Towards a Fault-Resilient Cloud Management Stack

Xiaoen Ju Livio Soares† Kang G. Shin Kyung Dong Ryu†
University of Michigan *IBM T.J. Watson Research Center*†

Abstract

Cloud management stacks have become a new important layer in cloud computing infrastructure, simplifying the configuration and management of cloud computing environments. As the resource manager and controller of an entire cloud, a cloud management stack has significant impact on the fault-resilience of a cloud platform. However, our preliminary study on the fault-resilience of OpenStack—an open source state-of-the-art cloud management stack—shows that such an emerging software stack needs to be better designed and tested in order to serve as a building block for fault-resilient cloud environments. We discuss the issues identified by our fault-injection tool and make suggestions on how to strengthen cloud management stacks.

1 Introduction

Cloud computing has become an increasingly popular computing platform. With this increasing popularity comes the issue of how to efficiently manage such a new computing infrastructure. Cloud management stacks have been introduced to address this challenge, facilitating the formation and management of cloud platforms.

Since cloud platforms usually consist of hundreds or thousands of commodity servers, failures are the norm instead of the exception [1, 3]. Cloud platforms, as well as applications running in the cloud, need to be designed to tolerate failures induced by software bugs, hardware defects and human errors. The scope of failures varies from one single process to an entire data center, with a timescale ranging from subseconds to days. The same requirement on fault-resilience also applies to cloud management stacks, which themselves run in the cloud environment. Unfortunately, to the best of our knowledge, the fault-resilience of cloud management stacks has not received enough attention from the cloud computing community. Despite the widespread awareness of the prevalence of failures, the community is still largely focusing on how to enhance the functionality of this emerging software stack.

We argue that fault-resilience should remain one of the most important features of any software designed for the cloud environment. Lack of fault-resilience significantly weakens the usefulness of cloud-scale software. We believe that fault-resilience is so fundamental that it needs to be deeply integrated into the design of cloud-scale software instead of being treated as an optional feature or an afterthought. This holds for cloud applications and, more importantly, for cloud management stacks, due to the latter's significant impact on cloud platforms.

In this paper, we explore the fault-resilience of OpenStack, a popular open source cloud management stack that has been drawing significant attention in recent years. By reporting the issues discovered in our study as well as making associated design recommendations and suggestions, we would like to re-stress the importance of fault-resilience for cloud management stacks and call for a collaborative effort in addressing OpenStack's fault-resilience issues.

2 Methodology

OpenStack is a high-level cloud operating system that builds and manages cloud platforms running on top of commodity hardware. It functions via the coordination and cooperation of its various service groups, such as compute services for provisioning and managing virtual machines (VMs) inside a cloud, image services for managing template VM images, and an identity service for authentication. Two major communication mechanisms are employed in OpenStack, with the Advanced Message Queuing Protocol (AMQP) for communications within the compute service group, and the Representational State Transfer (REST) mechanism for communications among other services, as well as with external users. OpenStack also relies on many external services, such

as local hypervisors and databases, and communicates with them using libraries that fulfill their communication specifications, such as libvirt for hypervisors and SQLAlchemy for database access.

We use fault injection to study the fault-resilience of OpenStack. We define fault-resilience to be the ability to maintain correct functionalities under faults. Specifically, we study the impact of faults on OpenStack's external API interface and its persistent states. These two aspects are important because OpenStack relies on the persistent states (e.g., states in its databases) to manage a cloud platform (i.e., which VMs are currently running at which physical hosts), and OpenStack's external API is the interface for communicating with users. A well-designed fault-resilient cloud management stack should maintain correct information in its persistent states and generate correct responses to users' requests via the external API in a timely manner, despite faults.

We use three fault types in our study: server crashes, transient server non-responsiveness and network partitions. These fault types, albeit simple, are common cases that lead to failure situations in a cloud environment. We inject faults into OpenStack at the locations where two services communicate. Take server crash faults as an example. In our single-fault-injection experiments, for a message sent from one service to another, we inject a crash fault to the sender service in one experiment, before the message is sent, and a crash fault to the receiver service in another experiment, before the message is received. We believe that our message-flow-driven fault injection can effectively reveal fault-resilience issues, because (1) message flows indicate the execution paths inside OpenStack for request processing, (2) heinous bugs affecting the fault-resilience of cloud systems such as OpenStack are usually related to the interaction of several services, which can be characterized by their message flows, and (3) bugs and issues within a single service can be located and resolved by mature single-process debugging techniques.

Faults are injected to three OpenStack services—authentication, image and compute services—as well as external supporting services such as database services, local hypervisor managers and AMQP brokers. We target at these services because they are indispensable for running OpenStack. We consider extending the coverage to other services (e.g., object store) our future work.

Our fault-injection tool consists of three components: a logging and synchronization module, a fault-injection module and a specification checking module. The logging and synchronization module logs messages among OpenStack services and coordinates the execution of OpenStack and the fault-injection tool via logging events (e.g., packet send/receive). The fault-injection module injects faults into OpenStack along message flows captured by the logging module. The specification checking module verifies whether OpenStack, with injected faults, complies with specifications related to its expected persistent states and its responses to users.

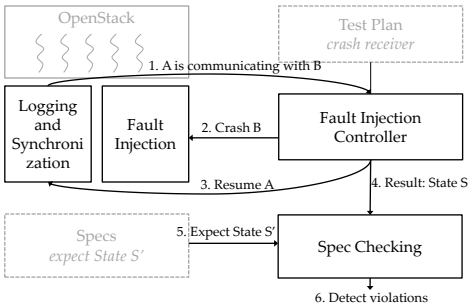

Figure 1: Fault-injection workflow example. 1: Detect fault-injection opportunity and pause execution. 2: Inject fault. 3: Resume execution. 4: Report results. 5: Check specifications. 6: Detect violations.

We focus on the fault-resilience of OpenStack during the processing of the most commonly used external requests, such as VM creation and deletion. For each request, we first execute it and record the message flows among OpenStack services. We then generate an execution graph from the message logs, which characterizes the execution path of OpenStack during the request processing. A node in the graph represents a communication event, recording the communicating entity (e.g., an API server process of the compute service) and the type of communication (e.g., an outgoing RPC message). An edge in the graph connects the sender and the receiver of a communication event. Based on the execution graph and a predefined fault-specification[1], we generate a collection of test plans, each consisting of a fault to be injected into a certain location in the execution graph. We then conduct fault-injection by initializing OpenStack into the same state as in the logging procedure, replaying the external request and injecting faults according to the test plans. After the tests complete, we collect results (both user observable responses from OpenStack and its internal persistent states) and verify them with predefined specifications. Upon verification failure, we manually examine OpenStack's implementation to identify bugs causing the invalid results. Figure 1 illustrates the fault injection process.

We use a simplistic experiment setting (similar to OpenStack's default setting) in which all OpenStack and external supporting services have only one instance. Up to three nodes are used in each fault-injection test for

[1] A predefined fault-specification indicates the fault type injected to an execution, e.g., a sender server crash.

Table 1: Summary of Fault-Resilience Issues

Category	Count
Out-of-the-box fault resilience solution	1
Timeout mechanism	2
Cross-layer coordination	1
Supporting library interference	1
Periodic check	5
Miscellaneous programming bugs	3

proper fault isolation.

3 Fault-Resilience Issues in OpenStack

Using the fault-injection tool described in the previous section, we identify several fault-resilience issues in OpenStack (Essex version), which, unless solved properly, lead to undesirable results in the cloud environment managed by OpenStack. Note that we only categorize and discuss the issues uncovered by our tool, which are by no means close to a complete set of fault-resilience issues in OpenStack or other cloud management stacks. Yet based on our own experience with OpenStack deployment as well as what other users have reported, we believe that the following issues, summarized in Table 1, are commonly found in real-world scenarios. It is also worth noting that, although the number of issues discussed in this study seems limited, some of them, such as the lack of timeout in REST communications (cf. Section 3.2) and the state transition deficiency (cf. Section 3.5), are repeatedly manifested with faults injected in common execution paths of OpenStack.

3.1 Out-of-the-Box Fault-Resilience

To the best of our knowledge, OpenStack has not been designed to enable a straightforward out-of-the-box fault-resilience solution via a unified deployment procedure. This may arise from a plausible opinion in the OpenStack community: fault-resilience is not mandatory for all use cases. Given the critical role OpenStack plays in the cloud, however, it would be reasonable to expect that production-level OpenStack deployment requires fault-resilience, which should be easily enabled across all services in the stack. Our experience shows that OpenStack falls short in this aspect.

For example, supporting services, such as the libvirt service, cannot be configured within OpenStack to restart automatically upon crashes. Similarly, the service failover feature—a key ingredient of high availability systems—can only be enabled outside OpenStack, via the support of services such as Pacemaker. Another example is that, when using Qpid as OpenStack's AMQP broker service, third-party modules need to be separately downloaded, compiled and configured in order for the durable queue features—a default Qpid setting inside OpenStack—to function properly. A unified deployment procedure delivered out of the box, aggregating all necessary configuration options relevant to fault-resilience, will significantly improve user experience.

3.2 Timeout Mechanism

Timeout is a common mechanism in distributed systems to prevent the malfunctioning of one component from blocking the rest of the system. One known difficulty in employing this mechanism is the selection of a proper timeout value. A timeout value that is too short leads to unnecessary disruption of system operation due to premature activation of the recovery logic. A timeout value that is too long, on the other hand, delays failure detection, thus negatively affecting the progress of the system.

Being a distributed system itself, OpenStack uses timeout values as a safety net for service coordination. For example, OpenStack associates a timeout value (60 seconds by default) for RPC calls, preventing the caller from indefinitely waiting for a reply. However, our tool shows that there are cases in common execution paths inside OpenStack where critical timeout values remain unspecified. For example, OpenStack uses the WSGI module in the Eventlet library to coordinate its services during the processing of external requests. The WSGI module in turn utilizes the httplib2 library for communication. Probably, due to backward compatibility considerations, httplib2 associates a timeout value only with connection establishment and request sending operations but not with response obtaining operations (e.g., via `HTTPConnection` class's `getresponse` function). As a result, if a network partition occurs between two communicating services, after a service sends a request to the other but before it obtains a response, then the request issuing service will remain blocked indefinitely. In this case, associating a proper timeout value for the response obtaining operation would solve the problem.

As discussed at the beginning of this section, however, timeout selection is no easy task. This problem is exacerbated by the fact that OpenStack relies on many external services, which may have their own timeout settings. Without fine-tuning these external timeout settings, OpenStack may demonstrate unexpected behaviors when faults occur in the external services. For example, if configured to use MySQL as its database service, OpenStack then uses the MySQLdb library to communicate with the database service. In this setting, our tool identifies that if a network partition occurs between an OpenStack service and the MySQL database, after a connection has been established but before the Open-

Stack service can issue a SQL statement execution command, then the OpenStack service may remain blocked for about 930 seconds—a common behavior among applications using the MySQLdb library. OpenStack does not provide a default timeout value specific to its services for SQL statement execution. Admittedly, such a value may be difficult to specify. But relying on the default behavior of supporting libraries may not be the optimal solution for OpenStack services, either.

We recommend the design of innovative approaches to systematic configuration of timeout values scattered in management stacks themselves as well as in supporting libraries and services, thus better orchestrating cloud management stacks in their entirety.

3.3 Cross-Layer Coordination

As described above, OpenStack uses many external services, such as Qpid, libvirt and MySQL, for efficiently achieving a rich set of functionalities. These services usually provide client libraries to facilitate communications between service applicants and providers. The common design pattern in OpenStack is to insert another set of layers of abstraction between OpenStack services and those external services, defining a uniform protocol for each category of services (e.g., database service category, local hypervisor management category and AMQP communication category) and abstracting away the incompatibility among lower-level supporting services (e.g., between SQLite and MySQL). The advantages related to such an encapsulation design are numerous. However, our study indicates that such additional abstraction layers, when implemented without sufficient caution, may introduce subtle bugs. Below we use a concrete example to elaborate this argument.

Consider the communication between OpenStack and Qpid service. Qpid provides a client library for binding to a Qpid service (functioning as an AMQP broker). On top of this library, OpenStack implements its own wrapper library for Qpid, whose functions are in turn invoked by a higher-level AMQP abstraction layer in OpenStack. A configuration option is exposed from the lower-level Qpid library via OpenStack wrappers to system administrators in order to set the upper bound of connection retries from the client side to a Qpid broker. Upon loss of connectivity when an OpenStack service sends an RPC message to a Qpid broker, the client-side Qpid library retries to connect back to the broker until the configured upper bound is reached. The state of the connection is considered temporarily erroneous during the connection retires. Once the retry limit is reached, the Qpid library transits the state of the connection to a permanent erroneous state and stops reconnection. However, this configuration option is not honored by the indirection layers in OpenStack. Specifically, OpenStack's Qpid wrapper keeps polling the state of the connection without differentiating whether the connection resides in a temporary or permanent erroneous state. This behavior is undesirable because the connection maintained by the lower-level Qpid library, once becoming permanently erroneous, will not be set to operational even if the root cause for the loss of connectivity (e.g., a network partition between the OpenStack service using the Qpid library and the Qpid broker) is resolved. The implementation of OpenStack's Qpid wrapper unnecessarily blocks the RPC issuer service in this case.

3.4 Supporting Library Interference

Issues related to the use of external libraries in OpenStack are not confined to the OpenStack-and-external-library boundaries. External libraries can, in reality, interfere with each other in unexpected ways. Notably, OpenStack utilizes the Eventlet and Greenlet libraries for cooperative threading. With those libraries, user-level threads in an OpenStack service are scheduled cooperatively: one thread continues executing without preemption until it reaches a scheduling point defined by the libraries, at which point the control flow is switched to another thread (if such a thread exists). This user-level thread scheduling requires the modification of certain blocking standard Python library calls, such as `sleep`, in order to prevent the caller thread from blocking the entire OpenStack service. The Eventlet library provides standard library function patches to achieve the cooperative scheduling. However, those patches are not 100 percent compatible with the standard library. Their subtle nuances may negatively affect the functionality of other external libraries used in OpenStack.

Consider the client library of Qpid again. Before an OpenStack service can use the connection objects provided by the Qpid client library to communicate with a Qpid broker, a connection pool needs to be instantiated. During this procedure, the Qpid client library internally uses a pipe to synchronize a waiter object which waits for the connection to a Qpid broker to be established and a connection engine object responsible for the connection establishment. The Qpid library implements this conventional consumer/producer synchronization by having the waiter issuing a `select` call on the read end of the pipe and, when the call returns with a file descriptor ready for reading, issuing a `read` call on the file descriptor.

However, our tool shows that this design causes the blocking of an entire OpenStack service if, at the connection pool creation time, no Qpid broker is active. The OpenStack service in question remains blocked even if a Qpid broker becomes active after the pool creation attempt. This bug is caused by the fact that in OpenStack,

the `select` function call is patched by the Eventlet library, and there is an incompatibility issue related to the patched version. The standard library version of `select` returns a non-empty collection of file descriptors in the read file descriptor list only if they are ready for reading. The Eventlet implementation, however, returns such a non-empty collection if the file descriptors are ready for reading or if there are exceptional conditions associated with them (such as the occurrence of a `POLLERR` or a `POLLHUP` event, due to the fact that `select` is implemented by `poll` in Eventlet). This nuance makes it possible for the Qpid library to read a not-yet-ready file descriptor after the return of a preceding `select` call, which in turn causes the blocking.

This issue, as well as the one in Section 3.3, clearly indicates that examining the fault-resilience of one module in an isolated environment is insufficient for guaranteeing the overall fault-resilience. We would thus suggest that cloud management stack developers conduct thorough testing on the overall fault-resilience of the stacks, with a focus on the compatibility of external libraries.

3.5 Periodic Check

Periodic checks are widely used in OpenStack, mainly for monitoring the well-being of various services. Our study suggests that this mechanism be extended to other aspects of OpenStack as well.

One such aspect is the persistent states of the cloud environment maintained by OpenStack. For example, our study shows that during the processing of a VM creation request, if the scheduler or the compute service crashes or restarts, then the created VM may remain in the transient `BUILD` state. This is an undesirable behavior because a created VM should enter a stable state—`ACTIVE` if the creation is successful and `ERROR` otherwise—in a timely manner. Having aborted the VM creation due to service crashes yet marking the VM creation as in progress would confuse users. A periodic service for checking the progress of request processing and resetting related VMs to their according stable states will prove useful in this case.

Similarly, if an RPC cast message related to the deallocation of a fixed IP is lost during the deletion of a VM, the IP remains in the allocated state while its associated VM has been deleted and the lease of that IP address has expired. Such an IP address cannot be reused by other VMs, causing network resource leakage. A periodic check is needed in this case to reap orphan IPs.

The Falcon spy network [2] seems a promising potential enhancement to existing periodic checking mechanisms in OpenStack. Falcon proposes the use of a hierarchical probing architecture in which heartbeat messages from one layer are monitored by the layer underneath it. In a cloud management stack, we may reuse the idea and design a layered spy network, within which the uppermost spy monitors the processing of requests in a service (e.g., the processing of a VM creation request by a given compute service), and the rest spies monitor lower abstraction layers in the management stack.

3.6 Miscellaneous Programming Bugs

Our fault-injection tool has also found bugs which seem to result from occasional inadvertence. For example, one flaw in OpenStack's Qpid wrapper is that, when an exception occurs during an open-connection operation to a Qpid broker, the wrapper constantly retries `open` without first using `close` to reset certain internal states maintained by the connection, causing subsequent calls to fail. Another example is that, when a user invokes an external compute API but the authentication operation fails due to an internal error condition in the authentication service, the compute API service replies a confusing error message to the user, mistakenly attributing the failed authentication to the use of an invalid token instead of reporting the internal error.

4 Conclusions

In this paper, we described our first attempt to address the fault-resilience issues related to the emerging cloud management stacks. With a preliminary fault-injection tool, we studied the fault-resilience of OpenStack, provided in-depth discussions on six categories of fault-resilience issues, and proposed suggestions on how to strengthen this software layer.

Acknowledgment

We thank Dilma Da Silva, David Oppenheimer (our shepherd) and the anonymous reviewers for their valuable suggestions. The work reported in this paper was supported in part by the US Air Force Office of Scientific Research under Grant No. FA9550-10-1-0393.

References

[1] AMAZON. Summary of the Amazon EC2 and Amazon RDS service disruption in the US east region. `http://aws.amazon.com/message/65648/`. Retrieved in March, 2013.

[2] LENERS, J. B., WU, H., HUNG, W.-L., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the falcon spy network. In *SOSP'11*.

[3] MICROSOFT. Summary of Windows Azure service disruption on Feb 29th, 2012. `http://blogs.msdn.com/b/windowsazure/archive/2012/03/09/summary-of-windows-azure-service-disruption-on-feb-29th-2012.aspx`. Retrieved in March, 2013.