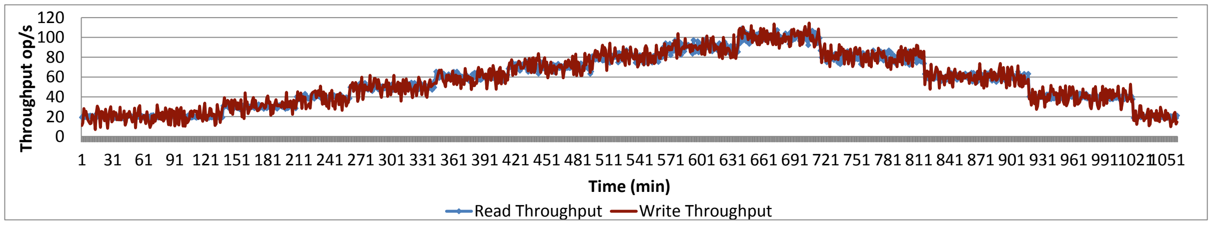

Figure 6. Workload Patterns

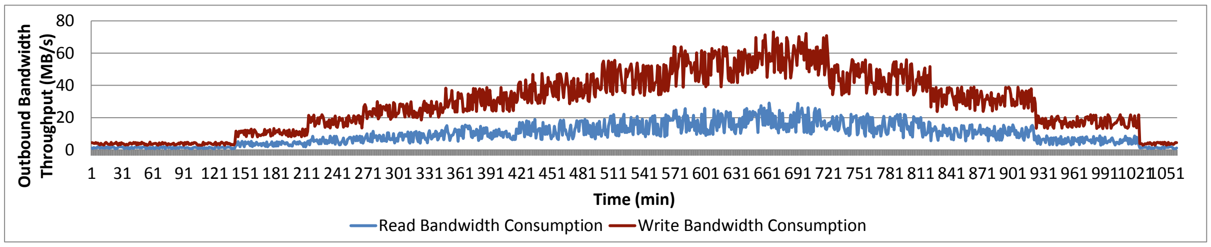

Figure 7. Proxy Server Outbound Bandwidth Usage for Reads and Writes

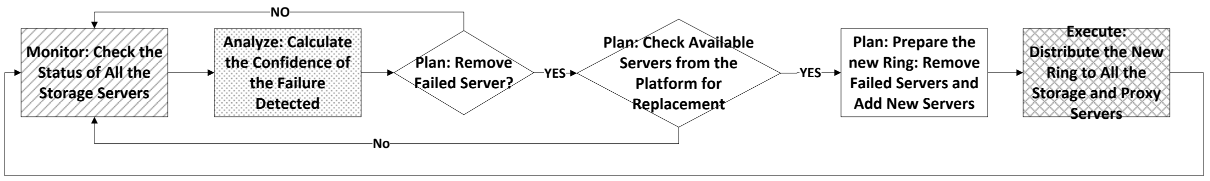
Monitor: Check the Status of All the Storage Servers
Analyze: Calculate the Confidence of the Failure Detected
Plan: Remove Failed Server?
NO
YES
Plan: Check Available Servers from the Platform for Replacement
No
YES
Plan: Prepare the new Ring: Remove Failed Servers and Add New Servers
Execute: Distribute the New Ring to All the Storage and Proxy Servers

Figure 8. Control Flow of the Self-healing Algorithm

platform and plans to join them as the substitute servers for the failed ones. If there are available servers in the platform, new Ring files are prepared by removing the failed servers and adding the new ones. Finally in the execution phase, the new Ring files, which record the updated namespace and server participation information, are distributed to all the current storage servers and the proxy servers.

After the self-healing cycle completes, the data in the failed servers will be recovered from the other replicas to the newly added servers by the Swift replicator daemons running in the background. It is intuitively clear that the frequency of server failures, which cause the data loss, should not exceed the recovery speed by the replicators otherwise some data may be lost permanently.

In Figure 9, we illustrate the self-healing process with a Swift setup presented in Section III-B. The horizontal axis shows the experiment timeline. The blue points along with the vertical axis present the cluster's health by showing the data integrity information obtained by a random sampling process, where the sample size is 1% of the whole namespace. The total number of files stored in our Swift cluster is around 5000. In order to simulate storage server failures, we randomly shut down a number of the storage servers. The decrease of the data integrity observed in Figure 9 is caused by shutting down 1 to 4 storage servers. The red

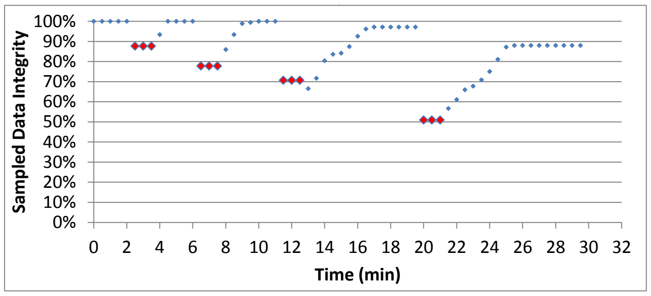

Figure 9. Self-healing Validation under the Control System

points show the control latency introduced by the threshold value, which is two in our case, set for the failure detector of the control system to confirm a server failure. When we fail more than 3 (replication degree) servers, which may contain all the replicas of some data, the final state of the self-healing cannot reach 100% data integrity because of data loss.

The choice of the threshold value for our failure detector provides the flexibility to make trade-offs. Specifically, a larger threshold value may delay the detection of server failures but with higher confidence. On the other hand, a smaller threshold value makes the control system react to server failures faster. However, the commitments of the server failures come with a potential cost. This cost is the rebalance of data on the substitute servers. Thus, with higher failure detection confidence, the data rebalance cost is minimized, but it may slow down the system reaction time. Furthermore, the cost of data rebalance is proportional to the data stored in the failed server. Thus, it is a good strategy to set a larger threshold value when there is a large amount of data stored in the storage servers.

VIII. Conclusions

This paper presents the first evaluation results in a simulated environment on the feasibility of applying OpenStack Swift in a community Cloud environment. It presents a detailed technical evaluation of Swift, regarding its bottom-line hardware requirements, its resistance to network latencies and insufficient bandwidth. Furthermore, in order to tackle with frequent server failures in the community Cloud, a self-healing control system is implemented and validated. Our evaluation results have established the relationship between the performance of a Swift cluster and the major environment factors in a community Cloud, including the proxy hardware and the network features among the servers. Whether it is feasible to deploy a Swift cluster in the community Cloud environment depends on the further research on these environment factor statistics. There also exist many unknown community Cloud environment factors to be discovered in the real deployments and evaluations. Thus, the evaluation of OpenStack Swift as a real community Cloud deployment is our future work.

Acknowledgment

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the FP7 project CLOMMUNITY funded by the European Commission under EU FP7 Grant Agreement number 317879. The authors gratefully acknowledge Vamis Xhagjika for his collaboration in development and configuration of the experiment testbed.

References

[1] J. C. Corbett, J. Dean, and et al., "Spanner: Google's globally-distributed database," in *Proc. OSDI*, 2012.

[2] Amazon simple storage servie. [Online]. Available: http://aws.amazon.com/s3/

[3] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, 2010.

[4] B. F. Cooper, R. Ramakrishnan, and et al., "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, 2008.

[5] K. Pepple, *Deploying OpenStack.* O'Reilly Media, 2011.

[6] G. Briscoe and A. Marinos, "Digital ecosystems in the clouds: Towards community cloud computing," in *Proc. DEST*, 2009.

[7] A. Marinos and G. Briscoe, "Community cloud computing," *Proc. CoRR*, 2009.

[8] B. Braem, C. Blondia, and et al., "A case for research with and on community networks," *SIGCOMM Comput. Commun. Rev.*, 2013.

[9] Guifi.net. [Online]. Available: http://guifi.net/

[10] Awmn the athens wireless metro-politan network. [Online]. Available: http://www.awmn.net/

[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. OSDI*, 2004.

[12] Y. Wang, W. Jiang, and G. Agrawal, "SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats," in *Proc. CCGrid*, 2012.

[13] Zmanda: The leader in cloud backup and open source backup. [Online]. Available: http://www.zmanda.com/blogs/?cat=22

[14] S. Toor, R. Toebbicke, and et al., "Investigating an open source cloud storage infrastructure for cern-specific data analysis," in *Proc. NAS*, 2012.

[15] Community networks testbed for the future internet. [Online]. Available: http://confine-project.eu/

[16] A community networking cloud in a box. [Online]. Available: http://clommunity-project.eu/

[17] Bryggenet community network. [Online]. Available: http://bryggenet.dk/

[18] Openstack cloud software. [Online]. Available: http://www.openstack.org/

[19] J. Diaz, G. von Laszewski, and et al., "Futuregrid image repository: A generic catalog and storage system for heterogeneous virtual machine images," in *Proc. CloudCom*, 2011.

[20] C. Chodorow, "Introduction to mongodb," in *Free and Open Source Software Developers' European Meeting*, 2010.

[21] Nimbus project. [Online]. Available: http://www.nimbusproject.org/

[22] Physics data analysis software. [Online]. Available: http://root.cern.ch/drupal/

[23] K. Pepple, "Openstack nova architecture," *Viitattu*, 2011.

[24] Openstack swift's documentation. [Online]. Available: http://docs.openstack.org/developer/swift/

[25] B. F. Cooper, A. Silberstein, and et al., "Benchmarking cloud serving systems with ycsb," in *Proc. SoCC*, 2010.

[26] Scaling media storage at wikimedia with swift. [Online]. Available: http://blog.wikimedia.org/2012/02/09/scaling-media-storage-at-wikimedia-with-swift/

[27] S. Hemminger and et al., "Network emulation with netem," in *Linux Conf Au*, 2005.