American Journal of Engineering and Applied Sciences

Original Research Paper

A Methodology for Continuous Evaluation of Cloud Resiliency

[1]Xiaoyong Yuan, [2]Long Wang, [4]Tiancheng Liu and [3]Yue Zhang

[1]*School of Software and Microelectronics, Peking University, Beijing, China*
[2]*IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA*
[3]*Aliyun Computing Co., LTD, Alibaba Group, Beijing, China*
[4]*China Research Laboratory, IBM Corporation, Beijing, China*

Article history
Received: 23-10-2015
Revised: 14-01-2016
Accepted: 25-01-2016

Corresponding Author:
Long Wang
IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA
Email: wanglo@us.ibm.com

Abstract: With the growth of cloud computing, resiliency of cloud is critical for enterprises' business. However, the continuous-changing of cloud makes evaluation of cloud resiliency more difficult. In this study, we design a methodology for automatic and continuous evaluation of cloud resiliency and implement it in a tool called CRGauge. The Continuous Evaluation Model methodology leverages fault injection techniques to inject faults and an open-source library to set up synthetic workloads for the test campaign. Our experiment results on OpenStack cloud platform show that resiliency of OpenStack is needed to be improved especially in heavy workloads.

Keywords: Cloud Resiliency, OpenStack, Continuous Evaluation, Fault Injection

Introduction

Cloud environments play a critical role in delivering IT services to end users because cloud offers high resource utilization, fast convenient resource provisioning and deprovisioning, continuous management, maintenance and upgrade of machines transparent to end users and low overhead of the management. Resiliency of cloud environments is essential for the high availability of IT services and is a major concern of enterprises. For example, Instagram and Vine suffer from about 1 h downtime on Amazon's EC2 cloud last year (Whittaker, 2013) and a great number of outages on Amazon EC2 are reported in (Von Eicken, 2011). According to a survey in (NNT, 2009), 73% of CIOs and CFOs wouldn't put their financial and accounting applications in the cloud and 57% wouldn't place any business critical applications in the cloud.

Resiliency measurement is prerequisite for understanding and enhancement of cloud system resiliency. Traditionally fault injection is applied to measure system resiliency. It usually takes several weeks to set up fault injection experiments for a system. Moreover, a resiliency expert is required for the fault injection setup. As cloud systems undergo continuous changes and new code/new features are rapidly applied to cloud systems, this manual setup of fault injection is not fit for the dynamic nature of cloud systems.

This paper proposes a methodology that automatically measures cloud system resiliency to enable continuous evaluation of cloud resiliency. To our best knowledge, there is no prior work in the literature that tries to address the continuous evaluation of resiliency of constant-changing cloud systems. DS-Bench is a relevant testing environment for cloud resiliency. However, it hasn't capability of continuous evaluation for cloud system (Banzai *et al*., 2010; Fujita *et al*., 2012). Besides manual setup of fault injection campaigns, prior arts mainly focus on benchmarking and measurement of system performance and dependability, e.g., SPEC (SPEC), TPCB (TPCB), DS-Bench and CloudBench (Silva *et al*., 2013). These benchmarks or measures lack the ability of facing changes according to the benchmark metrics defined in (Vieira *et al*., 2012).

Our methodology, Continuous Evaluation Model for Cloud Resiliency (CEM), devises basic modules which are required for continuous measurement of Cloud system resiliency. Figure 1 shows the architecture of the CEM methodology.

The deployment module, fault injection module and resiliency computation module are essential parts for continuous evaluation of cloud resiliency. The deployment module leverages capabilities of the target cloud system to set up workloads, deploy the fault injection engine for certain types and scenarios of faults/failures, prepare the resiliency-computation models and do other configurations.

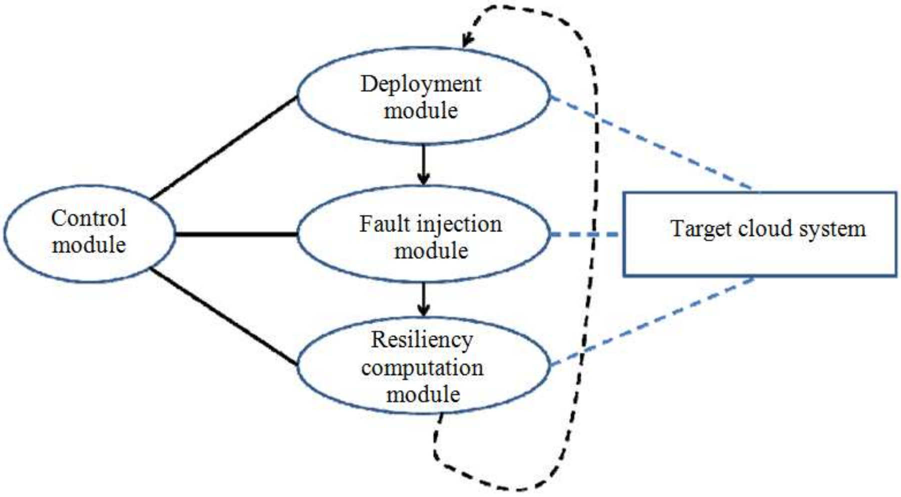


Fig. 1. Overview of the CEM Methodology

The fault injection module conducts fault injection experiments for all the scenarios specified by the deployment module. The outputs and results of the fault injection module are fed into the resiliency computation module, which consists of a number of models, to compute the cloud system resiliency for different scenarios. Then the computed resiliency data are presented to the users of the cloud and workload. This whole cycle of deployment, fault injection and resiliency computation phases repeats again and again for the continuous evaluation of cloud system resiliency. The control module supervises the three modules during the repeated cycles.

We implemented the CEM methodology in a tool called CRGauge. This tool is cloud-system independent and can be applied to different cloud systems when cloud-specific adaptors are available. Moreover, CRGauge treats the target cloud system as a black box, i.e., it does not interact with operational details and is evaluated only by cloud operations' outputs. For assessment of our CEM methodology and CRGauge tool, we then applied the CRGauge to the OpenStack (OpenStack) cloud platform and performed continuous evaluation of the OpenStack resiliency. OpenStack is a popular open-source cloud management system. OpenStack evolves very fast and a large number of experienced developers contribute to the OpenStack codebase and features. So it is critical to be able to evaluate the resiliency of OpenStack system in an automated and continuous fashion.

In summary, this paper has the following contributions:

- *Proposal of the CEM methodology for addressing the problem of automated continuous evaluation of Cloud system resiliency.* This problem is very important to Cloud systems which, unlike traditional computing systems, undergo rapid and continuous changes all the time
- *Design and development of the CRGauge tool that is a practical implementation of the CEM methodology.* The CRGauge tool is designed to be independent of cloud systems and treats target cloud systems as black box
- *Demonstration of the effectiveness and practicability of the CRGauge tool in evaluating the OpenStack cloud system's resiliency.* The experiement results show that resiliency of OpenStack cloud system becomes worse when facing heavy workloads

Rest of the paper is organized as follows. Section II presents an overview of the CEM methodology. The CRGauge tool implements the methodology. Discussions of individual components in CRGauge are given in SECTION III. Demonstration of the methodology and the tool on evaluation of OpenStack is presented in section IV. Finally, section V concludes the paper. Due to the page limitation, related work is discussed throughout the paper rather than in a separate section.

Continuous Evaluation of Cloud Resiliency

Continuous evaluation of resiliency is critical for cloud platforms and cloud services, because cloud platform is a dynamical environment with constant changes and new features or fixes are rapidly applied to the cloud environment.

Figure 1 illustrates the basic components of the Continuous Evaluation Model for Cloud Resiliency (CEM).

Deployment Module

Evaluation of cloud resiliency is performed by injecting different types of errors in various scenarios. In order to enable the evaluation in a continuous fashion, it is essential to automatically deploy the test environment, which includes the target cloud systems or services, workloads and setup of fault injection experiments. If the target cloud systems, services or workloads are under design, development or test phase, the deployment can be performed on them directly. If they are in the production phase, the production systems, services, or workloads must be cloned into a test environment by leveraging cloud capabilities, before the fault injection software and resiliency test campaigns are set up in the test environment. Thanks to the virtualization capability and automated management widely supported by a cloud platform, the cloning is not a big challenge.

The deployment module is designed to drive the deployment automation. The module reads the deployment specification and invokes tools like Puppet (Puppet), Chef (Chef) or developed scripts to conduct the deployment. APIs of the target cloud platform are invoked for certain cloud operations during the deployment, e.g., cloning of VMs. We use an Apache project called Libcloud (Libcloud) as an adaptor between the deployment module and the cloud-specific APIs so that the deployment module is portable across different cloud platforms (OpenStack, CloudStack (CloudStack), etc.).

Besides deployment of the test system, the deployment module also sets up test workloads according to the deployment specification. The test workload can be the real workload fed to the test system (the workload is fed to the production system at the same time) or synthetic workload generated by certain tools, e.g. CloudBench (Silva *et al*., 2013), CloudStone (Sobel *et al*., 2008). Whether to use the real workload or synthetic workload, together with the characteristics of the synthetic workload if it is used, is recorded in the deployment specification.

Setup of the fault injection is another task of the deployment module and is also documented in the deployment specification. This involves installation of the fault injection software into the test system, configuration of the fault injection software for injecting different types of faults at different locations and occasions and setup of the fault injection life cycle (e.g., creation of the VM images for the prior-fault injection states so that fault injection experiments are applied to the same initial state).

Fault Injection Module

Fault injection experiments are launched after the test environment and the fault injection softwares are deployed. As there are a number of fault injection tools in the literature and industry, we can use an existing fault injection tool.

Resiliency Computation Module

After fault injection, the resiliency computation module collects fault injection results and other relevant data and computes the resiliency of cloud systems and individual cloud services based on the collected data. Basically speaking, resiliency is measured by availability which is computed as:

$$Availability = \frac{MeanTimeToFailure}{MeanTimeToFailure + MeanTimeToRecovery}$$

So resiliency computation takes into account not only the failure behavior obtained from fault injection, but also recovery behavior for different types of failures. So resiliency computation requires specified resiliency models. An example resiliency model, for a VM, a Cloud component, or a Cloud service, is given in Fig. 2 and is represented as a state transition diagram.

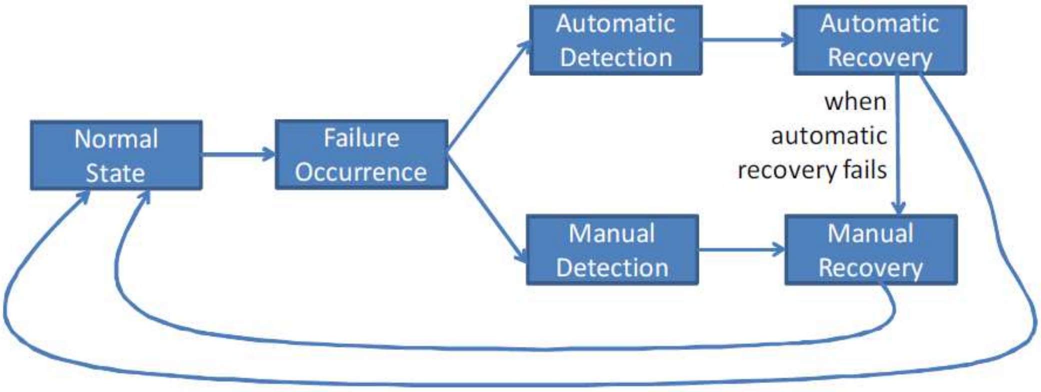


Fig. 2. An example resiliency model

In this example model, a failure occurrence can be detected either automatically or manually. When a failure is detected by the monitoring capability of the cloud platform, this automatic failure detection has low latency. When a failure is not detected by any monitoring capability and is finally detected by a user, e.g., from customers' complaints or reports, this manual detection has much longer latency. In different scenarios, the detection latency is different. For instance, administration team who are 24 h available can get notification of the failure from the user much sooner than that who only works 8 h a day. Usually an automatic recovery follows the automatic failure detection, as shown in Fig. 2. When the automatic recovery fails or the failure is detected manually, manual recovery will take over the recovery. The resiliency of the system is computed as the probability of the system being in the *Normal State*.

The parameters of the resiliency models for different failures should be available for the resiliency computation. Certain parameters, e.g., detection latency of automatic detection mechanisms and time spent in automatic recovery, can be measured by tools such as our implementation of the CEM methodology described below. Other parameters, e.g., probability of automatic recovery failures, average latency of manual detection and average manual recovery time, may be obtained from statistic data or user knowledge and expertise.

Control Module

For continuous evaluation of cloud resiliency, the whole evaluation cycle of "deployment-fault injection-resiliency computation" must be driven continuously and automatically. Control Module decides when to inject faults and when to stop tests and collect data according to the configuration set by the *Deployment Module*. The *Control Module* would clean up the cloud environment after a test cycle. The *Control Module* schedules the whole evaluation, including schedule of next evaluation and relaunching of the next evaluation process.

CRGauge: Cloud Resiliency Gauge

Based on the methodology for continuous evaluation of cloud resiliency above, we designed and developed a tool called CRGauge. Figure 3 illustrates the architecture of the CRGauge tool. CRGauge is a software package implemented in Python which is customizable, extendable and portable. The CRGauge tool is typically deployed in a machine outside of the target system.

Deployment Module

The deployment module consists of three components: *Loader*, *Deployer* and *Generator*, as shown in Fig. 3.

Loader reads in the specification file of the resiliency test campaign. The specification file is in XML format and contains fault type, injection target, injection time, workloads (request types, distribution parameters, arrival rate for each type of requests, etc.), timeout setting and other relevant information. Environment settings, which indicates cloud type, access to the target system and total time of the resiliency test campaign, are also included in the specification. The *Loader* parses the XML file and sends the parsed data to the *Deployer*.

A resiliency test campaign is defined as an XML file. Figure 4 illustrates an XML file that specifies an example of resiliency evaluation. This campaign is called "nova-apitest", where Controller1 Node will be injected with several nova-api crash faults. Nova-api is executed as one of processes in the component Nova, so we kill the process to simulate faults. The faults are planned to be generated five times per test, timeline of which is Poisson distributed. In the evaluation, we built up a cloud system by reusing three nodes production system and cloning a Compute2 Node in the production phase as a fresh compute node. After log in the target system, an evaluation is conducted by CRGauge. The evaluation contains 5 test cycles, each cycle lasting 360 seconds. Some workloads would be generated in the tests, including booting instances and terminating instances. Instances are booted at a randomly generated timeline which satisfies a uniform distribution to mimic a real world workload.

The *Deployer* component provisions the test system if the target system is in the test phase, conducts the required configurations and installs the fault injection tools in the test system. The configuration and installation operations require access into the test system. Authentication information such as the user and password is used by the *Deployer* to log in the VMs and execute commands during the operations. When a production system is cloned into the test system, silo VLANs are created for the test system so that the host names and IP addresses of the VMs can be preserved in the test system. This feature is needed for successful execution of certain software and workloads of the production system on the test system.

Certain parameters for the resiliency test campaign, including *duration* and *frequency,* are defined in the XML. *Duration* denotes the maximum time of a single test campaign so that if there is system hang or application hang incurred by the fault injection, the experiment is forced to end and next experiment starts. *Frequency* denotes how many experiments to be conducted in a resiliency test campaign.

Deployer first confirms the information transferred from *Loader* and defines the cloud name (e.g., OpenStack, CloudStack or other clouds) in the <*cloud*> section, so as to switch to the certain cloud API in Libcloud. In the second step, if the target system is a production one, *Deployer* could just log in the system by authentication information according to the <*host*> section. Otherwise, *Deployer* would clone VMs from other systems or deploy a new system by automation software or build-from-spec. The cloned VMs would be set as an attribute (cloned = "True") in the <*host*> section. Their former names are listed in the subsection <*formername*>. The automation software would be executed following the commands documented in <*automation*> section. Each software may have their own commands and the configuration file should be created before evaluation. For example, Cookbooks must be configured in Chef for deploying a cloud system. After deployment, the timer is started and notify *Controller* to begin evaluation.

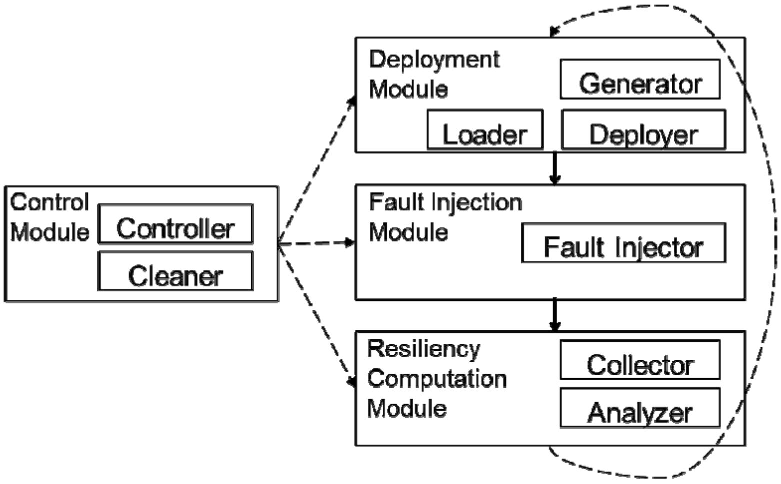
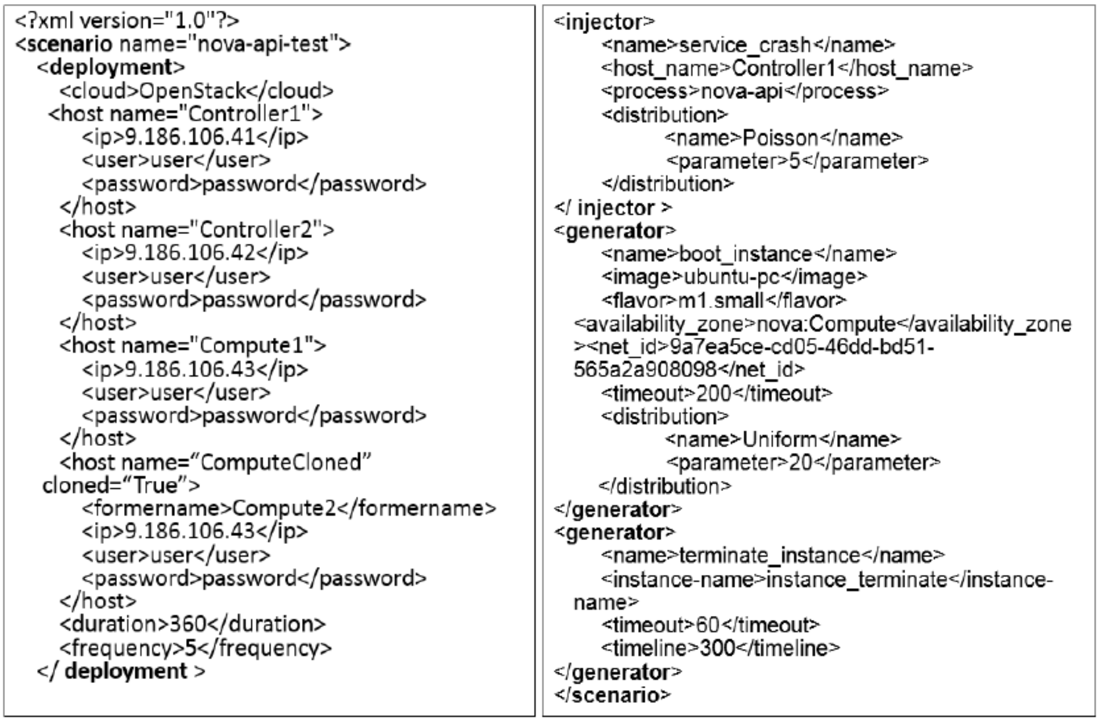


Fig. 3. CRGauge architecture


```
<?xml version="1.0"?>
<scenario name="nova-api-test">
  <deployment>
    <cloud>OpenStack</cloud>
   <host name="Controller1">
       <ip>9.186.106.41</ip>
       <user>user</user>
       <password>password</password>
    </host>
    <host name="Controller2">
       <ip>9.186.106.42</ip>
       <user>user</user>
       <password>password</password>
    </host>
    <host name="Compute1">
       <ip>9.186.106.43</ip>
       <user>user</user>
       <password>password</password>
    </host>
    <host name="ComputeCloned"
  cloned="True">
       <formername>Compute2</formername>
       <ip>9.186.106.43</ip>
       <user>user</user>
       <password>password</password>
    </host>
    <duration>360</duration>
    <frequency>5</frequency>
  </ deployment >
```

```
<injector>
      <name>service_crash</name>
      <host_name>Controller1</host_name>
      <process>nova-api</process>
      <distribution>
            <name>Poisson</name>
            <parameter>5</parameter>
      </distribution>
</ injector >
<generator>
      <name>boot_instance</name>
      <image>ubuntu-pc</image>
      <flavor>m1.small</flavor>
  <availability_zone>nova:Compute</availability_zone
  ><net_id>9a7ea5ce-cd05-46dd-bd51-
  565a2a908098</net_id>
      <timeout>200</timeout>
      <distribution>
            <name>Uniform</name>
            <parameter>20</parameter>
      </distribution>
</generator>
<generator>
      <name>terminate_instance</name>
      <instance-name>instance_terminate</instance-
  name>
      <timeout>60</timeout>
      <timeline>300</timeline>
</generator>
</scenario>
```

Fig. 4. Example XML file of a scenario

Generator creates a workload with requests during the resiliency test campaign. This component either feeds real requests to the test cloud environment if the records of real requests are available, or generates a synthetic suite of requests for the test campaign. These requests invoke the APIs of the test cloud environment for initiating different cloud operations. There are three different user categories: Basic users, advanced users and administrators. Basic users typically provision virtual machines and deploy their applications on the VMs; advanced users have more control on their VMs like establishing several isolated and private networks; administrators are responsible for management of VMs, e.g., *resize* and *live migration*. Example requests issued by different user categories are listed in Table 1.

The generated synthetic suite of requests is a combination of different categories of requests to mimic the real workload. The *Generator* allows users of the CRGauge tool to define probability distributions for different cloud operations, which enables flexible customization of the workloads.

Fault Injection Module

Fault injector tools inject various types of faults into the target system. The specification of the faults to be injected is exemplified by the <*injector*> section in Fig. 4. Attributes of fault injection include: *name,* the type of fault; *host_name*, the identity of the machine to be fault-injected; *distribution* or *timeline*, specification of fault injection occasion or trigger; and other attributes specific to individual fault types (e.g., *process*, target process of the fault injection and etc.).

Distribution and *timeline* are two ways of specifying the time fault injection triggered in the evaluation. *Distribution* specifies a certain distribution (Poisson Distribution, Uniform Distribution, etc.) and the time series of the fault injection occasions will be randomly generated by the *Controller* according to the distribution type and the parameter settings associated with the distribution. *Timeline* directly specifies the time series of the fault injection occasions.

In the example given in Fig. 4, a "*service_crash*" type of fault is specified to be injected into the process *nova-api* running at the machine *Controller1* during a test campaign. The fault injection module maintains a number of injectors for various types of faults. These injectors understand how to interpret the specified attributes for the corresponding fault types and how to perform the injection accordingly. The target of fault injection can be a VM, an application in the VM, hardware underlying VMs and cloud stack software. In this example, the fault injector will crash the Nova-api process, a component of the OpenStack cloud software, at probabilistic occasions with the specified Poisson distribution.

Particularly, as CRGauge deals with cloud resiliency, the focused types of faults in the fault injection module are those related to cloud stack software. Table 2 lists a couple of typical failure classes of cloud management software. Brief descriptions of the failure classes are given below.

Network Disconnection is one common type of hardware failure that causes unavailability of cloud management capabilities. For example, in OpenStack all of VM's network traffic goes through a *network node* and VM's network is managed by a *controller node*. A failure of the network node or the controller node brings network disconnection and simulates network disconnection.

Server Crash shuts down a running VM and tests the dependency between the VM and the cloud system.

Cloud Management Component Failure is a failure of cloud management components. These components are responsible for provisioning and managing virtual machines.

Table 1. Request description

User category	Typical operations
Basic user	Launch and manage instances
	Attach IP
	Attach volume
Advanced user	Upload and manage images
	Configure access and security for instances
	Create and manage networks
	Create and manage volumes
Administrator	Manage projects and users
	Create and manage roles
	Manage instances
	Manage volumes and volume types
	Create and manage images
	Manage flavors
	Check cloud quota and usage
	Create and manage aggregates

Table 2. Failure description

Failure class	Failure Target
Network Disconnection	management network Storage network Virtual network
Server Failure	server crash
Cloud Management Component Failure	service crash
Fabric Component Failure	communication failure (e.g., Message Queue) database failure
High Availability Service Failure	HAProxy crash Pacemaker crash Coresync crash

Fabric Component Failure is a failure of a cloud system's communication, database and relevant middleware. Most cloud management system is loosely coupled and leverages third-party software tools that provide certain capabilities. For instance, an OpenStack deployment may have different implementations of AMQP and uses RabbitMQ, Qpid, or ZeroMQ as the message broker for the AMQP implementations. The objective of the CRGauge is to study the resiliency of the cloud operations against failures of these cloud fabric components.

High Availability Service Failure is a failure of the cloud system's high-availability capabilities or components. Many cloud systems are equipped with high-availability capabilities or components for improving resiliency of the delivered cloud services. For example, *Keepalived* (*Keepalived*) or *Pacemaker* (*Pacemaker*) is popularly utilized in OpenStack deployments for fault tolerance and high availability. *HAProxy* (*HAProxy*) prevents both stateless services and stateful services of OpenStack, including database and AMQP services, from single point of failure. Faults are injected into these services to assess the impact of introducing them into the target cloud system.

The CRGauge tool is designed and implemented to support extendibility. When a new type of fault is to be supported by CRGauge, what needs to be done is just write one Python script within the CRGauge framework. The Python script will interpret the parameter settings relevant to the fault type and conduct the corresponding fault injection.

Resiliency Computation Module

Collector measures several metrics of cloud systems and detects failures in the test. It detects cloud failures through cloud software heartbeats and request based heartbeats. Components of cloud software always have a built-in heartbeat mechanism to check the aliveness of its service. For instance, OpenStack will check itself whether nova services and neutron agents are alive or not. *Collector* collects the heartbeat status from cloud software and determines cloud service aliveness. Another way to detect cloud failures is based on request heartbeat provided by *Generator*. As mentioned above, we set a timeout for each request in *Generator*. For requests generated in the test, *Collector* will check the correctness of the response and the response time. If the result is not correct or the processing of the request does not complete within timeout, the service is considered as unavailable.

Besides measuring failure behavior, *Collector* also performs the measurements of the recovery behavior in the target Cloud. Multiple types of recovery are conducted in the test and the corresponding recovery time values are recorded by the *Collector*.

Components in *Resiliency Computation Module* calculate the resiliency, or availability, based on a specific resiliency model. *Collector* transfers the measurement data to *Analyzer*, where user-specified resiliency models are used to compute the availability values for individual Cloud operations and/or the entire Cloud system. Certain parameters of the resiliency models are given in addition to the state transition diagrams of the target Cloud components' resiliency (as exemplified in Fig. 2), including probability distributions of different types of failures and recovery time values for certain types of failures that are difficult to measure programmatically (e.g., when an error is only detected by relevant stuff, three-shift daily work schedule and one-shift-only work schedule result in different recovery time).

With the combination of the measurement data and user-provided resiliency models and parameters, *Analyzer* calculates the resiliency of cloud resiliency (individual Cloud operations and/or the entire Cloud system).

Controller Module

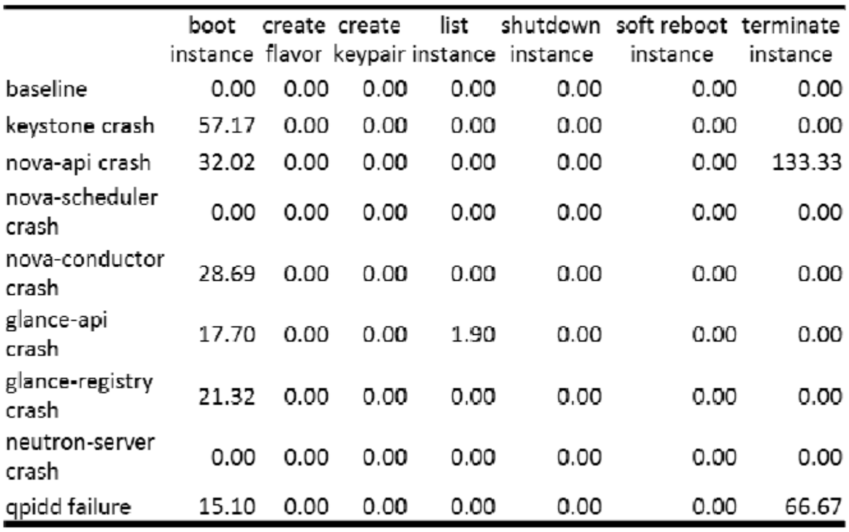
Controller is designed to plan and control all workflows in the CRGauge. After a cycle of evaluation, *Cleaner* cleans up cloud and rolls back to original environment and *Controller* starts the next round of test. CRGauge is designed to test cloud resiliency automatically, so it is necessary to clean up the test legacy and reset the target system to the original environment after a test. To eliminate the vestige from

Generator and *Fault Injector*, *Cleaner* leverages the cloud APIs and restores to the prior-experiment state. *Cleaner* will also reboot cloud services and operating system to eliminate the effect of *Fault Injector*. Snapshot would be leveraged by *Cleaner* if the target cloud system is composed by virtual machines.

Experimental Study and Analysis

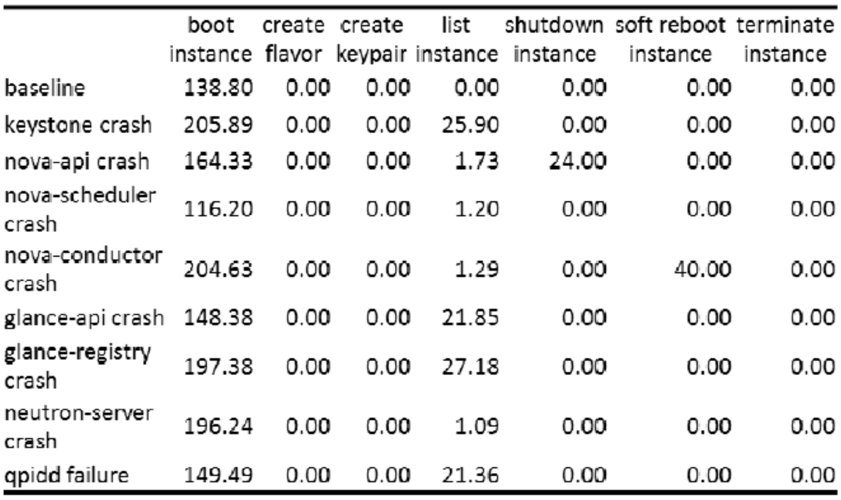
Experimental Setup

We evaluate cloud resiliency on RHEL 6.5 operating systems. In the test, VMs are all provisioned from an Ubuntu Server 12.04 image. We deploy an OpenStack cloud (Havana Release) with High Availability configurations according to "OpenStack High Availability Guide". The target cloud is consisted of part of OpenStack components, including Keystone (Identity service), Nova (Compute), Glance (Image service) and Neutron (Networking services). These components are deployed into two controller nodes, both of which can be backup for the other and one compute node. The compute node provides virtualization capability by KVM hypervisor. In controller nodes, we deploy HAProxy and Pacemaker services as well, in order to protect following OpenStack and fabric processes: Keystone, Nova-api, Nova-scheduler, Novaconductor, Glance-api, Glance-registry, Neutron-server, Neutron-dhcp, Neutron-metadata and Qpid. Stateless services are active-active, load-balanced by HAProxy and HAProxy itself is active-passive, load-balanced by VIP. Pacemaker monitors OpenStack stateful services and brings the backup of these services online as necessary (Haas, 2014).



	boot instance	create flavor	create keypair	list instance	shutdown instance	soft reboot instance	terminate instance
baseline	0.00	0.00	0.00	0.00	0.00	0.00	0.00
keystone crash	57.17	0.00	0.00	0.00	0.00	0.00	0.00
nova-api crash	32.02	0.00	0.00	0.00	0.00	0.00	133.33
nova-scheduler crash	0.00	0.00	0.00	0.00	0.00	0.00	0.00
nova-conductor crash	28.69	0.00	0.00	0.00	0.00	0.00	0.00
glance-api crash	17.70	0.00	0.00	1.90	0.00	0.00	0.00
glance-registry crash	21.32	0.00	0.00	0.00	0.00	0.00	0.00
neutron-server crash	0.00	0.00	0.00	0.00	0.00	0.00	0.00
qpidd failure	15.10	0.00	0.00	0.00	0.00	0.00	66.67

Fig. 5. Experimental results with light workload



	boot instance	create flavor	create keypair	list instance	shutdown instance	soft reboot instance	terminate instance
baseline	138.80	0.00	0.00	0.00	0.00	0.00	0.00
keystone crash	205.89	0.00	0.00	25.90	0.00	0.00	0.00
nova-api crash	164.33	0.00	0.00	1.73	24.00	0.00	0.00
nova-scheduler crash	116.20	0.00	0.00	1.20	0.00	0.00	0.00
nova-conductor crash	204.63	0.00	0.00	1.29	0.00	40.00	0.00
glance-api crash	148.38	0.00	0.00	21.85	0.00	0.00	0.00
glance-registry crash	197.38	0.00	0.00	27.18	0.00	0.00	0.00
neutron-server crash	196.24	0.00	0.00	1.09	0.00	0.00	0.00
qpidd failure	149.49	0.00	0.00	21.36	0.00	0.00	0.00

Fig. 6. Experimental results with heavy workload

Table 3. Experiment configuration for light and heavy workloads

Scenario	Duration (seconds)	Injected fault	Boot instance	Create flavor	Create keypair	List instance	Shutdown instance	Soft reboot instance	Terminate instance
Light	1200	1	6	5	5	5	1	1	1
Heavy	1200	10	25	10	10	20	1	1	1
Timeout			600	30	25	30	120	200	200

We define two typical scenarios to imitate light and heavy workloads. Total test duration is 1200 seconds. For light workload, cloud requests contains boot 6 VMs, terminate 1 VM, shutdown 1 VM, soft reboot 1 VM, create 5 keypairs, create 5 flavors and list cloud resources 5 times. For heavy workloads, cloud requests are similar to the light ones except the number of injections and requests. We inject following faults in the target cloud: Keystone crash, qpid failure, nova-api crash, nova-scheduler crash, glanceapi crash, glance-registry crash and neutron server crash. Table 3 illustrates the detailed information for these two scenarios. The third row shows the timeout setting for corresponding requests, by which we can determine whether a request performs well.

Result and Analysis

We tested resiliency on the target cloud system. The downtimes during the tests under light and heavy workloads are listed in Fig. 5 and 6.

In the light workload tests, most of requests pass the test in the case of fault injection. “Boot Instance” and “Terminate Instance” are two requests, where injections cause downtime in the cloud system according to the result. On one hand, almost all of the injections lead to incapability of provisioning VMs except nova-scheduler and neutronserver crashes. The target cloud suffers most badly when keystone and nova-api crashes, because they cause the longest downtime. On the other hand, only keystone and nova-api crashes result in downtime for terminating instance. These evidences may suggest keystone and novaapi are two vulnerable services in the target cloud.

In the tests with heavy workload, only “Create Flavor”, “Create Keypair” and “Terminate Instance” are not affected by fault injection. There exists more or less downtime when generating other requests. “Boot Instance” suffers worst among requests. We notice even baseline can cause a downtime during provisioning VMs, which means the target system sometime can't afford the heavy workload even without fault injection. Keystone crash can result in longest downtime, which means keystone crash has the greatest impact on cloud resiliency. Comparing two workloads, we can discover that the target cloud system performs less resiliently in heavy workloads than that does in light workloads. The target cloud system also can't withstand too many faults occurrence. We can conclude that “Boot Instance” is the most request that can't pass the test and keystone is the most vulnerable service in the target cloud.

Conclusion

In this study, we design the CEM methodology to continuously evaluate cloud resiliency. CRGauge tool is implemented based on the CEM methodology. In our experiments, we detect the resiliency problems in OpenStack cloud system, especially in the heavy workload. Because CRGauge is inherently capable of extend to more complex and practical scenarios, we will evaluate cloud resiliency with these scenarios in the future.

Acknowledgement

We thank Yanqi Wang for her discussions about the OpenStack resiliency, which is quite valuable for our understanding of OpenStack and experiment setup.

Author’s Contributions

Xiaoyong Yuan: Developed the CRGauge tool, conducted systematic fault injection experiments, and performed analysis of experiment results. He wrote many sections of the paper.

Long Wang: Analyzed the problem of continuous evaluation of cloud resiliency, and architected the CEM methodology and the CRGauge tool. He worked closely with Xiaoyong Yuan during the development of the tool and the experiments (e.g., design of experiments, adjustment of experiments based on results). He wrote many sections of the paper.

Tiancheng Liu: Studied the problem of continuous evaluation of cloud resiliency and co-architected the CRGauge tool. He worked closely with Xiaoyong Yuan during the development of the tool and the experiments (design of experiments, adjustment of experiments based on results). He made important revisions to most sections of the paper.

Yue Zhang: worked on the setup of the experiments, and gave important suggestions on design of experiments. He also revised the experiment part of the paper.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and no ethical issues involved.

References

Apache Libcloud, https://libcloud.apache.org/

Banzai, T., H. Koizumi, R. Kanbayashi, T. Imada and T. Hanawa *et al*., 2010. D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, May 17-20, IEEE Xplore Press, Melbourne, Australia, pp: 631-636. DOI: 10.1109/CCGRID.2010.72

Chef, https://www.chef.io/chef/

CloudStack, https://cloudstack.apache.org/

Fujita, H., Y. Matsuno, T. Hanawa, M. Sato and S. Kato *et al*., 2012. DS-bench toolset: Tools for dependability benchmarking with simulation and assurance. Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Jun. 25-28, IEEE Xplore Press, Boston, MA., pp: 1-8. DOI: 10.1109/DSN.2012.6263915

Haas, F., 2014. OpenStack High Availability Guide.

HAProxy, www.haproxy.org

Keepalived, http://www.keepalived.org

NNT, 2009. Cloud or fog? The business realities of cloud computing for UK enterprises. NTT Comm.

OpenStack. www.openstack.org

PaceMaker. http://clusterlabs.org

Puppet. https://puppetlabs.com/

Silva, M., M.R. Hines, D. Gallo, Q. Liu and K.D. Ryu *et al*., 2013. CloudBench: Experiment automation for cloud environments. Proceedings of the IEEE International Conference on Cloud Engineering, Mar. 25-27, IEEE Xplore Press, Redwood City, CA., pp: 302-311. DOI: 10.1109/IC2E.2013.33

Sobel, W., S. Subramanyam, A. Sucharitakul, J. Nguyen and H. Wong *et al*., 2008. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. Proc. of CCA.

SPEC. Standard Performance Evaluation Coporation (SPEC). https://www.spec.org/

TPCB. TPC Benchmarks, http://www.tpc.org/information/benchmarks.asp

Vieira, M., H. Madeira, K. Sachs and S. Kounev, 2012. Resilience Benchmarking. In: Resilience Assessment and Evaluation of Computing Systems, Wolter, K., A. Avritzer, M. Vieira and A. van Moorsel (Eds.), Springer, ISBN-10: 3642290329, pp: 283-301.

Von Eicken, T., 2011. Amazon EC2 outage: Summary and lessons learned. RightScale, Inc.

Whittaker, Z., 2013. Amazon Web Services suffers outage, takes down Vine, Instagram, others with it. CBS Interactive.