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

Introduction

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.

SOSG Data Structure

Our core data structure is the state graph. Table 1 summarizes the different data sources we use to generate the state graph.

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

In a state graph, there are three categories of vertices: *entities*, *states* and *events* (see Figure 1). Entity vertices (i.e. 3, 6) are the central pieces in the state graph, while the raw data only contain states (i.e. 4, 5) of an entity at a specific time, or events (i.e. 1, 2) involving certain entities.

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, i.e. c, d, e, f), as well as its associated events (entity-event, i.e. a, b). *Temporal edges* represent the time order of states (i.e. g) and events (i.e. h) that connected to the same entity.

SOSG Construction

We construct the state graph from the raw data sources in Table 1, without using any semantic information.

Step 1: Parse raw text data to generate event and state vertices. We provide parsers to extract the information from raw data into key-value pairs. Each *record* from the data source is turned into a vertex (i.e. 1, 2, 4, 5), we add the key-value pairs as its properties.

Step 2: Discover and generate entity vertices. We find the properties with many distinct values and each value appears multiple times, and use the values as identifiers. We generate an entity vertex (i.e. 3, 6) for each distinct identifier.

Step 3: Add spatial edges. We generate an edge connecting a state or event vertex with an entity vertex, *iff* the state or event literally contains the entity. In Figure 1, we add edges *a, b, c, d, e* and *f*.

Step 4: Add temporal edges. We group the state and event vertices by the associated entity, sort them by time, and create temporal edges according to ascending time order. This step adds the edges *g* and *h* in Figure 1.

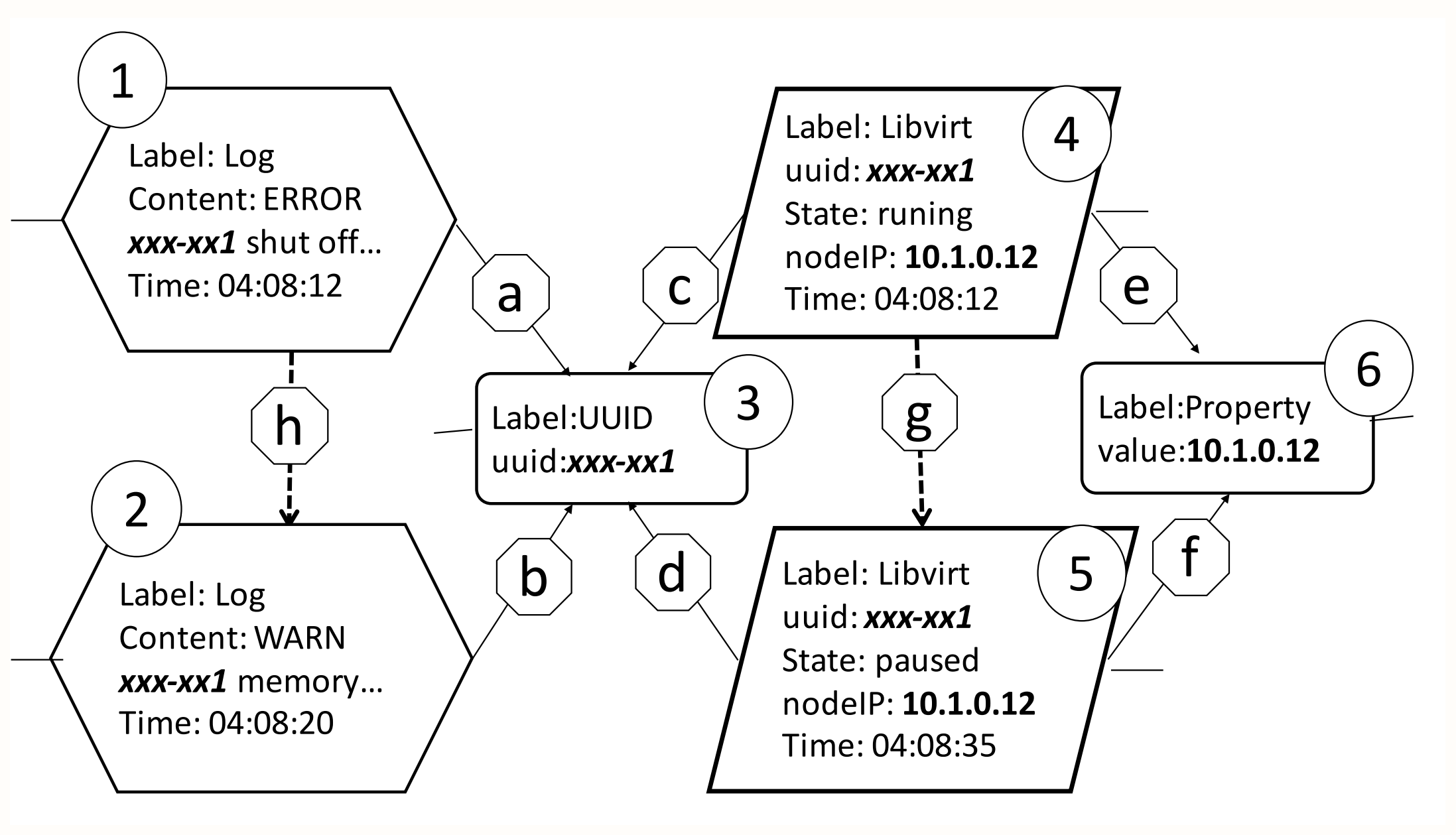

Figure 1 : A slice of an example state graph. The rectangles, parallelograms, and hexagons represent entity, state and event vertices, respectively.

SOSG Applications

System query as graph traversal

The state graph can answer system state queries from operators with a single method - graph traversal, avoiding the memorizing of many different commands.

Consider the query *If physical server A encounters a hard disk failure, which VMs are affected?* 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 image the block belongs to (*rbd info -p compute(or volumes) <image>,...*), where the image is used (*nova show <server>*, *nova volume-show <volume>* or *cinder show <volume>*). Each of the questions requires one or more system specific commands.

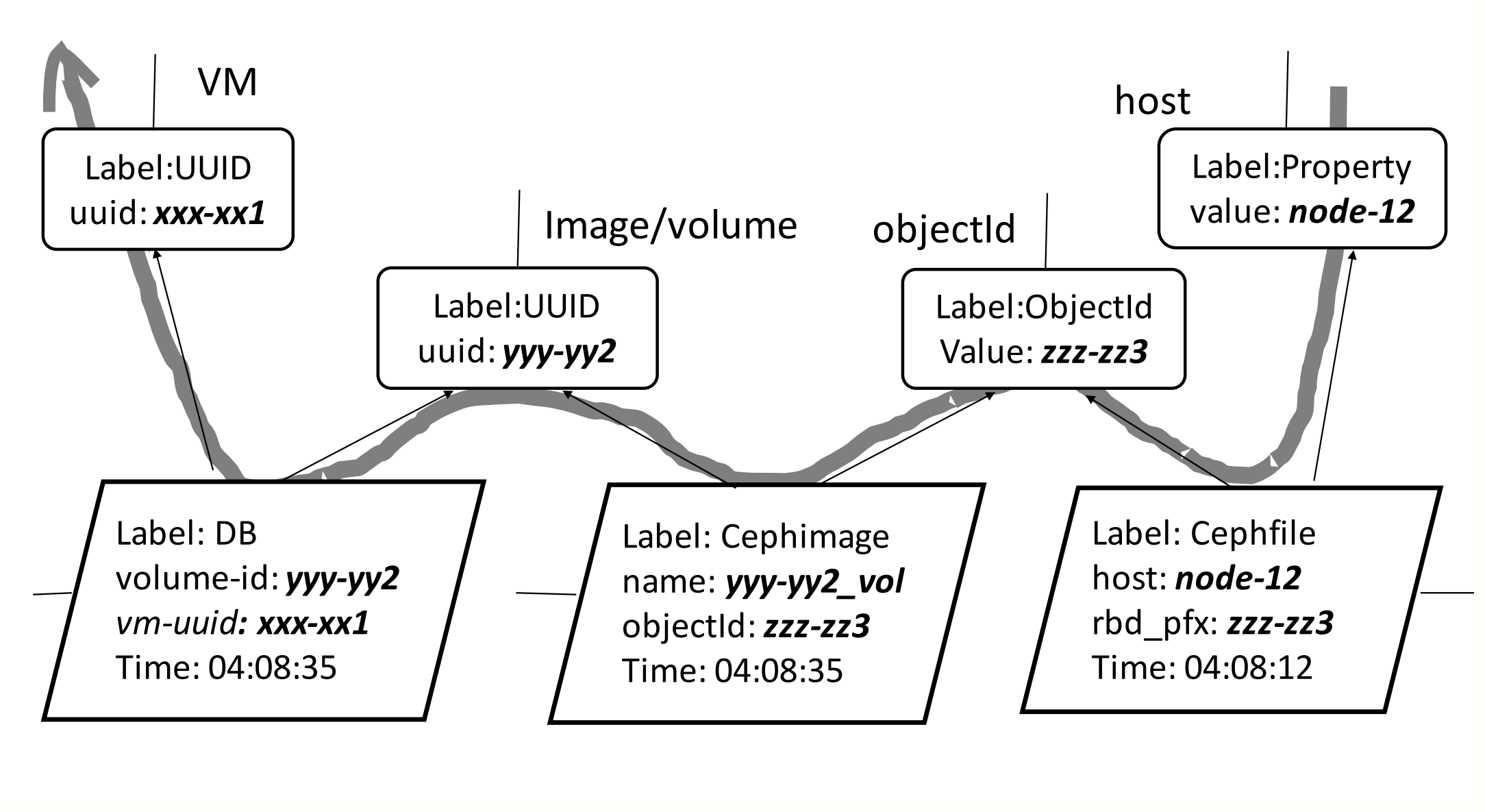

Figure 2 : An example path from host to VM across Ceph

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. The query only takes 35 seconds which is fast.

Anomaly detection case study

Failed VM Migration. It is abnormal in that the migrating VM is missing libvirt state, both from the source host and the destination host. The user reports the issue as a freezing migration process and has to delete the VM. A closer manual inspection shows that an exception happened during this migratiom, the storage (virtual disk) of the VM migrated but the computation did not, resulting a failed migration. An even deeper inspection at the 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 migration also suggests some bugs in OpenStack's resource management

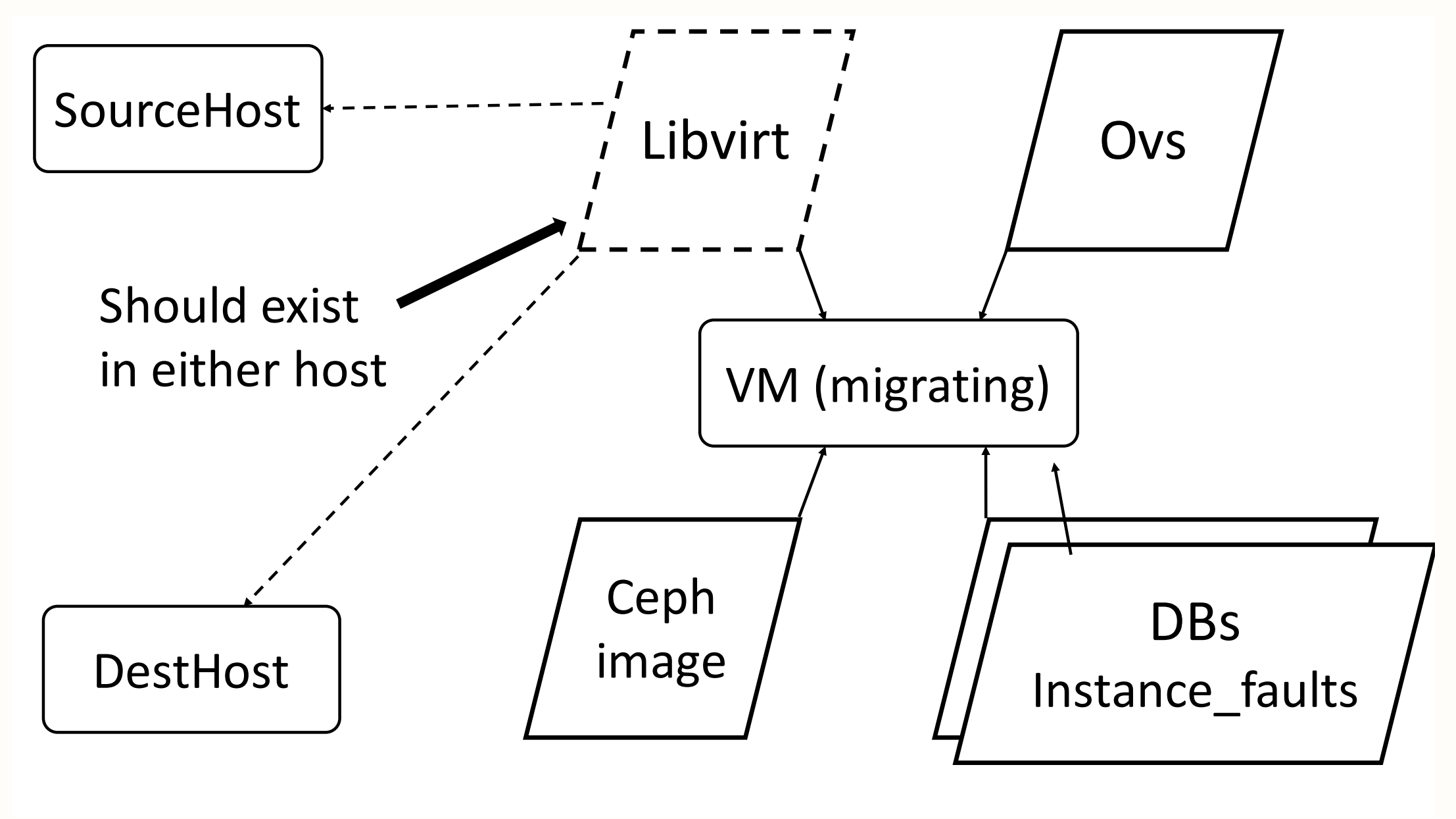

Figure 3 : Migration failure: the expected Libvirt state is missing in both source and destination nodes

Future Work

We will concentrate on the following future work:

Analyzing events and state history We would like to have a model that maps events to the corresponding anomalous state, which might help predict the failure before it actually happens.

Supporting incremental update of state graph. We need to keep updating the state graph online in an incremental way to caputure the continous evolving of states and events.

Including other data sources. We want to incorporate other static data sources (i.e. source code, bug reports and documentations) into the graph, and hope to provide more insights into how to fix the bugs discovered.

Applying SOSG to other systems. 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.