Toward achieving operational excellence in a cloud

S. A. Baset
L. Wang
B. C. Tak
C. Pham
C. Tang

A cloud pools resources such as compute, network, and storage and delivers them quickly and automatically on-demand through software. In addition, it provides automatic and policy-driven management of resources through software. Such a system comprises many components, whose states change rapidly. To manage it effectively, cloud service providers need to clearly understand the behavior of operations across components, and be able to fix errors as early as possible. The task of building such capabilities (referred to as operational excellence) in a cloud system is challenging because components maintain internal state and interact in non-intuitive ways to perform automated operations. In this paper, we discuss the concept of operational excellence for a cloud system, discuss the challenges in achieving the operational excellence, and describe our vision. Toward our vision, we present a set of techniques to determine the causal sequences of system events across distributed components. We also model configured system states using casual sequences of system events, gather observed system states, and continuously verify the configured and observed states across system components. We apply these techniques to study OpenStack®, an open source infrastructure-as-a-service platform.

Introduction

In state-of-the-art cloud systems (CSs), resources such as compute, network, and storage are defined and delivered as software. Furthermore, upper-layer management operations, such as software updates or backups, are also driven by software. A user of these systems can create a software defined network (SDN) [1], provision virtual machines (VMs) and attach them to the SDN, configure VMs to automatically receive software updates driven by policies (e.g., IBM SmartCloud* Enterprise+ (SCE+) [2]), automatically reconfigure a service in response to workload changes or performance degradation, or automatically recover from failures through high-availability support. None of these operations needs intervention from a cloud service provider; instead, these operations are issued automatically by computer programs, which maintain substantial internal states in many components of the cloud platform. These states, known as *soft states* (or software states), are defined and controlled by software and provide cloud platforms with the flexibility and intelligence needed to adapt to different workloads and scenarios.

The presence of soft states across multiple components and layers of a cloud platform, however, increases the operational and management complexity of the system. As an example, SDN [1] allows for the creation of virtual networks that span the physical network. The mapping between the virtual and the physical layer is stored as soft states across one or more components in a CS. As a result, the operational complexity increases when performing logical operations such as VM migration. During a VM migration to a new physical server, not only the VM disk and run-time state needs to be migrated, but the virtual network must also be extended to the new physical host. The automation across multiple components and layers must act in concert for a successful VM migration operation. Any failure during this operation can result in the creation of partial soft-state, which, if not appropriately managed, can give an inconsistent view of the system. Similar issues about soft-state arise when managing and delivering other resources such as compute and storage in software, or upper

Digital Object Identifier: 10.1147/JRD.2014.2298927

layer management operations such as software updates. The automation across multiple components and layers interacts in non-intuitive ways during provisioning, monitoring, or other cloud operations, which makes it difficult to understand the normal execution of the system. The task is further exacerbated by the use of rapidly evolving open source cloud platforms (such as OpenStack** [3], an open source infrastructure-as-a-service [IaaS] platform) in which normal execution flows change substantially across releases.

In this paper, we define the "operational excellence" in a CS to be a set of capabilities for understanding normal execution and identifying and recovering from faults quickly and efficiently. Without such capabilities, the management of a CS can quickly get out of a cloud service provider's control. Operational excellence requires tools that monitor all software resources, including compute, network, and storage, as well as requests that traverse across components and layers of a CS. The tools should be transparent to the system and application execution (i.e., no source code of the system or application, or code instrumentation into the system or application, is needed), while providing sufficient information for the cloud service provider to obtain an in-depth understanding of the CS operation. Based on the obtained understanding, the cloud service provider can still use the tools to study individual components and scenarios of interest in the CS, continuously verify observed states by comparing them against corresponding configured states, and quickly resolve problems.

We have been developing multiple techniques for creating a set of such tools. Our techniques include construction of causal sequences of systems events across distributed components of a CS, injection of faults at various points in the execution flow based on the causal information, precise tracing of failure propagation across distributed components to develop a knowledge base for common faults, building a runbook to guide fast root-cause analysis and error recovery, and continuous verification of configured and observed states across components and log data. The major contributions of this paper include an operational excellence concept and vision for a CS (see the section "Operational excellence vision for a cloud system"), the challenges involved in achieving operational excellence (see the section "Challenges for achieving operational excellence in a cloud system"), and development and presentation of a subset of aforementioned techniques (see "Toward achieving operational excellence in an OpenStack cloud") for achieving operational excellence in an OpenStack-based CS.

Challenges for achieving operational excellence in a cloud system

A CS is a flexible and automated computing environment. In this section, we discuss major challenges for achieving operational excellence in a CS.

Layered automation

To manage the complexity of software defined deployment and management, a CS is usually engineered into multiple layers, and each layer comprises multiple components. The operations among components at the same layer are automated as much as possible. Further, the operations across layers are usually performed through well-defined boundaries (e.g., application programming interface [API] calls). As a result, it is highly desirable for cloud service providers or cloud users to trace the execution of operations and ensure that the operations execute as expected and under control. This capability demands sophisticated tools to be invented to precisely track automated operations among components at the same layer and across layers.

State consistency

As mentioned, a CS comprises many interdependent components. During normal execution, all the components' views of the system state should be consistent. For example, in a CS built using OpenStack, the state of each VM recorded in the OpenStack databases must be consistent with the state of the VM viewed by the hypervisor and any virtual switch (e.g., Open vSwitch [4]). A partial failure of VM provisioning can lead an OpenStack controller to "believe" that the provisioned VM is live and running, but the hypervisor detects a provisioning failure. When there is an error that causes state inconsistency, it should be detected as early as possible. Aside from errors, changes of configuration of a component in a CS can also bring inconsistency of states.

In a CS consisting of hundreds of components or more, proper configuration of these components is a big challenge, especially when the states of the system undergo changes all the time. State changes initiated by software can be unexpected, which greatly impedes error diagnosis and recovery. Thus, it is critical to validate system states by inspecting and correlating configured and observed states across system components and logs.

Problem diagnosis and correction

In a CS, the presence of layered automation and soft state makes it difficult to detect failures, evaluate their impact, and identify the root cause. Tools for inspecting the behavior of a CS and analyzing performance of its individual components, when certain conditions are met, are needed. Then, quick and accurate correction of a problem is a must for operational excellence. One way is to set up a knowledge base that identifies common problems and lists their remediation. Unfortunately, the fast development cycle of a CS built using OpenStack makes the setting up of this knowledge base challenging. That is, the knowledge base can soon be out-of-date. To speed up the setup of knowledge base for a new release or feature, one can trace the execution of logical operations in a pre-production setting and perform

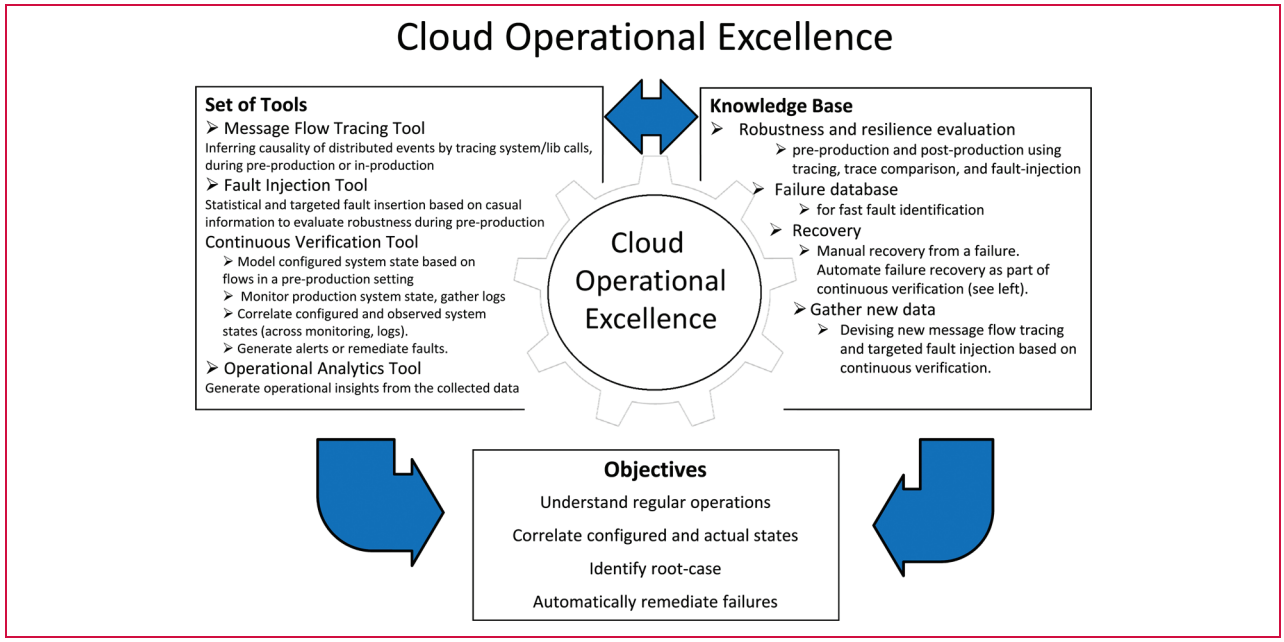

Figure 1

Overarching vision of operational excellence in a cloud system.

large-scale fault injection experiments. The knowledge base can then be populated with the message traces (during normal and faulty operation), injected faults, and logs. During a fault encountered in a production system, the cloud service provider can consult the knowledge base and take remediation action. If the remediation information does not exist, the cloud service provider can augment the knowledge base with the remediation information, in case the problem arises again. The remediation information can also be used to build remediation automation.

Component upgrade and reconfiguration

Operations such as patching, upgrading, and reconfiguration are automatically performed in a CS. These operations must be monitored to ensure their correctness. Tools are needed to expose the concrete steps performed by the automatic upgrading or reconfiguration, and interpret these steps with the cloud service provider- or user-centric semantics. For example, in OpenStack, upgrading a component may involve database changes. Without such tools, software-initiated automatic changes in a CS can easily get out of control.

Operational excellence vision for a cloud system

Figure 1 illustrates our vision for achieving operational excellence in a CS. The core of providing operational excellence in a CS consists of a set of tools and monitors, as well as a knowledge database constructed from these tools, monitors, and operational experiences. The tools and monitors enable a number of essential capabilities for understanding normal operations, diagnosing problems, and recovering from failures. Based on data gathered from these tools and operational experiences, a knowledge base is constructed and augmented for understanding normal operations and for manual recovery. The operations for recovering from a failure can then be automated and new operations or tracing can be identified (hence, the double arrow in Figure 1 between "Set of Tools" and "Knowledge Base").

The "Set of Tools" in the top left box of Figure 1 lists the four basic building blocks for operational excellence in a CS. The "Message Flow Tracing Tools" transparently intercept system and library calls and use the network connection information and thread information available at the system and library calls for inferring causality of distributed events (messages) among the cloud components. This message flow tracing can be performed for logical operations (e.g., creation or deletion of VMs), during pre-production and in-production. The tracing is the fundamental capability for cloud operational excellence because it exposes the step-by-step dissection of multiple components executions and constructs the entire execution flow for requests of interest.

The "Fault Injection Tool" injects faults to evaluate the robustness of a new release of a CS. The fault injection

tool is able to conduct two types of fault injections: i) statistical fault injection and ii) targeted fault injection. The statistical fault injection is required to evaluate the robustness and resilience of a CS and the applications running in it. The targeted fault injection can be used to study specific components of a CS or applications, as well as specific scenarios or use cases. This capability is particularly useful in evaluating the rapidly growing cloud software such as OpenStack, where a new version is released every six months. By targeting the fault injection toward newly added or improved components or operations, their robustness can be quickly evaluated before deploying them into production. The statistical and targeted fault injection techniques are complementary to the test cases run by a developer or a provider.

The results of a comprehensive fault injection and tracing study are used to set up a knowledge base for regular and failed operations (as shown in the top right box of Figure 1). When a failure occurs in production, the cloud service provider or service users can consult the failure database to localize the error by comparing the output, relevant logs, and execution flows with the data in the failure database. After the error is localized, the cloud service providers can then find the problem root cause and perform outage recovery. This procedure can then be automated to perform fast root-cause analysis and recovery. The failure database can also be augmented with the observed failures in production and their remediation.

The "Continuous Verification Tools" gather the observed system states of various components periodically or on-demand and compare them with the configured states. The configured states are typically stored in CS databases or application databases (or their equivalents). Due to the rapidly evolving nature of the CS software, the configured states may change (e.g., new configuration database tables are added in a new release). Examples of the configured states of a logical component such as a VM include its identifier in the system databases, media access control (MAC) address, network address, physical machine on which the VM is configured to run, the directories where the VM disks are stored, and the user request identifiers that create or modify a VM. The tracing tool can be used to quickly construct models of the configured system states for logical components (e.g., a VM or a virtual disk) or logical operations (e.g., VM creation). The model can specify how to gather these configured states from the system using the monitoring tools. The monitoring tools gather the observed states and compare the observed states against the model of the configured states to identify any discrepancy. "Continuous Verification Tools" can be programmed to automatically take a quarantine or remediation action, or generate an alert. A cloud service provider can use the alert information (e.g., error string in the log file) to search the knowledge base and identify possible reasons for the fault. If a possible reason is not found, a cloud service provider can update the knowledge database about the incurred fault and any remediation action. The inferences gained from continuous verification and knowledge database can be used to guide the gathering of new message flows and fault insertion points.

Finally, the "Operational Analytics Tool" analyzes the operation data, including monitoring, alerts, errors, and logs, to generate operational insights for the cloud service provider.

Toward achieving operational excellence in an OpenStack cloud

In this section, we describe a set of techniques for tracing execution across components, building a model of configured system state by leveraging the traced execution, and performing continuous verification in a CS built using OpenStack [3]. The development of fault-injection and operational analytics techniques is ongoing and is not included in this paper.

OpenStack architecture

OpenStack [3] is a distributed IaaS software system. Its components communicate with each other via REST (Representational State Transfer) protocols, SQL (Structured Query Language), and AMQP (Advanced Message Queuing Protocol) [5] to perform cloud operations. OpenStack has six main components: compute (nova), image management (glance), authentication (keystone), networking-as-a-service (quantum), block storage (cinder), and object storage (swift). **Figure 2** depicts the architecture of OpenStack for Grizzly Release (April 2013) [6]. Each component comprises one or more processes (or services) and maintains a database for storing persistent information. The use of the central database is potentially a limiting factor for scalability.

The compute (nova) component comprises an API server (nova-api), a scheduler (nova-scheduler), and a conductor (nova-conductor) that run on an OpenStack controller machine (as shown in Figure 2), as well as compute and network workers (nova-compute and quantum-agent) that run on physical servers hosting VMs. The compute components interact with each other through an asynchronous message-passing server (RabbitMQ** [7] or Qpid** [8]) running advanced message queuing protocol (AMQP). The use of AMQP allows easy scaling of compute controller services and workers (by simply running more copies of them). The persistent state is stored in a database. For detailed architecture of OpenStack, we refer the reader to [9].

In our previous work [10], we demonstrated the complexity of operating OpenStack by tracking the evolution of logical requests such as creating or deleting a VM in a constrained setting. The work demonstrated a significant increase in complexity of operating OpenStack; i.e., the

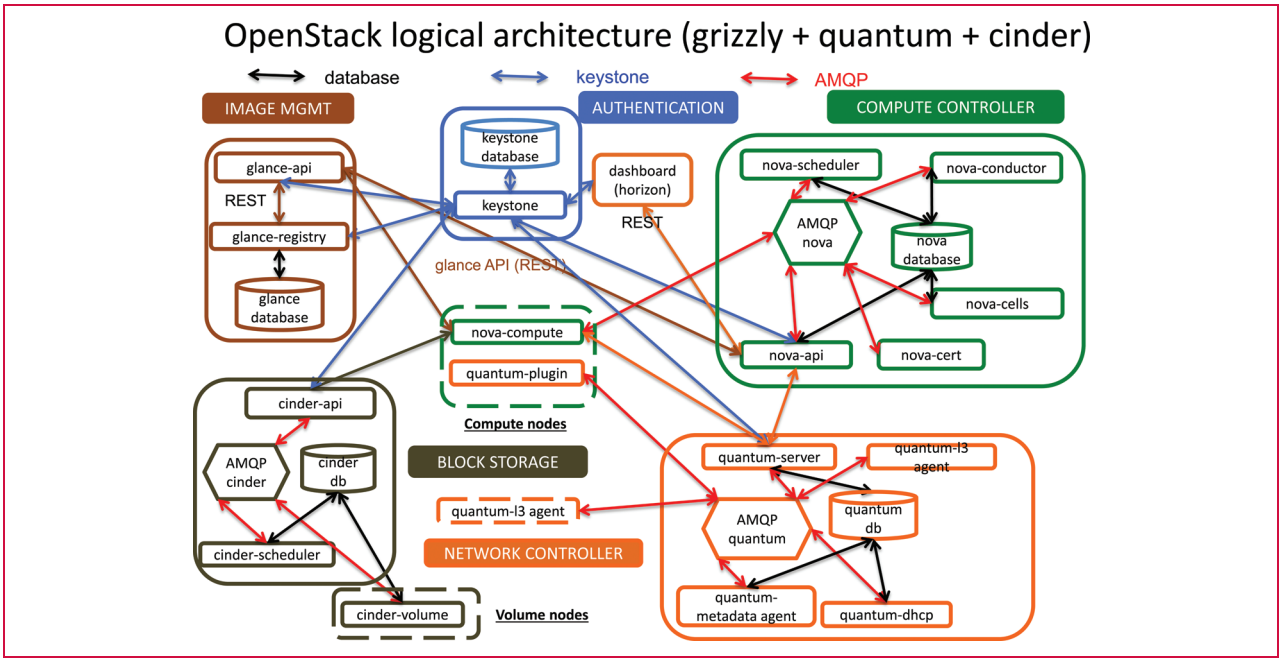

Figure 2

OpenStack logical architecture.

number of SQL INSERT and UPDATE operations for a single VM create has increased from 13 in the Diablo release (September 2011) to 111 in the Grizzly release (April 2013). The rapid increase in the number of SQL operations is largely attributed to the distributed nature of OpenStack, and its maturity from a mere curiosity to enterprise readiness. Without tracing tools, it is difficult to quickly and precisely determine the configured states for a VM. Moreover, without the continuous verification tools, the management of an OpenStack cloud can quickly get out of control.

Tracing

Tracing the causality of events (or messages) within complex distributed applications is a challenging task. The fundamental source of difficulty arises due to the lack of ability to observe the internal behavior of a request handling process among components. Consider a case when a message is received by one component, and another message is generated by the same component shortly after. Can we say that the second message is causally related to the first message? Could it be possible that they are simple unrelated events that happen to occur within a close time interval?

One way to address this difficulty is to instrument the application source code. If we can modify the code so that incoming and outgoing messages of a component carry unique identifiers, then it becomes much easier to trace the message causality. However, it is impractical to assume that source code will be available for all the components of a CS. Even with the availability of source code, understanding the code and identifying the right code segment to instrument is a daunting task. This task is further exacerbated by the rapid development cycles, which may change the casual relationship among messages. Another approach is to employ statistical techniques to make a best-effort guess of causality. For example, one can attempt to discover likely correlations between incoming and outgoing message via time stamps. The problem with this approach is that the accuracy of the tracing result depends very much on the quality of the data collected. In addition, because it is a statistical approach, the output does not guarantee a precise trace.

Our tracing approach alleviates these problems by making the following design decisions. First, we rely on the operating system (OS)-level information in building message traces. We observe various system events such as SEND or RECV system calls (or LIBC calls), thereby avoiding any need for modifying application source code. Second, we monitor these events per thread. From our experience, the granularity of the gathered information must be at the thread-level in order to achieve precise tracing results. If the granularity is higher (i.e., process-level or system-level), the gathered data becomes an aggregation of multiple parallel activities and renders the job of separating these activities significantly difficult.

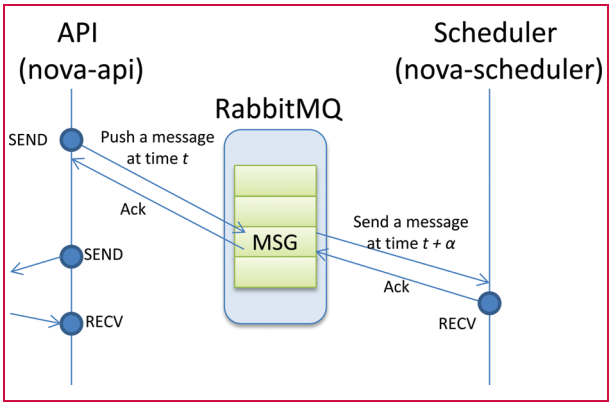

Figure 3

Tracing messages through queues.

Our tracing technique is based on the following principle. When a component receives a message, the message is handed over to one of the worker threads of the component. Then, the thread starts to process the message until the thread generates outgoing messages to another component. This implies that we can discover the relationships between incoming and outgoing messages if we are able to monitor the threads activities. In theory, it is possible that a pair of incoming and outgoing messages handled by the same thread may be semantically unrelated. If that happens, our causality discovery algorithm would still correlate them falsely. However, we assume that such a "corner case" is very rare in practice. Event monitoring necessary for implementing the aforementioned principles can be done at multiple levels, i.e., application-level, kernel-level, and hypervisor-level. In this work, we have employed the application-level solution that works by interposing our custom library in front of the original LIBC shared library using LD_PRELOAD mechanism. In this custom library, we intercept all the I/O-related LIBC calls, and record the invocations of these calls with parameter values as well as the information of the threads making these invocations.

Although our technique for message tracing is an advancement over instrumentation-based and statistical approaches, it does not solve all the tracing problems. One notable challenge that needs to be handled in our technique is the existence of message queues. As shown in Figure 2, OpenStack compute (nova), network (quantum), and block storage (cinder) components make use of a message queue (AMQP) for inter-process communication. When a compute (nova) component inserts a message in the queue, the thread that invokes a SEND to the queue process returns with success immediately after the SEND completes. The message can then be picked up by another compute component from the queue at an arbitrary time in the future. One example case is illustrated in **Figure 3**. The API (nova-api) component pushes a message to the RabbitMQ at time t and continues execution of other tasks. The scheduler (nova-scheduler) component, later, picks up the message at time $(t + \alpha)$ and processes it, where α is an arbitrary time duration that is not determinable a priori. From the thread information, we cannot infer that the message picked up at $t + \alpha$ is the same as the one arriving at t. Therefore, our technique, which tracks the message causality based on the thread information, is unable to track the flow of such messages across queues.

To address this challenge, we employ an application-dependent technique of examining the message contents and extracting any unique identifier(s) used by the components. In the messages exchanged among compute (nova) components, the unique identifier is *request id*. Our technique uses this field to bind the incoming and outgoing messages of the queue into a single message flow from the source component to the target component across the queue. For example, in Figure 3 the message that is pushed to the queue by API (nova-api) and the message that is received by scheduler (nova-scheduler) carry the same value for the *request id* field. For successful tracing involving queues, our tracing technique depends on the unique identifiers exposed in the messages. Without such identifiers, the only other available technique for tracing is a direct comparison of message contents at source and target.

From the trace, we identify the events of interest (e.g., a VM creation request) and identify the root of this event. The tree rooted at this event can be traversed to discover all the sub-operations for this event.

Comparing trace flows

The analysis tools for a CS must be able to compare the traced execution of logical operations under consideration (e.g., VM creation). In this paper, we use tracing tool in a non-production setting. Evaluating the feasibility of this tool in a production system is part of ongoing work. The comparison of traced execution can be used for studying of features in new releases of software, as well as an in-depth understanding of CS's normal behavior. Specifically, using our analysis tools, we compare the traced execution by translating the flows under consideration into strings, and then computing the edit distance between the two strings. Here we describe the procedure.

Recall from Figure 2, that OpenStack has multiple components, and each component has one or more services (or processes). We denote each OpenStack service (or process) by a letter and encode the message flow among them during processing of a request as a string of the letters. For example, the execution flow of a VM provisioning in the Diablo release of OpenStack (September 2011) starts as follows: client (E) → API server (nova-api) (F) → Authentication server (keystone) (D) → Database (mysql)

Table 1 Comparison of execution flows for the virtual machine-provisioning operation in different OpenStack releases. The lengths of the strings translated from the longest paths of the execution flows are denoted by the parenthesis, and the edit distances of these strings are listed in the table.

	Diablo (2,146)	*Essex (867)*	*Folsom-nova-network (2,872)*	*Folsom-quantum (3,741)*	*Grizzly (March 28) nova-network (1,676)*	*Grizzly nova-network (1,731)*
Diablo (2,146)	0	1,457	838	1,741	1,284	1,234
Essex (867)	1,457	0	2,045	3,020	974	1,030
Folsom nova-network (2,872)	838	2,045	0	1,107	1,961	1,927
Folsom-quantum (3,741)	1,741	3,020	1,107	0	2,787	2,747
Grizzly (March 28) nova-network (1,676)	1,284	974	1,961	2,787	0	114
Grizzly nova-network (1,731)	1,234	1,030	1,927	2,747	114	0

(C) → Authentication server (keystone) (D) → −. (Here, the—symbol indicates the external process generating the request.) The letters in the parenthesis denote the corresponding services before the parenthesis. The arrow indicates the direction in which the processing of the request traverses as identified by our tracing tool. Thus, the result string out of this translation is EFDCD . . . .

The edit distance between two strings is a classical concept, which is defined as the minimum number of operations (insertion, deletion, or replacement of a letter) for changing one string into the other. For example, the edit distance between "bke" and "abcde" is 3 (insertion of "a", insertion of "c" and replacement of "k" with "d"). Dynamic programming is employed to compute the edit distance. As the algorithm for computing the edit distance is quite classical, the functionality of computing the edit distance is available in common libraries (we use the edit distance library in Python programming language).

One particular challenge in the comparison of two execution flows arises due to message fragmentation. Due to the Transmission Control Protocol (TCP) stack implementation, a message may be fragmented unpredictably, and the fragments are visible at the LIBC call level. The fragmentation is usually non-deterministic, i.e., a message may be cut into, say, four fragments and be observed by the tracing tool as four back-to-back messages. We process the message trace to deal with the message fragmentation; i.e., all consecutive SEND or RECV events that have identical source and destination addresses (including the ports information) should be grouped into a single event.

The message flow constructs a tree rooted at the start of an event of interest. To compare two different message flows for the same event of interest across releases, we calculate the edit distance for the longest path for root to leaf node in two message flow trees. From our analysis, we found that the length of the longest path overwhelmingly dwarfs the length of other possible traversable paths.

Table 1 shows the comparison results for the longest path in the execution flow of a VM creation operation in the last four releases of OpenStack. For the Folsom release (September 2012), we show the results under two different system configurations (using Quantum, an implementation of a subset of software-defined network features, and nova-network, the implementation before Quantum). For the Grizzly release (March 2013), we also show the results for a March 2013 build of the code. The lengths of the strings translated from the longest path are shown in parenthesis. The edit distances of the strings are listed in the table.

The table shows that there are large changes of execution flows between major releases of OpenStack or configurations, while the changes between a release and its recent build before the final release are small. For example, Folsom (nova-network) and Grizzly (March 28 nova-network) (the March 28, 2013, build of the Grizzly release) have an edit distance of 1,961 for the execution flow of a VM creation operation. For the two strings with the lengths of approximately 3,000, the edit distance of 1,961 is quite large (about $1{,}961/3{,}000 = 65\%$ change). However, Grizzly (March 28 nova-network, the March 2013 build of Grizzly) and Grizzly (nova-network), the April 2013 release of Grizzly, have only about $114/1{,}700 = 6.7\%$ change. Through the identified changes, we were quickly able to identify the configured states for resources such as a VM or a block-storage device, which we then used to construct a configured state model for continuous verification.

In summary, our techniques of tracing and flow comparison allow us to automatically construct flows for a logical operation under different configurations and perform a quick comparison, which helps a cloud service provider to rapidly consume an ever-changing open source platform such as OpenStack.

Continuous verification

To ensure operational correctness, it is imperative to perform continuous verification of soft-states across CS components, as they undergo rapid and automatic changes to meet on-demand requests. At a high level, the idea is to build a tool that compares the configured states of a CS (or its components) with the observed states gathered using monitoring tools and logs, and identify any discrepancies for the cloud service provider, who can then manually or automatically take actions to address the discrepancies.

There are several challenges in performing continuous verification of a CS. First, how are the configured states for a CS component or the entire CS identified? Second, how does the verification tool determine which states to gather and when? Third, how does the verification tool identify the discrepancies between configured and observed states?

Identifying configured states for a CS or its components or a resource provisioned in a CS is challenging. First, a CS or its components may be deployed and configured with automation through tools such as Chef [11] or Puppet [12], or an orchestrator that drives these tools. The configured state partially resides in these tools, and is also configured directly on the end-points. Once the system is configured, it is ready to deliver compute, network, and storage as software.

During the operation of a CS, the configured state of a newly created resource in a CS may be distributed across multiple components of a CS. For example, in a CS built using OpenStack (see Figure 2 for OpenStack architecture), there are compute, network, and storage components, each having their own databases. Upon a resource provisioning (e.g., a VM), the configured state is stored in multiple tables across these component databases, making the identification of configured states challenging. Further, the compute, network, and storage components in turn interact with other non-OpenStack components such as Open vSwitch [4] and libvirt [13], who maintain their own configured states. The task of determining the exact configured states is exacerbated by the fact that OpenStack can run in multiple configurations, and is continuously evolving (a new version is released every six months). Moreover, the permutation of resource creations may alter the set of configured states.

In this paper, we propose a model-driven approach for specifying the configured states of CS resources such as VMs and block storage devices. The CS is built using OpenStack. The configuration settings of OpenStack in this pre-production setting reflect a production deployment. We construct the model for the configured states by running logical operations such as creating a VM using our tracing tool in a pre-production setting. The number of logical operations to be run under our tracing tool is pre-determined and reflects the operations available to the user of a CS. Example of these operations in an IaaS CS include creating a VM, deleting a VM, creating a SDN, deleting a SDN, creating a block-storage device, attaching VM to a block storage device, and removing block-storage device from a VM. For each logical operation, the gathered trace includes interaction among all components, including component databases, Open vSwitch, etc. From the trace data, we create a model of the configured states for a resource from the flow data. The creation of model is currently manual and requires OpenStack expertise. The idea is that OpenStack experts will create the model once, which can be used by a support team to identify problems, or collect information about identified problems before engaging the next level of support team. The model represents configured states across OpenStack component databases, and other dependent components. The model identifies the components to query for gathering the configured state. Each identified configured state is augmented with the information for collecting the actual value for this state. The model is specified in the following format:

$$\text{VARIABLE} \leftarrow \text{CONFIGUREDSTATE}| \; [\text{OBSERVEDSTATE}[\rightarrow \text{ACTION}]]^*,$$

where "VARIABLE" is an attribute or a field of interest for a cloud service provider. It can be a single value or a list. The "CONFIGUREDSTATE" and "OBSERVEDSTATE" specify the configured and observed state respectively. The configured state or observed state can be specified as a single value or a list of strings, or computed dynamically by specifying keywords that invoke tools or query components. Borrowing from the regular expression notation, we can specify multiple keywords for OBSERVEDSTATE. An optional parameter for each OBSERVEDSTATE keyword is ACTION, which specifies the action to take if there is match or mismatch observed between the configured and observed state. The default action is do-nothing. All keywords have one or more parameters, which are specified in square brackets.

Presently, we define the following keywords:

- SQL: Issues an SQL query to gather the configured state.
- GET: Obtains values from a component.
- LOG: Gathers all log entries that contain the VARIABLE, where a VARIABLE is a single value or a logical conjunction or disjunction of parameters.
- EXISTS: Checks if a file or a directory exists on the destination machine.
- DIFF: Runs a COMMAND on the destination machine and compares output with the configured state.

- RUNNING: Runs a COMMAND on a machine to check if the specified programs are running.

The configuration of the CS (e.g., where is the component database server located, which service is running where, where are logs stored, and which tool to use for searching logs) and the model is fed to a verifier program (VP) that invokes the rules and generates output for every rule. The observed states can be gathered periodically or on-demand using monitoring tools such as Ganglia [14] and notifications generated using tools such as Nagios [6]. In our proof-of-concept, the output is gathered on-demand, i.e., when the VP invokes model. If the observed state is already collected using Ganglia or Nagios, the VP can be updated to run model on the already collected observed states. If there is a mismatch observed between configured and observed states, and an ALERT keyword is specified, the VP generates an output string with an ALERT prefix. If a LOG keyword is used, the VP dumps entries across all log files that contain the VARIABLE. The per-resource output from VP can be stored in a NoSQL database.

Below, we give an example usage of subset of these rules for an "instance" resource in a CS build using the OpenStack Grizzly [15] release. Using the tracing tool, it was easy to build configuration models for multiple releases of OpenStack running in different configurations. Currently our continuous verification tool only generates alerts based on identified discrepancies. The VARIABLE is always passed as a parameter to the keywords for observed states but is not shown for brevity, unless exposition is not clear.

Comparing host and instance lists

InstanceList ← SQL[SELECT nova.instances.uuid from nova.instances WHERE host = '$PhysicalHost' and deleted_at IS NOT NULL;] | DIFF[COMMAND, $InstancePhysicalHost)] → ALERT

(where PhysicalHost is specified by the user. Checks if the list of instances running on a physical server in the configuration databases match with the list of instances actually running on the physical server. The physical server is specified by the user.)

Instances

InstanceId ← Instance uuid (supplied by the user) | RUNNING [COMMAND,$InstancePhysicalHost] → ALERT
(This rule stores the instance-id of a VM instance under consideration and uses virsh to check if an instance exists and is running. If an instance is not found or not running, an alert is generated.)

InstanceId ← Instance uuid (supplied by the user) | LOG [$InstanceId]
(This rule finds all log entries that contain this instance-id.)

InstancePhysicalHost ← SQL [SELECT host from nova. instances WHERE nova.instances.uuid = '$InstanceId']|
(This rule stores the configured physical host for this VM instance.)

InstanceDiskPath ← /var/lib/nova/instances/$InstanceId| EXISTS [$InstancePhysicalHost] → ALERT
(This rule generates an alert if an instance disk path is not found on the physical server.)

Request identifier

Unique_Request_Id ← SQL
[SELECT nova.instances_actions.request id WHERE ($InstanceId == nova::instance_actions::instance uuid and DB::nova::instances actions :: action == 'create'] | LOG [$Unique_Request_Id]
(This rule stores the configured request identifier for a VM creation from compute database. Further, it finds all log entries with this identifier.)

Storage

ImageId ← SQL[SELECT nova.instances.image WHERE nova.instances.id == $InstanceId] | EXISTS [$InstancePhysicalHost, $InstanceDiskPath/$Image_id)] → ALERT, LOG

RamdiskId ← SQL[SELECT nova.instances.ramdisk_id WHERE nova.instances.id == $InstanceId)] | EXISTS [$InstancePhysicalHost, $InstanceDiskPath/$Ramdisk_id)] → ALERT, LOG

KernelId ← SQL[SELECT nova.instances.kernel_id WHERE nova.instances.id == $InstanceId] | EXISTS [$InstancePhysicalHost, $InstanceDiskPath/$Kernel_id] → ALERT, LOG

(These rules find the kernel, ramdisk, and image identifier in configuration databases, check if they exist on the physical host, and find logs with these ids.)

Network

MacAddress ← SQL[SELECT network.ports.mac_address (WHERE $InstanceId == network.ports.device_id)] | DIFFER[COMMAND, $InstancePhysicalHost)] → ALERT, LOG

NetworkId ← SQL[network.ports.network_id (WHERE $InstanceId == network.ports.device_id)] |

SubnetId ← SQL [network.ipallocations.subnet_id

(WHERE $NetworkId ==
network.ipallocations.networkid)] |
TunnelId ← SQL [network.ovs_network_bindings.
segmentation_id (WHERE $NetworkId ==
network.ovs_network_bindings.segmentation id)] |

(These rules store the mac address of a VM, network and subnet identifier, and GRE tunnel ids to which this VM is connected from configuration databases. Further, the first rule in "Network" gathers the actual values for mac address from $InstancePhysicalHost)

As part of ongoing work, we will be deploying this tool in a production setting to perform continuous verification and quickly diagnose problems.

Related work

The problem of runtime tracing of distributed applications has received much attention. One approach is to apply statistical techniques to infer the causality of messages [16–17]. The goal of these approaches is to understand the overall runtime behaviors or to diagnose performance problems. Another approach is to instrument the source code to insert message identifiers [18–20] into logs. The identifiers in the logs provide accurate information about the message flows and hence, they allow users to look at the individual message flows to pinpoint anomalous behaviors. However, it is considered impractical to assume the availability of source code for all the components in a distributed system. Even with the availability of source code, it is difficult and time-consuming to understand the code to find the right location of instrumentation. There is also an approach of runtime tracing, vPath [21], that gleans information from the underlying OS kernel or even the hypervisor and, with the assumption of application behavioral model, reconstructs the message flows accurately without instrumenting the application code. We found that the benefits brought by the vPath technique aligned well with our practical goals and, therefore, we adopted the principles of vPath in building the tracing mechanism. However, unlike the vPath's hypervisor-level tracing, we have developed a user-level solution for intercepting the LIBC library calls.

There are numerous public cloud offerings in the industry such as Amazon EC2** [22], Windows Azure [23], and GoGrid [24]. They all run their own proprietary version of cloud management software and, unfortunately, little is known about their specifications. For this reason, there has been a motivation for developing an open version of cloud management platform software that is lightweight and easy to experiment with. Eucalyptus [25], Nimbus [26, 27], Open Cirrus [28], OpenNebula [29], and OpenStack [3] are examples of such open source CSs.

Conclusion

A CS comprises many components distributed across multiple layers. It rapidly and automatically delivers compute, network, and storage as software. Further, a CS also delivers the upper layer management functions such as software updates though full automation driven by policies. In this paper, we proposed a set of capabilities for understanding the behavior of fully automated operations across CS, and identifying and fixing problems as soon as they occur. We refer to the task of building such capabilities as operational excellence. We highlight the challenges of achieving operational excellence in a CS, and state our vision. These challenges include multitude of configuration options, layers of software-defined automation, and rapid development cycles (e.g., every six months for OpenStack). Toward that vision, we have presented tracing and continuous verification techniques in order to quickly gain insights into the operations of a CS built using OpenStack. Our tracing techniques allow determining full execution flows for a logical operation (e.g., a VM creation) and compare the flows across releases and configurations, while our configuration model built using tracing flows provide a mechanism for comparing configured and actual states across components and layers.

As part of ongoing work, we are improving the configuration model specification, and building fully automated tools for correlating the configured state with the monitoring data in a production IaaS cloud. As such, tracing, modeling of configured states, continuous verification of runtime states, and robustness analysis and problem diagnosis through fault injection are critical in achieving operational excellence in a CS.

References

1. B. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Commun. Surveys Tuts.* [Online]. Available: http://hal.inria.fr/docs/00/87/76/27/PDF/hal-survey-v3.pdf
2. IBM SmartCloud Enterprise+. [Online]. Available: http://www-935.ibm.com/services/us/en/managed-cloud-hosting/
3. OpenStack. [Online]. Available: http://www.openstack.org/
4. Open vSwitch. [Online]. Available: http://openvswitch.org/
5. Advanced Message Queuing Protocol (AMQP). [Online]. Available: http://amqp.org/
6. Nagios. [Online]. Available: http://www.nagios.org/
7. RabbitMQ. [Online]. Available: http://www.rabbitmq.com/
8. Apache Qpid. [Online]. Available: http://qpid.apache.org/
9. OpenStack Architecture. [Online]. Available: http://docs.openstack.org/admin-guide-cloud/content/conceptual-architecture.html

10. S. A. Baset, C. Tang, B. C. Tak, and L. Wang, "Dissecting open source cloud evolution: An OpenStack case study," in *Proc. 5th USENIX Workshop HotCloud Topics Comput.*, San Jose, CA, USA, Mar. 2013, pp. 1–6.
11. Opscode Chef tool. [Online]. Available: http://www.opscode.com/chef/
12. Puppet tool. [Online]. Available: https://puppetlabs.com/
13. libvirt: The Virtualization API. [Online]. Available: http://libvirt.org/
14. Ganglia Monitoring System. [Online]. Available: http://ganglia.sourceforge.net/
15. OpenStack Grizzly Release. [Online]. Available: http://www.openstack.org/software/grizzly/
16. M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proc. 19th ACM SOSP*, New York, NY, USA, 2003, pp. 74–89.
17. A. Anandkumar, C. Bisdikian, and D. Agrawal, "Tracking in a spaghetti bowl: Monitoring transactions using footprints," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, New York, NY, USA, 2008, pp. 133–144.
18. P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modeling," in *Proc. 6th Conf. Symp. OSDI*, Berkeley, CA, USA, 2004, vol. 6, p. 18.
19. K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang, "Hardware counter driven on-the-fly request signatures," in *Proc. 13th Int. Conf. ASPLOS*, New York, NY, USA, 2008, pp. 189–200.
20. B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Mountain View, CA, USA, Tech. Rep. Dapper-2010-1. [Online]. Available: http://research.google.com/pubs/pub36356.html
21. B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vPath: Precise discovery of request processing paths from black-box observations of thread and network activities," in *Proc. Conf. USENIX Annu. Tech. Conf.d*, Berkeley, CA, USA, 2009, p. 19.
22. Amazon Elastic Compute Cloud. [Online]. Available: http://aws.amazon.com/ec2/
23. Microsoft Azure. [Online]. Available: http://www.microsoft.com/azure/
24. GoGrid cloud hosting. [Online]. Available: http://www.gogrid.com
25. Eucalyptus. [Online]. Available: http://www.eucalyptus.com/
26. K. Keahey, I. Foster, T. Freeman, and X. Zhang, "Virtual workspaces: Achieving quality of service and quality of life in the grid," *Sci. Program.*, vol. 13, no. 4, pp. 265–275, Oct. 2005.
27. Nimbus. [Online]. Available: http://www.nimbusproject.org/
28. A. I. Avetisyan, R. Campbell, I. Gupta, M. T. Heath, S. Y. Ko, G. R. Ganger, M. A. Kozuch, D. OHallaron, M. Kunze, T. T. Kwan, K. Lai, M. Lyons, D. S. Milojicic, H. Y. Lee, Y. C. Soh, N. K. Ming, J.-Y. Luke, and H. Namgoong, "Open Cirrus: A global cloud computing testbed," *Computer*, vol. 43, no. 4, pp. 35–43, Apr. 2010.
29. OpenNebula. [Online]. Available: http://opennebula.org/

Received September 5, 2013; accepted for publication October 1, 2013

Salman A. Baset *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (sabaset@us.ibm.com).* Dr. Salman is a Research Staff Member at the IBM T. J. Watson Research Center. He received a B.S. degree in computer system engineering from GIK Institute of Engineering Sciences and Technology, Pakistan, in 2001, and M.S. and Ph.D. degrees in computer science from Columbia University, in 2004 and 2011, respectively. His current research at IBM is focused on tracing and fault injection in distributed systems, particularly OpenStack, DevOps, over-subscription of physical machine resources, automation of software and operating system updates in the cloud, and cloud service-level agreements. He has been elected as the Release Manager of the Standard Performance Evaluation Corporation (SPEC) Open Systems Group (OSG) cloud benchmarking subcommittee, which is working on standardizing a cloud benchmark. He is also a coauthor of a cloud benchmark framework report published by SPEC. Previously, he was involved in the Internet Engineering Task Force (IETF) and is a coauthor of RELOAD protocol for building peer-to-peer communication systems. He is a recipient of the Young Scholars Award by the Marconi Society in 2008 and a best paper award at IPTCOMM (Principles, Systems, and Applications of IP Telecommunications) in 2010. Currently, he is the chair of Distributed and Fault Tolerant Computing (DFTC) Professional Interest Community (PIC) at IBM.

Long Wang *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (wanglo@us.ibm.com).* Dr. Wang is a Research Staff Member the IBM T. J. Watson Research Center. He obtained a Ph.D. degree from the Department of Electrical and Computer Engineering at the University of Illinois at Urbana–Champaign in 2010. Before that, he received a B.S. degree from the Department of Computer Science at Peking University in 2000 and an M.S. degree from the Department of Computer Science at the University of Illinois at Urbana–Champaign. His research interests include fault tolerance and reliability of systems and applications, dependable and secure systems, distributed systems, cloud computing, operating systems, and system modeling, as well as measurement and assessment.

Byung Chul Tak *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (btak@us.ibm.com).* Dr. Tak is a Research Staff Member at the IBM T. J. Watson Research Center. He received his Ph.D. degree in computer science in 2012 from Pennsylvania State University. He received his M.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2003, and his B.S. degree from Yonsei University in 2000. Prior to joining Penn State, he worked as a researcher in the Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea. His research interests include virtualization, operating systems, and cloud computing.

Cuong Pham *University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA (phammanhcuong@gmail.com).* Mr. Pham is a third-year Ph.D. student in the computer science department at University of Illinois at Urbana–Champaign.

Chunqiang Tang *Facebook, Menlo Park, CA 94025 USA (tangchq@gmail.com).* Dr. Tang is presently affiliated with Facebook. Previously Dr. Tang was a Research Staff Member at the IBM T. J. Watson Research Center. He received a Ph.D. degree in computer science from the University of Rochester in 2004. His research interests lie primarily in the systems area (including distributed systems, operating systems, computer networks, and storage systems) and secondarily in novel applications of information retrieval (including consumer-centric medical informatics and peer-to-peer information retrieval). He holds 30 patents, has published 53 papers, and won two Best Paper Awards. His research has also resulted in multiple systems that are now being sold as part of IBM's advanced commercial products. Because of the real-world impact of his research, he won 12 IBM awards. He served IBM's Research community as the Chair of the Services Computing PIC (Professional Interest Community), where PICs are IBM's internal counterparts of ACM Special Interest Groups (SIGs).