Debugging OpenStack Problems Using a State Graph Approach

Yong Xiang, Hu Li, Sen Wang, Charley Peter Chen, Wei Xu

Institute for Interdisciplinary Information Sciences, Tsinghua University

{xiangy13, lihu12, wangs12}@mails.tsinghua.edu.cn, {ctj2015, weixu}@mail.tsinghua.edu.cn

Abstract

It is hard to operate and debug systems like OpenStack that integrate many independently developed modules with multiple levels of abstractions. A major challenge is to navigate through the complex dependencies and relationships of the states in different modules or subsystems, to ensure the correctness and consistency of these states. We present a system that captures the runtime states and events from the entire OpenStack-Ceph stack, and automatically organizes these data into a graph that we call *system operation state graph* (SOSG). With SOSG we can use intuitive graph traversal techniques to solve problems like reasoning about the state of a virtual machine. Also, using a graph-based anomaly detection, we can automatically discover hidden problems in OpenStack. We have a scalable implementation of SOSG, and evaluate the approach on a 125-node production OpenStack cluster, finding a number of interesting problems.

Keywords state graph, debugging, anomaly detection, OpenStack

1. Introduction

So-called cloud computing infrastructures are designed to organize servers, networking devices and storage systems into a virtual resource pool, hoping to simplify system operation. OpenStack [1] is a very popular open source cloud management system. However, even after several years of development, there are still many issues with OpenStack, making the system itself challenging to operate.

Beyond the commonly discussed code quality control issues in the OpenStack developer community, we believe systems like OpenStack are essentially hard to debug and operate due to the way they are designed.

OpenStack integrates many open source or proprietary software systems to perform different tasks, and the management system itself also has a number of asynchronously connected modules. There are six major components in OpenStack, namely Nova (compute), Neutron (networking), Cinder (block storage), Swift (object storage), Glance (image) and Keystone (authentication), as well as a number of optional components to provide layer-3 routing, accounting etc. Each module maintains its own database, and communicates through a persistent queue service [2].

Many of the modules provide extensible interfaces, allowing different backends for actual implementation. For example, in our configuration, Nova uses *libvirt* [3] to control a *kvm* [4] hypervisor to provide server virtualization, and Neutron uses *Open vSwitch (OVS)* [5] to provide virtual networks. To make things even more complicated, it is common to use Ceph [6] as the storage backend, which introduces another complex distributed system.

OpenStack and many similar systems are hard to operate mainly because of the following reasons.

1) The system states are distributed over the entire system and exist in many levels of abstraction, with a considerable amount of duplications. For example, the relevant states of a virtual machine (VM) spread in Nova, libvirt / kvm, Neutron, Open vSwitch, the routing agent, as well as the storage service. For the VM to work, two conditions must hold: 1) all these components work, and 2) their states are consistent. Unfortunately, neither of the two always holds in OpenStack.

2) Development benefits from the modularity in OpenStack design, but maintenance requires the operator to understand all modules, which is an impossible task. Each module has its own set of tools for state monitoring and other operations, and these components are located at different machines.

Even experienced operators may spend lots of time looking into the piles of manuals and running many commands to answer a simple query like *should I shut down physical machine A, which VMs would be affected?* The question is more difficult than just listing the VMs on the machine, as the operator also needs to know what other services run on the machine. For example, there might be a routing service or a disk block on the server that other VMs are actively using.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

APSys 2016, August 4–5, 2016, Hong Kong, China.
Copyright © 2016 ACM 978-1-4503-4265-0/16/08. . . $15.00
DOI: http://dx.doi.org/10.1145/2967360.2967366

3) Existing automated debugging tools, both log-based [7] and code analysis based [8], only work on a single code base with consistent identifiers. There is no system-wide ID in OpenStack, especially crossing different layers and integrating with Ceph. In addition, the state updates are asynchronous and hard to track.

In this paper, we propose a novel approach to capture the knowledge about the system runtime states in a graph, which we call *system operation state graph* (SOSG). SOSG is designed to solve problems for operators. We show that with a simple procedure, we can automatically construct the graph, which reveals many hidden links among different system modules. SOSG is designed to be general. We do not assume much knowledge about the target system, but only need the log file locations and a list of commands to extract states from different modules. Specifically, we do not need to understand the semantics of these data. Also, all required raw data are at a component level rather than the system level, and thus easy to provide by module developers.

Specifically, SOSG captures *entities* in the entire OpenStack system at different layers. There are numerous types of entities, including VM, network and storage blocks. For an entity, we also capture its *states*, such as Nova database records and libvirt states for a VM. We keep not only the current state, but also previous states. We also capture the *events* (i.e. log messages) related to the entity, which are useful in debugging.

SOSG automatically discovers the links among different entities, even across multiple modules, using a syntactic string matching on automatically discovered identifiers in events and states. Capturing both system states and events, and analyzing the links among entities eliminate the need for a global ID in systems, and thus provide more information for debugging. For example, based on common strings in file / directory names and unique IDs in logs, we can infer the relationship between a VM and a Ceph data block it uses, which never appears in logs.

We present two applications of SOSG. 1) we turn ad hoc system state queries into graph traversals, simplifying these queries, especially those spanning multiple modules; 2) we perform an automatic anomaly detection to find VMs that behave differently, which might indicate problems.

We implement SOSG on both Neo4j [9] and GraphX [10] and evaluate it using real data from a professionally operated production-quality 125-node OpenStack-Ceph cluster with over 100 active users. We process 40 GB raw data into a graph with 43 million vertices and 57 million edges, on which we perform traversals and anomaly detections using a GraphX cluster. We successfully detect many hidden or user-visible failures such as resource reclaim issues, state inconsistencies, and VM migration failures.

Although this paper mainly focuses on OpenStack debugging, we believe our approach is general in that 1) the data we use, such as log files, state query commands and database records, widely exist in many systems, and 2) our graph construction is completely syntactic, depending on little knowledge about OpenStack itself.

In summary, our contributions in this paper include:

1) We propose an approach that organizes information about system states and events into a single graph representation, SOSG, with which we can solve many complicated state queries with a common graph traversal.

2) We design an anomaly detection algorithm that automatically analyzes the state graph and finds many problems based on the graph structure.

3) We provide two scalable implementations of the SOSG. Preliminary evaluations using real operation data from a production OpenStack cluster show promising results.

The rest of the paper is organized as follows: we review related work in Section 2. We show how to process the raw system operation data into a state graph in Section 3, and present the applications of the state graph in Section 4. We report the results of several case studies in Section 5. We discuss the future work and conclude in Section 6.

2. Related Work

Automatic system diagnostics. Bugs are inevitable in systems, and people have designed many approaches to automatically detect system bugs, using both static analysis of the source code and runtime data such as logs. The authors of [7] detect problems using common identifiers in text logs. It is important to combine multiple data sources. California Fault Lines [12] combines router configurations, syslogs with email logs to recover from failures of an email service. SherLog [8] is an example of using powerful static code analysis to help improve logging. And [11] finds similar code patterns to build an actionable alert prediction model. All these methods are limited to a single code base, which we do not have.

Debugging multiple frameworks is a new topic. Pivot Tracing [13] monitors multiple frameworks in distributed systems using dynamic instrumentation, and supports relational operators to process the collected data on the fly. Paper [14] diagnoses distributed system performance changes by tracing request flows end-to-end across components. Adding the traces creates common identifiers in a heterogeneous system. Our approach only uses existing information in systems without extra instrumentations, and thus it is easier to deploy.

Anomaly detection has been widely used in system problem detection [15]. [16] takes advantage of Spark to perform large scale anomaly detection, and applies it to detect VM performance problems. [17] uses anomaly detection to find faults in a multi-tier web system with redundancy. These projects use a small number of data sources, while we mainly focus on analyzing states across different system components.

Table 1. Data sources and their corresponding types

Type	Data source
DB	OpenStack databases updates (triggers)
Libvirt	libvirt status `Python API`
Ovs	OVS status `ovsdb-client dump`
Cephimage	Ceph image list `rbd info`
Cephfile	Ceph block file `ls /ceph/dir/file`
Cephlog	log files from Ceph
Log	logs from all OpenStack components

Anomaly detection methods. There are many anomaly detection methods for different types of data. [18, 19] provide techniques to simplify data from heterogeneous sources to improve anomaly detection result. Distance-based anomaly detection [20–22] is a special technique allowing the anomaly to be described by a probability model. Graph anomaly detection is also a well studied topic [23, 24].

Scalable graph computation. The recent development of efficient graph computation frameworks, such as Pregel [25], Power-Graph [26], GraphX and Apache Giraph [27], enables our approach. Specifically, we use GraphX to process the giant state graph efficiently.

Knowledge base and knowledge graphs. The knowledge graph is a special case of the knowledge base (KB), a well studied topic in artificial intelligence. There are many popular knowledge base systems, such as Knowledge Vaults [28], YAGO [29–31], DBpedia [32], Freebase [33], and NELL [34]. And also many efforts are devoted to build these KBs [35] [36]. Our state graph is similar to the knowledge graph, but it specifically targets on machine generated system states, and thus can also be built automatically.

3. State Graph

In this section, we introduce our core data structure, the system operation state graph, and show how we construct the graph from a number of heterogeneous raw text files. In the next section, we introduce two applications we have developed on the graph, graph-traversal based state queries and automatic anomaly detection.

3.1 Data structure

Our core data structure is the state graph. It is a special version of the property graph [37] automatically constructed from raw operation data. The property graph is a directed multigraph [10] allowing user defined properties attached to any vertex or edge. The multigraph supports parallel edges to capture multiple relationships between the same vertex pairs.

Table 1 summarizes the different data sources we use to generate the state graph. For each *record* in any data source, there is a corresponding vertex in the state graph. We use the data source to determine the *data type* of the vertex, as Table 1 shows. The vertex contains a list of key-value pairs (i.e. host: n005) as properties.

Vertices. In a state graph, there are three categories of vertices: *entities*, *states* and *events*. Figure 1 shows an example of a state graph with all these categories of vertices.

Entity vertices are the central pieces in the state graph, as they represent instances of components or resources in the system, such as a VM, a disk block or a physical server. Specifically, we do not distinguish the *identifier of an entity* from the *entity* itself. That is, we treat an *IP address* the same as *the server with the IP address*, or the *UUID of a VM* the same as the *VM*. This choice is due to the limitations of the raw textual data, and the lack of need for distinction. Vertex 3 and 6 in Figure 1 are examples of entity vertices.

However, the raw data do not directly contain entities. Instead, they contain *states* of an entity at a specific time (i.e. a VM is running/stopped/paused in libvirt), or *events* involving certain entities (i.e. a log message says that a VM starts to shut down). As we detail in Section 3.2, we extract all entity vertices from the state and event vertices in the graph. For example, vertex 1, 2 are both event vertices and vertex 4, 5 are both state vertices in Figure 1.

Edges. There are two types of edges in the state graph: *spatial edges* and *temporal edges*. *Spatial edges* capture the relationship between an entity and its states (entity-state), as well as its associated events (entity-event). Note that in our current representation, we do not have entity-entity edges. Instead, we represent an entity-entity relationship using a path of (entity1 $\rightarrow$ state / event $\rightarrow$ entity2). Note that a state or event vertex acts as the “bridge” between two entities. In Figure 1, edges *a, b, c, d, e, f* are all spatial edges. There is an edge between vertex 1 and 3 because the event contains the entity. From the figure we can infer the relationship between entity vertices 3 (xxx-xx1) and 6 (10.1.0.12) by following the path ($3 \rightarrow d \rightarrow 5 \rightarrow f \rightarrow 6$).

Temporal edges represent the time order of states and events that connected to the same entity. The temporal edges make it easy to traverse events or state changes in time. The edge always points to the ascending time direction. Edge *g* and *h* in Figure 1 are both temporal edges.

3.2 State graph construction

Our goal is to construct the state graph from the raw data sources in Table 1, without using any semantic information. Here we outline our construction algorithm.

Step 1: Parse raw text data to generate event and state vertices. The raw data are heterogeneous, including free texts, semi-structured texts and structured records. They are encoded in different formats such as JSON, csv or mysql dump files. We provide a set of parsers for each data format to extract the information from text into key-value pairs. Each *record* (as defined by the parser) from the data source is turned into a vertex, with the data source encoded as the data type of the vertex. Then we add the key-value pairs to

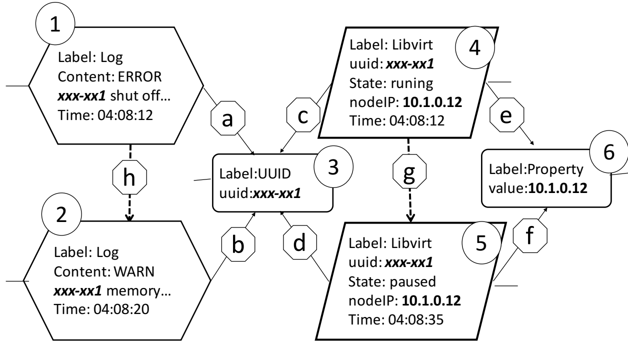


Figure 1. A slice of an example state graph. The rectangles, parallelograms, and hexagons represent entity, state and event vertices, respectively. The numbers and letter labels are used in the discussion in the text.

the vertex as its properties. In Figure 1, vertices 1, 2, 4 and 5 are generated in this step.

Step 2: Discover and generate entity vertices. Note that not every single property of the state and event vertices represents an entity. We need to discover the property values that might be an identifier for some entity. While manually labeling them is feasible in a small system, we decide to automatically discover them based on statistical properties, following the identifier discovery method in [7]. Simply speaking, we find the properties with many distinct values and each value appears at multiple places, and use the values as identifiers. In OpenStack, this method works quite well. In this paper, we require a property to have at least 2 distinct values, and the ratio between the number of distinct values and total values to be greater than a threshold. We try different thresholds, and we show that we can detect entity identifiers quite accurately. We take these identifiers and generate an entity vertex for each distinct value. In Figure 1, vertices 3 and 6 are generated in this step.

Step 3: Add spatial edges. We generate an edge connecting a state or event vertex with an entity vertex, *iff* the state or event contains the entity. The *containing* is purely syntactic, that is, the identifier of the entity literally appears in a state or event. The simple syntactic rules lead to an applicability to many different systems, while effective enough, as we will show in our evaluation.

We also label the edge with the property name, indicating the semantics of the relationships. Though these labels are not used in this paper, they can be used to form natural language queries in future work. Note that for efficiency reasons, in a real implementation, we combine this step with Step 2, with some careful bookkeeping when computing of the distinct values. We omit the details here due to space constraints. In Figure 1, we add edges *a, b, c, d, e* and *f*.

Step 4: Add temporal edges. To create the temporal edges, we firstly group the state and event vertices by the entity they are associated with (a many-to-many association). For all events / states associated with an entity, we sort them by time, and create temporal edges according to ascending time order. This step adds the edges *g* and *h* in Figure 1.

Summary. After the four steps above, we have a complete state graph capturing both temporal and spatial information. The procedure has two advantages: 1) it is fully syntax driven, using only textual and simple statistical features, without any external semantic information or human labeling; 2) every step in the procedure contains many independent operations, and thus is easy to parallelize with a graph computation framework. Currently we generate the graph from scratch, and we are working on algorithms to incrementally update the graph to support online applications.

4. Applications of the State Graph

While there are many potential applications of the state graph, we present two of them in this preliminary paper.

4.1 System state query as graph traversal

The direct application of the state graph is answering system state queries, so that system operators can find all state information in a homogeneous data structure with a single method - graph traversal, instead of memorizing tons of different commands.

Queries only involving a single entity are straightforward. We only need to find the *most recent* state / event vertex connected to the entity vertex, and look up its properties. It is trickier to discover states involving multiple entities, such as the physical location of *a specific data block in a volume of a VM*. In this case, we need to traverse the graph through a (entity $\rightarrow$ state / event $\rightarrow$ entity $\rightarrow$ state / event $\rightarrow$ $\ldots$) path. We can use breadth-first-search (BFS) to find the path, and can use data type information to reduce the search space for BFS. We omit the details of the optimization due to space constraints, and instead provide a concrete example in Section 5.2.

We implement the graph traversal on both Neo4j [9] and GraphX, and we provide a number of convenient functions for common tasks, such as `listCephfilesForVM`, or `listVMsInSubnet`. Note that these functions are based on a common underlying graph traversal mechanism, showing that our core techniques are general.

4.2 Anomaly detection

While the graph traversal can find answers to specific queries from operators, there are millions of states in the system, and many issues remain hidden for a long time without being noticed. For example, we have a number of resource release failures, hidden for months in our public system (detail in Section 5.3). We would like to automatically analyze the entire graph to find these hidden problems. Currently we focus on detecting abnormal states, as the problem detection from events has been extensively studied [7, 38, 39], and

not all the problems are captured in event logs. We leave the combining of events and states as important future work.

As a preliminary attempt, we use graph-based anomaly detection. We choose anomaly detection because it is an unsupervised algorithm, and does not require manually labeled failure samples. The basic assumption of anomaly detection is that most of the states and entities are normal, and any deviation from the normal case indicates problems. This is true in our production OpenStack, where most VMs have a common set of states. Thus, our goal is to find VMs whose states are different from their peers.

Feature extraction: subgraph of a VM. The first task is to extract the *features* of a VM which capture the state. As the normal operation of a VM depends on all components the VM uses, such as network, disk and security groups, we need to find the subgraph that roots at the VM entity vertex and also includes all its dependencies. We can naively perform a BFS starting from the VM entity to find relevant entities.

One practical difficulty is that many entities, such as a subnet, are shared among different VMs. Naive BFS will expand the subgraph through this shared subnet vertex into the resources of other VMs. Thus, we introduce an algorithm augmenting the naive BFS with a *collaborative pruning* methodology to discover shared vertices, and prevent the BFS search from passing through the shared vertices.

Intuitively, the algorithm works as follows. We start BFS traversal from every single VM vertex. Then on every vertex along the path, we record a list of BFS traversals that have reached it before. During a traversal, if we reach a vertex that has been reached by another BFS, the later BFS stops there. In this way, we find not only subgraphs rooted at each VM, but also the information about shared resources.

Distance-based anomaly detection. We define a distance metric between two subgraphs to capture whether the two subgraphs are similar to each other. The goal of the distance metric is to tell a problematic VM from the normal ones.

We try to capture the structure information of the subgraphs in the distance metric. Specifically, we capture the information about *triplets* [10]. A triplet is two vertices along with the edge. We encode a triplet information such as the data type of the source / destination vertex, and its location in the subgraph (i.e. the depth of the BFS traversal). Then we define the distance between the two subgraphs using a generalized Jaccard distance [41].

Finally, we apply the distance-based anomaly detection to find objects with fewer than k neighbors within a radius of r. The parameters k and r directly affect our detection results. As a first attempt, we determine them empirically. As a future work, we will investigate more powerful anomaly detection techniques with more intuitive parameters.

5. Preliminary Evaluation

We will discuss some case studies in our preliminary evaluation in this section. Our evaluation is based on real operation data from a 125-node production cluster that runs OpenStack (Icehouse) and Ceph. There are 5 OpenStack controller nodes, 120 compute nodes, 40 of which also doubles as Ceph storage nodes. Each node has 12 Xeon cores, 128 GB of RAM and 10 GE network. This cluster offers computation and storage services for about 100 active users. The cluster utilization is moderate, with 601 active virtual machines, resulting an overall CPU utilization of 67%, RAM utilization of 71% and disk utilization of 13%.

Raw data collection. We periodically snapshot the state of libvirt (every 60 sec), OVS (every 60 sec), Ceph image (every 600 sec) and Ceph block file (every 3600 sec, with duplicates removed). To capture the database states, we first dump the entire OpenStack DB right at the beginning of the experiments, and create triggers to log all database updates. We also collect log files from all OpenStack and Ceph components. The average collected operation data is 600 MB per hour from all 125 servers. The OVS state snapshot represents about half of the data as the collector does not remove the duplicate records. Logs account for about 24% of the data. Cephfile and Libvirt make up 15 % and 9% respectively. We use a Spark [40] cluster running inside the VMs in the same OpenStack cluster to analyze these data.

In this preliminary evaluation, we present a few case studies of real bugs that take us a long time to diagnose with traditional command line tools. We show SOSG can help discovering / diagnosing these problems. We leave more quantitative user-study of the technique as future work.

5.1 State graph construction performance

We evaluate the performance of constructing state graph from the raw data as discussed above. We build the graph using a 3-day operation data that consist of a number of text files with total 40 GB. We construct the graph following the procedure discussed in Section 3.2. We use a 15-node Spark cluster to accelerate the process.

It takes 5 minutes to parse the text files and construct the event and state vertices, and 8 minutes to expand these vertices to discover entity vertices. Then it takes 12 minutes to add all remaining edges. The total processing time for the 3-day worth of data takes 25 minutes, which is an acceptable cost considering the convenience this graph brings to operators. The resulting graph contains about 43.3 million vertices and 56.6 million edges. Most of the vertices are events (log entries).

5.2 Graph traversal example

We provide a concrete query example. Consider the query *If physical server A encounters a hard disk failure, which VMs are affected?* To answer the question, we need to figure out the relationship among a particular VM, a Ceph image used by the VM and the disks used by the Ceph image.

In the traditional way, the operator needs to look up many information, including which blocks are stored on the disk (`ls /var/lib/ceph/osd/...`), which Ceph im-

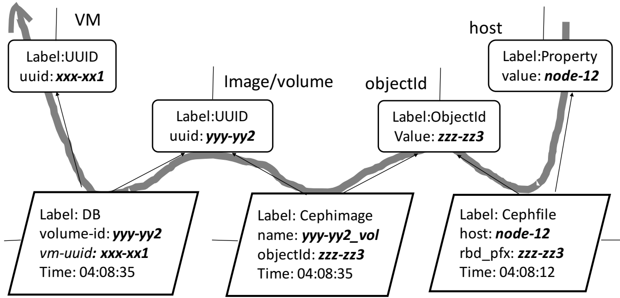


Figure 2. An example path from host to VM across Ceph

age the block belongs to (for a disk (or volume), he may use `rbd info -p compute(or volumes)` to list all images, then `rbd info -p compute(or volumes) <image>` to get the block_name_prefix, and finally match a block file with an image where the filename literally contains the block_name_prefix), where the image is used (for the disk, he may recognize that the VM uuid is literally contained in the image name and confirm it by `nova show <server>`; for the volume, he may use `nova volume-show <volume>` or `cinder show <volume>` to find out the attached VM). Each of the questions requires one or more system specific commands.

Figure 2 shows one of the paths that our graph traversal algorithm automatically discovers. The path starts from a physical server and ends with a VM, across the Ceph states, like (`Property` → `Cephfile` → `ObjectId` → `Cephimage` → `UUID` → `DB` → `UUID`). The query is fast too, and only takes 35 seconds in our setup.

5.3 Case studies: anomalies detection

While we are still going through the manual process evaluating the false positive rates, we present three interesting anomalies as case studies. The first two are both cases of hidden problems. The third one represents a complicated failure that confuses users.

Case 1. Open vSwitch ports not deleted on VM deletion. It is hard to see a resource deletion failure behind the OpenStack UI. The anomaly detection algorithm captures a subgraph, with relevant portion shown in Figure 3(b). It is abnormal because the entity vertex is connected to a sequence of OVS state vertices, whereas the normal case (Figure 3(a)) does not.

As we try to confirm the graph structure-based detection is semantically valid, we inspect the properties of each vertex. We find that the VM instance has been deleted months ago, but still has many active OVS states associated with it. This anomaly indicates that the virtual ports of the VM are not deleted, resulting a resource leak. Note that it is not a common problem (otherwise it will not be an anomaly).

Case 2. Database record does not match physical states. It is common to see inconsistencies between the OpenStack database record and the actual physical state. Figure 3(c) shows a case. The subgraph is picked out by anomaly detection mainly because there are thousands of DB state vertices directly connect to the VM entity vertex.

Looking for the semantic reason, we find that the VM has been in `deleted` state for months, but the libvirt, Cephimage and OVS states still remain. Again, looking into the state vertices connected to the VM vertex, we can see the possible reason: `nova.instance_faults` shows $1,704$ failures in $1,707$ of `nova.instance_actions`. This finding not only indicates an inconsistency case, but also suggests some serious bugs in OpenStack's retrying / recovery mechanism.

Case 3. Failed VM Migration. Lastly, we present a more complex and user visible failure during the VM migration. The user reports the issue as a freezing migration process and has to delete the VM. Anomaly detection algorithm picks out the abnormal subgraph too, as Figure 3(d) shows. The subgraph is abnormal in that the migrating VM is missing libvirt state, both from the source host and the destination host.

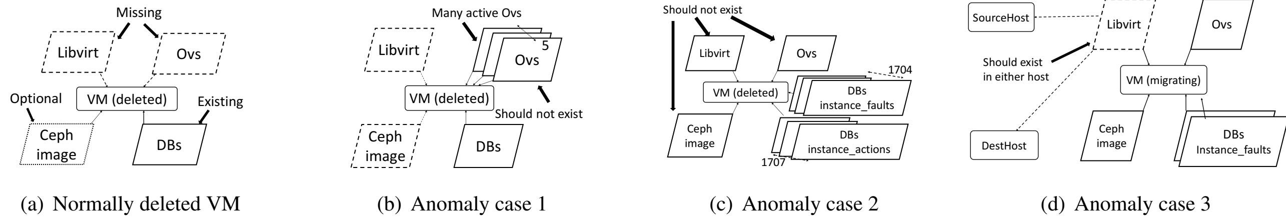
A closer manual inspection shows that an exception happened during this migration from node-118 to node-38, and two database state vertices `nova.instance_faults` connected to the VM show that `cannot remove config /etc/libvirt/qemu/instance-0000155e.xml: Read-only file system` on the source and `error removing image` on the destination. As a result, the storage (virtual disk) of the VM migrated but the computation did not, resulting a failed migration. In addition, the missing libvirt state vertices on both node-118 and node-38 serve as another evidence of the unsuccessful migration.

An even deeper inspection at the logs and events associated with this VM vertex indicates that the VM encountered a migration problem: there are 653 repeated `Instance not resizing, skipping migration` records out of all 1653 log lines related to the instance. This repeated skipping of a small-instance (2 VCPUs, 4 GB RAM) migration also suggests some bugs in OpenStack's resource management. While we can detect the anomaly from traditional log analysis only, it is beneficial that we also detect the abnormal state, as it may suggest a quick fix in production (i.e. reinitializing the VM).

6. Future Work and Conclusion

We will concentrate on the following important directions as our future work.

Analyzing events and state history. As we indicate (with manual analysis) in Section 5.3, event sequences (logs) are an indication *why* the system ends up in an inconsistent state. We would like to have a model that maps events to the corresponding anomalous state. The model might help predict the failure before it actually happens.



(a) Normally deleted VM (b) Anomaly case 1 (c) Anomaly case 2 (d) Anomaly case 3

Figure 3. Portions of normal and abnormal subgraphs. Solid box = existing, dashed = missing, dotted lines = optional. Same as in Figure 1, the rectangles, parallelograms and hexagons represent entity, state and event vertices, respectively.

Including other data sources. The state graph is a great way to automatically discover links among different information about a system, both runtime and static. We want to incorporate other static data sources, such as the source code, bug reports and documentations into the graph, and hope to provide more insights into how to *fix* the bugs discovered.

Incremental update of state graph. As states change and events occur continuously, we need to keep updating the state graph online in an incremental way. Supporting the incremental update requires some special considerations in the graph data structure, considering the limitations of GraphX.

Applying SOSG to other systems. Though we have only presented SOSG as a tool for debugging OpenStack, we believe the approach is general. We would like to apply it to detect problems in other distributed systems, such as big data frameworks and general web services organized in a service oriented architecture (SOA).

Conclusion. As both researchers and system operation practitioners, we keep wondering *"What is the core set of knowledge in system operation?"* Most of the time, we believe that it is the experience of knowing about all dependencies, or links among different system components, and the knowledge about different tools to inspect and change the states of these components. Much of the knowledge is too trivial to remember, impossible to transfer to a new system, and hard to teach to another person. All of these problems make system operation hard.

The above is our motivation to build SOSG which captures the runtime information, including both states and events, and discovers the hidden links among these pieces of information. By leveraging modern graph computation capacity, we can process a vast amount of redundant data and automatically construct the graph. With the graph, we turn the typical task such as ad hoc probing of different system components into an intuitive graph traversal problem, making the exploration of heterogeneous systems easier. We also develop a subgraph-based anomaly detection method to automatically analyze system states to find hidden problems. We evaluate SOSG with data from our production OpenStack cluster with dozens of components, and demonstrate its effectiveness.

7. Acknowledgement

We would like to thank Mingtian Yin, Peilun Li, Shun Zheng, Yuhan Su, Yiran Li, Wei Yin, Kui Shi, Sam Su and our colleagues from the lab, Futurewei and EasyStack for their help in building and presenting SOSG. We also thank the anonymous APSys reviewers and our shepherd Lin Tan for their insightful comments and guidance in preparing this paper. This research is supported in part by the National Natural Science Foundation of China (NSFC) Grants 61361136003, 61379088, China 1000 Talent Plan Grants, Tsinghua Initiative Research Program Grants 20151080475, and a Google Faculty Research Award.

References

[1] https://www.openstack.org/

[2] https://www.rabbitmq.com/

[3] https://libvirt.org/

[4] http://www.linux-kvm.org/page/Main_Page

[5] http://openvswitch.org/

[6] Weil, Sage A., et al. "Ceph: A scalable, high-performance distributed file system." Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006.

[7] Xu W, Huang L, Fox A, et al. "Detecting large-scale system problems by mining console logs[C]." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009: 117-132.

[8] Yuan, Ding, et al. "SherLog: error diagnosis by connecting clues from run-time logs." ACM SIGARCH computer architecture news. Vol. 38. No. 1. ACM, 2010.

[9] http://neo4j.com/

[10] Gonzalez, Joseph E., et al. "Graphx: Graph processing in a distributed dataflow framework." 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014.

[11] Ranking I A A. "Finding Patterns in Static Analysis Alerts[J]". 2014.

[12] Turner D, Levchenko K, Snoeren A C, et al. "California fault lines: understanding the causes and impact of network failures[J]". ACM SIGCOMM Computer Communication Review, 2011, 41(4): 315-326.

[13] Mace, Jonathan, Ryan Roelke, and Rodrigo Fonseca. "Pivot tracing: dynamic causal monitoring for distributed systems." Proceedings of the 25th Symposium on Operating Systems Principles. ACM, 2015.

[14] Sambasivan R R, Zheng A X, De Rosa M, et al. "Diagnosing Performance Changes by Comparing Request Flows."." NSDI. 2011.

[15] Wang C, Viswanathan K, Choudur L, et al. "Statistical techniques for online anomaly detection in data centers." Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on. IEEE, 2011: 385-392.

[16] Solaimani M, Iftekhar M, Khan L, et al. "Statistical technique for online anomaly detection using spark over heterogeneous data from multi-source VMware performance data." Big Data (Big Data), 2014 IEEE International Conference on. IEEE, 2014: 1086-1094.

[17] Id T, Kashima H. "Eigenspace-based anomaly detection in computer systems[C]." Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2004: 440-449.

[18] Zhao Y, Zheng Z, Wen H. "Bayesian statistical inference in machine learning anomaly detection." Communications and Intelligence Information Security (ICCIIS), 2010 International Conference on. IEEE, 2010: 113-116.

[19] Dasgupta D, Majumdar N S. "Anomaly detection in multidimensional data using negative selection algorithm." wcci. IEEE, 2002: 1039-1044.

[20] Knorr E M, Ng R T, Tucakov V. "Distance-based outliers: algorithms and applications[J]." The VLDB Journal The International Journal on Very Large Data Bases, 2000, 8(3-4): 237-253.

[21] Xie M, Han S, Tian B. "Highly efficient distance-based anomaly detection through univariate with PCA in wireless sensor networks." Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on. IEEE, 2011: 564-571.

[22] Khoa N L D, Babaie T, Chawla S, et al. "Network anomaly detection using a commute distance based approach[C]." Data Mining Workshops (ICDMW), 2010 IEEE International Conference on. IEEE, 2010: 943-950.

[23] Noble, Caleb C., and Diane J. Cook. "Graph-based anomaly detection." Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003.

[24] Akoglu, Leman, Hanghang Tong, and Danai Koutra. "Graph based anomaly detection and description: a survey." Data Mining and Knowledge Discovery 29.3 (2015): 626-688.

[25] Malewicz, Grzegorz, et al. "Pregel: a system for large-scale graph processing." Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010.

[26] Low, Yucheng, et al. "Distributed GraphLab: a framework for machine learning and data mining in the cloud." Proceedings of the VLDB Endowment 5.8 (2012): 716-727.

[27] Avery, Ching. "Giraph: Large-scale graph processing infrastructure on hadoop." Proceedings of the Hadoop Summit. Santa Clara (2011).

[28] Dong, Xin, et al. "Knowledge vault: A web-scale approach to probabilistic knowledge fusion." Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2014.

[29] Suchanek, Fabian M., Gjergji Kasneci, and Gerhard Weikum. "Yago: a core of semantic knowledge." Proceedings of the 16th international conference on World Wide Web. ACM, 2007.

[30] Hoffart, Johannes, et al. "YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia." Artificial Intelligence 194 (2013): 28-61.

[31] Mahdisoltani, Farzaneh, Joanna Biega, and Fabian Suchanek. "Yago3: A knowledge base from multilingual wikipedias." 7th Biennial Conference on Innovative Data Systems Research. CIDR Conference, 2014.

[32] Auer, Sren, et al. "Dbpedia: A nucleus for a web of open data." Springer Berlin Heidelberg, 2007.

[33] Bollacker, Kurt, et al. "Freebase: a collaboratively created graph database for structuring human knowledge." Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008.

[34] Carlson, Andrew, et al. "Toward an Architecture for Never-Ending Language Learning." AAAI. Vol. 5. 2010.

[35] Weikum, Gerhard, and Martin Theobald. "From information to knowledge: harvesting entities and relationships from web sources." Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2010.

[36] Craven, Mark, et al. "Learning to construct knowledge bases from the World Wide Web." Artificial intelligence 118.1 (2000): 69-113.

[37] I. Robinson, J. Webber, and E. Eifrem. Graph Databases. OReilly Media, Incorporated, 2013

[38] Fonseca, Rodrigo, et al. "X-trace: A pervasive network tracing framework." Proceedings of the 4th USENIX conference on Networked systems design and implementation. USENIX Association, 2007.

[39] Attariyan, Mona, Michael Chow, and Jason Flinn. "X-ray: automating root-cause diagnosis of performance anomalies in production software." Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). 2012.

[40] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[41] https://en.wikipedia.org/wiki/Jaccard_index