Anomaly Detection for OpenStack Services with Process-Related Topological Analysis

Tomonobu Niwa*, Yuki Kasuya† and Takeshi Kitahara‡
KDDI Research, Inc.
2-1-15 Ohara, Fujimino, Saitama, JAPAN
Email: {*to-niwa, †yu-kasuya, ‡kitahara}@kddi-research.jp

***Abstract*—OpenStack has become the de-facto standard open source software for managing virtualized infrastructure for NFV, however, operators are facing increased complexity of fault management for OpenStack due to its black-box modular architecture and half-yearly version updates. This hinders operators from promptly identifying the root cause of failure or anomalies in OpenStack services. In this paper, we propose an anomaly detection framework for OpenStack in order to identify the root process of anomalies underlying OpenStack services. The framework utilizes a process relational graph and an anomaly detection technique with a centroid-based clustering algorithm. We demonstrate experiments with regards to two use cases and prove the framework to enable discovery of the root process that is responsible for the anomalous situation.**

***Keywords*—OpenStack, Fault management, Machine learning, GraphDB**

I. Introduction

Service providers and telecom companies must tackle issues related to virtualized infrastructure technologies. Telecom companies expect Network Function Virtualization (NFV) to accelerate time-to-market of services and reduce OPEX through the adoption of virtualization technology instead of the existing manual-based operational style. As implementation of virtualized infrastructure in the NFV field, OpenStack [1] has become almost de-facto software since the mushroom growth of OpenStack. This growth has been supported by a successful open source community of more than 70,000 members over the world including application developers, cloud providers, telecom companies, hardware vendors and so forth [2]. OpenStack is designed as modular architecture, and its services are coupled loosely and associated with each other through standardized APIs. It accelerates the development process in terms of independent coding and granular testing. Hence, diverse functional requirements for the virtualized infrastructure stemming from various fields including NFV are being introduced into OpenStack with half-year development cycles. At present, the functions required for NFV such as live migration of Virtual Machine (VM) and support for Data Plane Development Kit (DPDK) have been implemented.

However, the need to keep up with such development cycles makes the operation of OpenStack complex. Operators need to completely grasp the latest developments in about a dozen OpenStack services, including Nova, Glance, Neutron, and Keystone [3]. OpenStack allows a user to select which services to install so that the OpenStack environment can differ according to the user requirement, which leads to further operational complexity. Under such difficult circumstances, the key concern is fault management to maintain the health of the system: detect, mitigate and recover fault/anomalous conditions as soon as possible. In particular, anomaly detection that adequately triggers recovery action is essential.

There are several solutions for detecting the root cause of an anomaly. One straightforward approach is basic log analysis monitoring of the various types of log files generated by OpenStack services and hooking predefined phrases as fault events. Root cause analysis by correlating logs and Remote Procedure Call (RPC) communication has also been proposed [4], [5]. However, some kinds of OpenStack services (e.g., nova-consoleauth, which is responsible for authorizing users access to the console of VM) rarely output log messages in a pragmatic log level "info." If the log level is set to "debug," the management system will be exposed to a continuous heavy load. In addition, there is no strict governance for log descriptions, which are different from service to service and version to version as well, and thus maintenance is costly. Another approach is to periodically call OpenStack APIs to check whether the service is alive [6]. This is simple but its effectiveness is limited, because the root cause of an anomaly cannot be identified even if an invalid response to the API is received. This is because operators need to understand the interaction between OpenStack services related to the API. Yet another approach is scenario-based online testing in a commercial environment. OpenStack has an official testing project called "Tempest" [7], which enables operators to test hundreds of pre-configured scenarios. This approach is useful for detecting bugs and/or misconfiguration but is not suitable for keeping track of the system health. Hundreds of tests periodically run in a commercial environment and the results, including the execution time, are verified. However, operators do not favor online testing, for example, a test VM is created and deleted by a commercial VM. The maintenance of test scenarios depending on the version is also cost-ineffective as well as log analysis. In summary, the complexity of anomaly detection for OpenStack comes from the independency of services frequently updating the version in a short cycle, which imposes increasing costs on operators.

The above approaches depend on the versions and/or specifications of OpenStack. Therefore, we focus on the fact

978-3-901882-98-2 © 2017 IFIP

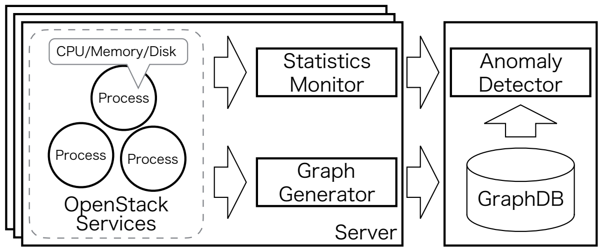


Fig. 1. Anomaly process detection framework.

that OpenStack services are composed of various processes. If we can recognize the anomalous processes underlying OpenStack services, even if the root cause of the anomaly is blind, it would be possible to provisionally react to an anomaly to keep OpenStack well. This is because anomalous services/processes can be isolated since OpenStack has high modularity. In this paper, we propose the root "process" of the anomaly detection framework for OpenStack by monitoring statistics and relationships between processes composing OpenStack services, which will help operators to identify the root cause of an anomaly as well. Our proposed framework does not depend on the versions and services installed, and the operator can recognize an anomalous process without deep knowledge about OpenStack. The approach was demonstrated in a testbed and the experiment verified its feasibility. The results show the framework successfully discovers defective processes that cause OpenStack malfunctions.

II. Anomaly Process Detection Framework

OpenStack consists of a dozen services that interact with each other, and operators face a complex maze in an anomalous system situation. In order to specify an anomaly, we propose a topological anomaly detection approach that consists of the three functional parts that we developed: Graph Generator, Statistics Monitor and Anomaly Detector, as shown in Fig. 1. Graph Generator creates a graph of the relationships between processes. Statistics Monitor collects the statistical data and computes the degree of divergence from typical statistics behavior. Anomaly Detector receives the graph and the degree of divergence, and identifies the anomalous process.

A. Graph Generator

OpenStack provides a virtualized infrastructure through a variety of complementary services, each of which is composed of several processes including middleware such as database

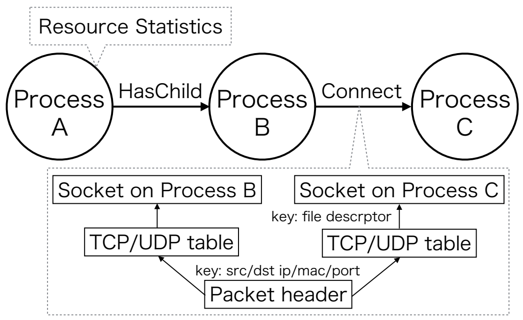


Fig. 2. Graph model.

TABLE I
Metrics of resource statistics.

Category	Metrics	Description
CPU	user	Time consumed by user process
	system	Time consumed by system process
	children_user	Time consumed by child user process
	children_system	Time consumed by child system process
	voluntary	Number voluntary context switches performed
	involuntary	Number involuntary context switches performed
Memory	rss	Non-swapped physical memory a process has used
	vms	Total amount of virtual memory used by the process
	shared	The amount of memory potentially shared with other processes
	text	The amount of memory devoted to executable code
	lib	The amount of memory used by shared libraries
	data	The amount of physical memory by other than executable code
	dirty	Number of dirty pages
Disk	read_count	Number of read operations performed
	write_count	Number of write operations performed
	read_bytes	Number of bytes read
	write_bytes	Number of bytes written
	read_chars	The amount of bytes passed to read() and pread() syscalls
	write_chars	The amount of bytes passed to write() and pwrite() syscalls

and message queue processes. They intricately communicate via a network. We utilize this feature, that is, Graph Generator creates an OpenStack-process graph by capturing the communications and statistics of processes. Note that we assume OpenStack services run on Linux. Figure 2 shows a graph model we defined, where a node indicates a process and an edge is a relationship between the processes. There are two types of edges: "HasChild" and "Connect." The former means a process generates its child process. Several OpenStack processes folk sub-processes and then "HasChild" edge helps to identify which processes are made up of the services. The latter denotes the communication between the processes. The process consumes sockets to communicate, which are assigned unique identifiers, called file descriptors. In Linux, TCP/UDP tables in /proc/net/* have routing information such as source/destination IP address, mac address, port number, and an information related to file descriptor. Therefore, the relationships between TCP/UDP connections and the process that utilizes them can be recognized. Thus, "Connect" edge can be generated by associating captured packet headers passing between processes with the TCP/UDP connection utilized by the process. In addition, each node has an attribute with respect to resource statistics consumed by the process, which is explained in the next section in detail. Although Graph Generator works in each server where OpenStack services run, an additional load is relatively small, which is less than 5 percent of the CPU load.

B. Statistics Monitor

With regard to each process, three categories of resource statistics are collected as shown in Table I. While collectable statistics of processes are limited compared with that of Linux operating system, typical metrics for CPU, memory and disk I/O can be gathered. Statistics Monitor working in each server collects these metrics at a small interval, and then computes the degree of divergence from typical statistical behavior of each process. In order to specify the conditions of the process, we utilize a centroid-based clustering algorithm, which works by updating candidates for centroids to be the mean of the

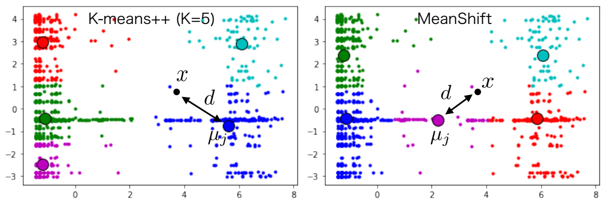


Fig. 3. Nova-compute statistics clustered by K-means and MeanShift.

points in a cluster. In other words, the centroids represent the typical behavior of the process. In this paper, we utilize two methods: K-means++ [8] and MeanShift [9] clustering. The former algorithm relies on the distance between points and the latter is based on the density.

1) K-means++: K-means algorithm updates candidates for centroids so that the following criteria is minimized:

$$\sum_{i=1}^{n} \min_{\boldsymbol{\mu_j} \in C} \|\boldsymbol{x_i} - \boldsymbol{\mu_j}\|^2 \tag{1}$$

, where $\boldsymbol{x_i}$ is the vector of normalized statistical data and n is the number of points. In K disjoint clusters C, each cluster has a centroid $\boldsymbol{\mu_j}(j = 1, 2, \cdots, K)$. K-means has the problem that the clustering result depends on the initialization of the centroids. K-means++ resolves the issue through initializing the centroids to be generally distant from each other.

2) MeanShift: MeanShift also chooses centroids. Given a candidate centroid $\boldsymbol{\mu_j}$ for iteration t, the candidate is updated according to the following:

$$\boldsymbol{\mu_j}^{(t+1)} = \boldsymbol{\mu_j^t} + m(\boldsymbol{\mu_j^t}) \tag{2}$$

$m(\cdot)$ is a mean shift vector that is calculated for each centroid so that the density of points is maximized.

$$m(\boldsymbol{\mu_j}) = \frac{\sum_{\boldsymbol{x_i} \in S(\boldsymbol{\mu_j})} f(\boldsymbol{x_i}, \boldsymbol{\mu_j})\boldsymbol{x_i}}{\sum_{\boldsymbol{x_i} \in S(\boldsymbol{\mu_j})} f(\boldsymbol{x_i}, \boldsymbol{\mu_j})} \tag{3}$$

, where $S(\boldsymbol{\mu_j})$ is the neighborhood of points within a given distance around $\boldsymbol{\mu_j}$, and $f(\cdot)$ is the kernel function for deciding the weight of $\boldsymbol{x_i}$. In this paper, radial basis function (RBF) kernel, which is a kind of Gaussian filter, is utilized.

Figure 3 illustrates an example clustering result by both algorithms for the "nova-compute" process, which is a Nova service function for managing VMs. There are 5 centroids in both cases, while the coordinates are slightly different. These centroids symbolize typical statistical behavior, and then the degree of divergence from that is calculated as the distance between a point to be verified and the nearest centroid. Furthermore, so as to make the distance unified, we defined the degree of anomaly as d, which is the distance divided by the standard deviation σ_j of the cluster selected as the nearest. The degree of anomaly from each process will be sent to Anomaly Detector at an interval of several seconds and the averaged values are verified.

$$d = \left\| \frac{(\boldsymbol{x} - \boldsymbol{\mu_j})}{\boldsymbol{\sigma_j}} \right\| \tag{4}$$

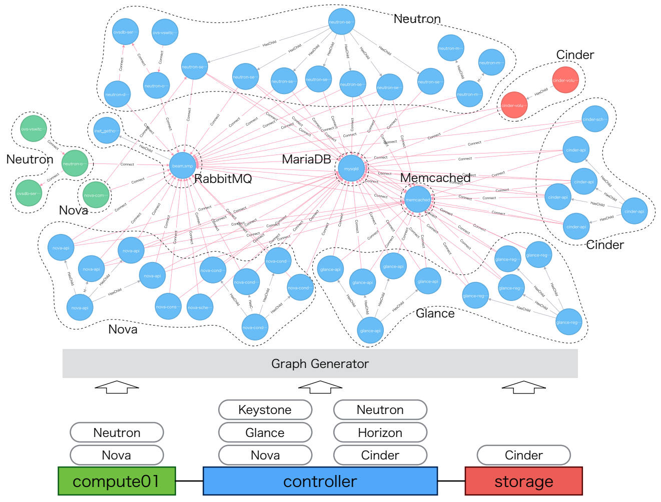


Fig. 4. Testbed and OpenStack-process relationships.

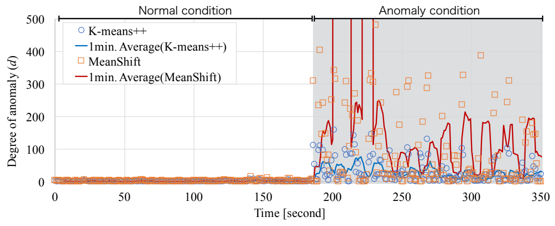
C. Anomaly Detector

Our purpose is to identify the root process that causes an anomalous situation. Toward this goal, Anomaly Detector verifies d in each process and extracts a subgraph where nodes have d over a threshold ε and connected by "HasChild" edges to them. This subgraph is equivalent to the set of processes influenced by the anomaly. The root process of the anomaly is discovered by forwarding nodes of the subgraph. Eventually, the process that is responsible for the anomaly can be detected.

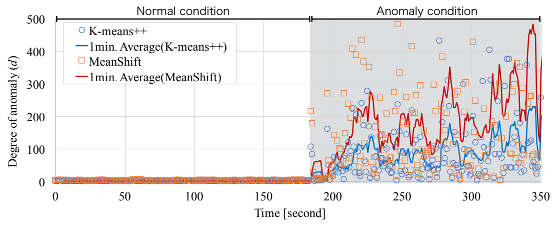
III. Experiment and Evaluation

In order to verify the proposed framework, we set up a testbed depicted in Fig. 4. The testbed consists of three servers: controller, compute01 and storage servers. While the testbed does not completely simulate a commercial environment in terms of the scale, it enables to verify the functionality including an accuracy of anomaly detection. The controller server has 3.10 GHz CPU with 4 cores, 16 GB memory and 1TB HDD, and runs 64 bit CentOS 7.3. The remaining two servers have the same specifications except for 8 GB memory. Six types of OpenStack services, Keystone, Glance, Nova, Neutron, Horizon and Cinder, are installed across the servers along with the basic system configuration described in an install tutorial. The OpenStack version is Newton, which is the 14th official release. The proposed framework is mainly implemented by Python. As a database to store the graph data transmitted from Graph Generators, Neo4j [10] which is an open source graph database equipped with a graphical web interface and query language is utilized.

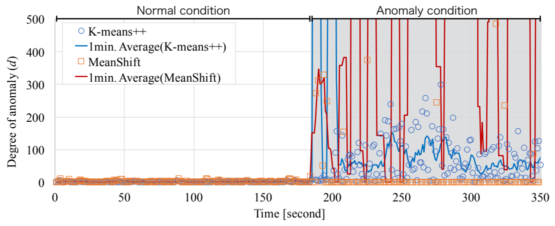
Figure 4 also shows the graph configured by the testbed, which contains 53 nodes and 115 edges to be stored. As seen from the graph, several middleware play important roles: mysqld node for MariaDB service is attached to 31 edges, beam.smp node for RabbitMQ service, which is a default message queueing middleware in OpenStack, has 29 edges, and memcached node has 20 edges. Several types of processes: nova-api, nova-conductor, glance-api, glance-registry and cinder-api, have 4 child processes, while neutron-server has 7 child processes.



(a) Use Case 1: mysqld node on controller server



(b) Use Case 2: beam.smp node on controller server



(c) Use Case 2: nova-compute node on compute01 server

Fig. 5. Degree of anomaly.

Assuming normal condition to train, OpenStack services were randomly operated in an hour, such as building VMs, deleting VMs, registering images and so forth.

A. Anomaly use cases

Statistics Monitor collects statistics in 10-second intervals and then computes the degree of anomaly for each process. In the paper, we demonstrated two anomaly use cases.

1) Heavy load in database: MariaDB stores various information utilized by OpenStack services. We applied an additional load to the database by the client simulator, mysqlslap. The simulated 100 clients simultaneously accessed and executed SQL on the connected database. The CPU load is less than 10 percent under normal conditions, while it reached 100 percent in the simulated case.

2) Queue congestion: OpenStack services communicate with each other via the message queuing service RabbitMQ. Hence, the queue congestion degrades the OpenStack services. To simulate this situation, we intentionally input dummy messages into the queue subscribed by one of nova-compute processes running on compute01 server. The dummy messages were transmitted at the rate of 100 messages per second, while a few messages are passed in the normal case.

B. Evaluation

The computed degrees of anomaly were periodically plotted at 10-second intervals and values averaged over 1 minute were evaluated. The number of K-means++ clusters was set to 3 in the evaluation since it achieves better results than other parameters while it differs by processes. Figure 5 represents the results of anomaly detection; (a) for Use Case 1 and (b)-(c) for Use Case 2. The part of the white background color is normal condition and that of gray is under the anomaly simulated. The average lines by K-means++ and MeanShift in both use cases fluctuated immediately after the anomaly occurred. Note that the MeanShift approach is more sensitive. Table II shows a comparison with two approaches. As we can confirm from the figure, both approaches achieved high accuracy, however, MeanShift was likely to overfit in the experiments because it was inferior to K-means++ in terms of precision. Thus, we can conclude K-means++ outperformed in this testbed.

TABLE II
DETECTION ACCURACY.

(a) K-means++

	Threshold ε											
	5	10	15	20	25	30	35	40	45	50	55	60
Accuracy	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
Precision	0.81	0.94	0.94	0.96	0.97	0.97	0.97	0.97	1.00	1.00	1.00	1.00
Recall	0.99	0.95	0.89	0.83	0.75	0.70	0.64	0.58	0.51	0.47	0.43	0.39
F-measure	0.89	0.94	0.92	0.89	0.85	0.81	0.77	0.73	0.67	0.64	0.60	0.56

(b) MeanShift

	Threshold ε											
	5	10	15	20	25	30	35	40	45	50	55	60
Accuracy	0.96	0.99	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00
Precision	0.37	0.59	0.67	0.72	0.76	0.82	0.87	0.91	0.90	0.94	0.93	0.93
Recall	1.00	0.99	0.97	0.96	0.93	0.92	0.89	0.88	0.87	0.86	0.85	0.82
F-measure	0.54	0.74	0.80	0.82	0.83	0.87	0.88	0.89	0.89	0.90	0.89	0.87

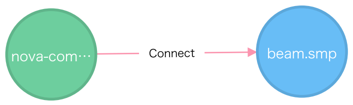


Fig. 6. Subgraph for Use Case 2.

In order to discover the root process of an anomaly, subgraphs were generated with K-means++ and the threshold $\varepsilon = 10$ since the parameter can achieve the highest F-measure value as shown in Table II. For Use Case 1, only mysqld node was extracted and then we could recognize the root process of the anomaly apparently. On the other hand, two anomalous nodes were detected in Use Case 2: nova-compute node on the compute01 server and beam.smp node on the controller server. Figure 6 illustrates the subgraph. The root process of the anomaly was specified by tracing the edges toward the direction of the arrow and then beams.smp nodes could be identified as the root. Use Case 2 simulated the queue congestion and thus the correct result was obtained.

IV. Summary

OpenStack has become the de-facto standard open source software for managing virtualized infrastructure in NFV. In this paper, we propose an anomaly detection framework for OpenStack that enables the root process of an anomalous situation to be discovered by generating a process-relational graph and per-process anomaly detection using a centroid-based clustering algorithm. We demonstrated two anomaly use cases for experiments. The results show the framework is successful for identifying the root process of an anomalous OpenStack condition.

References

[1] "OpenStack Community," [Online] https://www.openstack.org/

[2] "OpenStack User Survey," [Online] https://www.openstack.org/assets/survey/April2017SurveyReport.pdf

[3] "OpenStack services," [Online] https://docs.openstack.org/admin-guide/common/get-started-openstack-services.html

[4] Dhruv Sharma, Rishabh Poddar, Kshiteej Mahajan, Mohan Dhawan and Vijay Mann, "Hansel: Diagnosing Faults in OpenStack," in Proc. the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT) 2015.

[5] Xiaoen, Livio Soares, Kang G. Shin, Kyung Dong Ryu and Dilma Da Silva, "On Fault Resilience of OpenStack, " in Proc. the 4th annual Symposium on Cloud Computing (SOCC) 2013.

[6] Pooya Musavi, Bram Adams and Foutse Khomh, "Experience Report: An Empirical Study of API Failures in OpenStack Cloud Environments," in Proc. 27th International Symposium on Software Reliability Engineering (ISSRE) 2016.

[7] "Tempest Testing Project," [Online] https://docs.openstack.org/developer/tempest/

[8] David Arthur and Sergei Vassilvitskii, "k-means++: The Advantages of Careful Seeeding," in Proc. the 18th annual ACM-SIAM symposium on Discrete algorithms (SODA) 2007.

[9] Dorin Comaniciu and Peter Meer, "Mean Shift: A Robust Approach Toward Feature Space Analysis," IEEE Trans. Pattern Analysis and Machine Intelligence, Vol. 24, No. 5, pp. 603-619, 2002.

[10] "Neo4j," [Online] https://neo4j.com/