Experimental Assessment of Cloud Software Dependability Using Fault Injection

Lena Herscheid, Daniel Richter, Andreas Polze

Experimental Assessment of Cloud Software Dependability using Fault Injection

Lena Herscheid, Daniel Richter and Andreas Polze
{firstname.lastname}@hpi.de
Hasso Plattner Institute, Germany

Abstract. In modern cloud software systems, the complexity arising from feature interaction, geographical distribution, security and configurability requirements increases the likelihood of faults. Additional influencing factors are the impact of different execution environments as well as human operation or configuration errors. Assuming that any non-trivial cloud software system contains faults, *robustness testing* is needed to ensure that such faults are discovered as early as possible, and that the overall service is resilient and fault tolerant. To this end, *fault injection* is a means for disrupting the software in ways that uncover bugs and test the fault tolerance mechanisms. In this paper, we discuss how to experimentally assess software dependability in two steps. First, a model of the software is constructed from different runtime observations and configuration information. Second, this model is used to orchestrate fault injection experiments with the running software system in order to quantify dependability attributes such as service availability. We propose the architecture of a fault injection service within the OpenStack project.

Keywords: fault injection, dependability, distributed systems, cloud systems, availability

1 Introduction

Fault tolerance is an essential dependability requirement especially in distributed and cloud software systems, where components can fail in arbitrary ways, but a high availability or reliability of the service is nevertheless required.

To evaluate the dependability of complex software systems, a sufficiently realistic dependability model of the system needs to be built. Prominent dependability modeling languages such as fault trees or reliability block diagrams originally stem from the hardware domain. Designed for static systems consisting of thoroughly stress-tested hardware components with known error rates, they cannot reflect dynamic software traits [1]. Since the behaviour of cloud software largely depends on interaction, timing, and environment factors, we believe that experimentation and runtime data is needed to construct such a dependability model.

Runtime software fault injection is a technique for artificially introducing faults, or error states, into the application execution in order to uncover bugs. In contrast to compile-time injection, not the source code, but rather the running system, potentially

adfa, p. 125, 2011.

including its execution environment, is modified. Fault injection experiments are used to gain insights into the system's failure behaviour and its resiliency.

We propose to use a fault injection service for experimentally obtaining dependability data from OpenStack[1] applications. We intend to use such data for building more realistic quantitative dependability models from distributed and cloud software, where much of the failure behaviour depends on dynamic and environmental factors. Ultimately, the framework discussed in Sections 0 and 0 will serve to answer the following research questions:

- How does the unavailability of some nodes affect performance?
- How does availability and consistency degrade in the presence of faults? With the CAP theorem [2] in mind, how are availability and consistency related to network partitions?
- Which components suffer most from faults? Are there single points of failure?

2 Relationship to Cloud-based Solutions

In the cloud computing community, the need for resiliency testing of production systems has been acknowledged recently [3]. There has been a paradigm shift -- from trying to avoid failures at all costs to embracing faults as opportunities for making the system more resilient. The rationale behind fault injection testing of deployed software can be summarized as follows [4]:

"It's better to prepare for failures in production and cause them to happen while we are watching, instead of relying on a strategy of hoping the system will behave correctly when we aren't watching."

Even in the unlikely case that the software itself is bug-free, the infrastructure will eventually break, making resilience mechanisms necessary.

Fault injection can be viewed as the software equivalent to hardware stress testing. Based on the assumption that software systems will unavoidably suffer from external as well as internal faults, fault injection is a means for reducing uncertainty: The deployed system is observed under a realistic "worst case" faultload, and it should maintain a desired level of availability, performance, and service quality despite this faultload. Anecdotal evidence suggests that testing recovery mechanisms under faultload targets a relevant class of cloud software outages [5]:

"All these outages began with root events that led to supposedly tolerable failures, however, the broken recovery protocols gave rise to major outages."

Such outages can be avoided by routine fault injection experiments.

The amount of effort put into the fault tolerance and resilience of cloud applications is often determined by Service Level Agreements (SLAs), and the trade-offs between development efforts, costs for redundancy, availability, and consistency are usually application-specific and based on management decisions. Fault injection can also serve as a tool for testing disaster recovery plans, which are necessary to avoid high losses in revenue in the case of failures. Gunavi et al. [5] therefore propose a new kind of (node) "Failure as a Service" (FaaS). This is what we aim at providing.

---

[1] www.openstack.org, 12/24/2014

3 Related Work

Table 1 summarizes related work in the area of distributed software fault injection. [10], [11], [12], [13] discuss tools for injecting faults into distributed systems mainly at the operating system and middleware levels. More recent approaches specific to cloud software are printed bold in table 1. *Chaosmonkey* [6] is a widely used, Java-based tool for Amazon Web Services (AWS) application resilience testing. The approach of injecting faults into deployed systems was first successfully employed by Netflix and later on adapted by other companies offering cloud-based software[2].

Further approaches to providing resiliency testing services have been presented [14] [9] [5] [15] [15]. Their focus mainly lies on hardware fault models, namely network partitions and latency, as well as node crashes. There are also "fault model agnostic", configurable solutions such as [7] [8] [4]. These solutions provide a framework for easily injecting faults, but the fault classes themselves have to be implemented to some extent by the user. Some fault injection testing for OpenStack has also been explored [14].

Table 1. Related work on distributed software fault injection. Work explicitly dealing with cloud platforms is printed bold.

Name	Fault Types	Implementation
ChaosMonkey [6]	node crash, network latency, zone outage	AWS VM termination
Testing OpenStack	node crash, network partition	systemd, iptables
FATE [7]	programmable	instrumentation (AspectJ)
PreFail [8]	programmable, disk failures	based on FATE
AnarchyApe	node crash, network latency, kernel panic, permissions faults	distributed Java scripts
FSaaS for Hadoop [9]	node crash, network latency + partition, packet loss, fork bomb	based on AnarchyApe
Failure as a Service	machine crashes, network failures, disk failures, CPU overload	distributed failure agent scripts
GameDay [4]	application specific	manual
Orchestra [10]	message delay + byzantine incorrectness	message manipulation in the OS protocol stack
Grid-FIT	message delay + byzantine incorrectness	message manipulation in the grid middleware
NFTAPE [11]	bit-flips in registers + memory, communication + I/O errors	custom fault injecting execution environment
FIAT [12]	memory faults	custom fault injecting execution environment

[2] E.g. StackExchange: http://blog.codinghorror.com/working-with-the-chaos-monkey/, 12/23/2014

The toolsets shown in Table 1 can be used with deployed applications with some additional integration effort. Our work is different from these approach in the sense that it aims at providing a standalone OpenStack service, and also targets a broader fault model which includes software faults.

When testing resiliency by means of fault injection, the *representativeness* of the emulated faults should be of major concern. A recent study [16] shows that state-of-the-art fault injectors do not inject faults according to realistic distributions, observed in real-world systems. Our fault injector will therefore inject software faults, based upon anecdotal evidence (e.g., as described in [5]), instead of hardware faults only.

4 Fault Injection as a Service

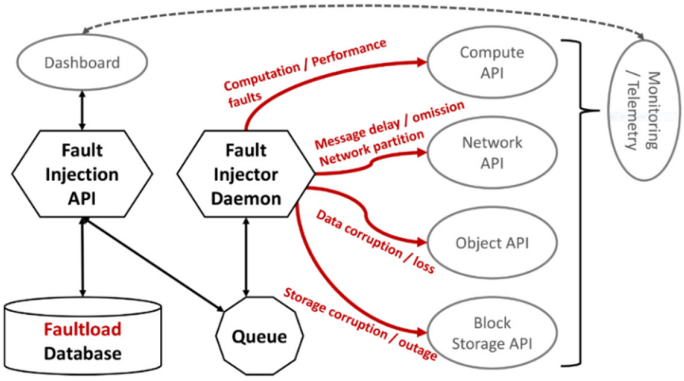

Fig. 1. Proposed architecture for "fault injection as a service". The fault injection daemon accesses other OpenStack service APIs in order to inject different classes of faults. Monitoring and telemetry data should be gathered in order to evaluate how resilient the software is.

Figure 1 depicts the architecture we propose for integrating fault injection as a service into the existing OpenStack ecosystem. Providing an API similar to other OpenStack services will make the tool easy-to-use and accessible. The fault injection daemon, which obtains fault scenarios from a database containing "faultload", should access the REST APIs of other OpenStack services in order to inject faults from different classes.

4.1 OpenStack

OpenStack[3] is a prominent open source "cloud operating system", or *cloud management stack*. Encompassing a number of independent software projects interacting with

[3] www.openstack.org, 12/24/2014

each other through REST APIs, OpenStack is frequently deployed as a modular and configurable Infrastructure as a Service (IaaS) solution for web applications.

With an increasing number of applications built on top of OpenStack, a tool for experimentally assessing the dependability of deployed software is desirable. Therefore, we suggest to integrate fault injection into the OpenStack ecosystem as its own REST service. Our approach is targeted at testing a wide range of software artifacts: OpenStack services themselves, as well as applications deployed on top of OpenStack IaaS, get tested.

4.2 Implementing Fault Classes

Faults from some classes can be injected by merely using API calls in a non-intrusive fashion, others would require the insertion of custom hooks. In the following paragraphs, we discuss the targeted fault classes at different levels in the software stack:

Physical node faults: Since hardware must be assumed to fail at random due to environmental stress or burn-in and wear-out phenomena, this fault class definitely has to be covered by an injection tool. The temporary or permanent outage of physical nodes can be simulated by cutting off messages from those nodes. In the case of compute nodes, API calls may also be used to shut them down. Furthermore, single nodes might suffer from increased CPU utilization (for instance because of other, compute-intensive jobs running on them). This fault class can be implemented either using per-node hooks, or by starting more compute jobs via the API.

Virtualization level faults: Relevant fault classes might be hypervisor service crashes or incorrectness (intrusive), or simply erroneous network configuration. Problems arising from the concurrent hosting of multiple VMs, over-commitment, or violated CPU quotas can be injected at the hypervisor level. Further, environment variables in VMs could be modified to assess their resilience against such robustness threats.

Service level faults: The interaction between different OpenStack services might be malfunctioning. For instance, the API might be used in unexpected or incorrect ways. Such faults can be introduced by the fault injector daemon directly. Exhausted rate limits can be simulated by re-configuring the compute API[4].

Network faults: Anecdotal evidence suggests that the assumption of a reliable, unpartitioned network does not hold in modern distributed systems[5]. To represent such network faults, message loss or delay, as well as partial or total network partitions can be introduced into the networking service.

---

[4] http://docs.openstack.org/trunk/config-reference/content/configuring-compute-API.html, 12/24/2014

[5] https://github.com/aphyr/partitions-post, 12/24/2014

5 Runtime Dependability Models

shows how we intend to use the fault injection service. The vision is a comprehensive framework which uses various sources to automatically run orchestrated fault injection experiments on a deployed cloud software system. From the observed runtime behaviour under faultload, dependability models, including quantitative data, shall be derived automatically.

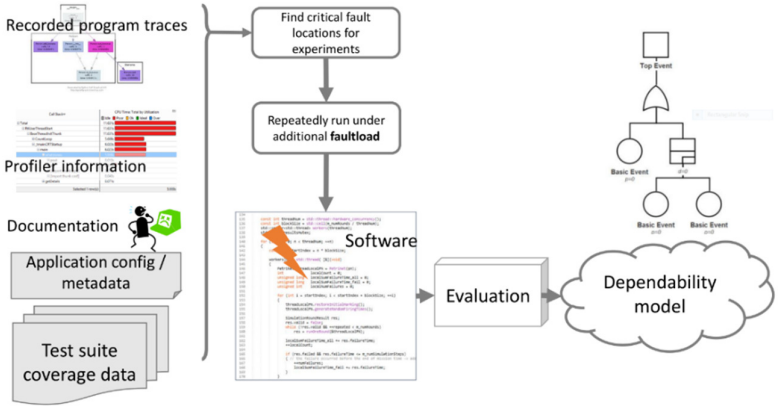

Fig 2. Proposed experimental framework: Fault injection is used as a tool for getting runtime behaviour based dependability models. The faultload is designed according to application-specific information sources, such as profiler data, configuration, and documentation.

A platform-independent fault model, which takes into consideration dynamic information, is needed for cloud software. Such a fault model would answer the question of which faultload is adequate: *Which fault classes should be injected, and how often?*

There is a vast amount of related efforts in the quality assurance domain: literature such as the Orthogonal Defect Classification (ODC) [17], as well as online resources. A recent trend is mining data from bug trackers and using collaborative defect databases such as the Common Weaknesses Enumeration (CWE)[6]. Pretschner et al. [18] propose a fault model tailored for automatic testing. A literature study, with the goal of understanding state of the art software fault models, is currently in progress.

We believe that *fault activation* and *error propagation* patterns are of special interest for a broad class of distributed software failures. Understanding such environment-dependent patterns would enable the design of realistic faultloads and injection locations.

[6] http://cwe.mitre.org/index.html, 12/24/2014

6 Conclusion and Future Work

As we have discussed, fault injection is a promising means for assessing the resilience of cloud software applications. Since the technique is used on deployed software, it yields more realistic failure data than static analysis approaches. We have proposed a draft architecture for "fault injection as a service" within the OpenStack ecosystem. The implementation of the service itself is work in progress.

In order to assess the system's behaviour under faultload, it needs to be observed.

Running fault injection campaigns against a deployed system has the advantage that regular monitoring mechanisms can be used to observe how fault tolerant the system is. In the case of OpenStack, the in-built telemetry service *Ceilometer*[7] provides information on the current state of the system. We still need to integrate monitoring into our architecture. Further approaches might be extracting dependability traits from logs, as demonstrated in [19].

A fault injection service would allow for non-intrusive resiliency tests, and can also be harnessed to experimentally assess the importance of single nodes with regard to overall application dependability. Ultimately, our goal is to use it for obtaining cloud software failure data, so that quantitative dependability models can be applied to make predictions.

References

1. Beizer, B.: Software is Different. Ann. Softw. Eng. 10(1-4), 293-310 (#jan# 2000)
2. Brewer, E.: CAP twelve years later: How the" rules" have changed. Computer 45(2), 23-29 (2012)
3. Tseitlin, A.: The antifragile organization. Communications of the ACM 56(8) (2013)
4. Allspaw, J.: Fault Injection in Production. Queue 10(8), 30:30--30:35 (#aug# 2012)
5. Gunawi, H., Do, T., Hellerstein, J., Stoica, I., Borthakur, D., Robbins, J.: Failure as a service (faas): A cloud service for large-scale, online failure drills. University of California, Berkeley, Berkeley 3 (2011)
6. Netflix: Chaos Monkey. (Accessed 2013) Available at: https://github.com/Netflix/SimianArmy
7. Gunawi, H., Do, T., Joshi, P., Alvaro, P., Hellerstein, J., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Sen, K., Borthakur, D.: FATE and DESTINI: A Framework for Cloud Recovery Testing. In : Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, Berkeley, CA, USA, pp.238-252 (2011)
8. Joshi, P., Gunawi, H., Sen, K.: PREFAIL: A Programmable Tool for Multiple-failure Injection. In : Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, New York, NY, USA, pp.171-188 (2011)
9. Faghri, F., Bazarbayev, S., Overholt, M., Farivar, R., Campbell, R., Sanders, W.: Failure Scenario As a Service (FSaaS) for Hadoop Clusters. In : Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management, New York, NY, USA, pp.5:1--5:6 (2012)

[7] https://github.com/openstack/ceilometer, 12/24/2014

10. Dawson, S., Jahanian, F., Mitton, T.: ORCHESTRA: A Fault Injection Environment for Distributed Systems. Tech. rep., In 26th International Symposium on Fault-Tolerant Computing (FTCS (1996)
11. Stott, D., Floering, B., Burke, D., Kalbarczpk, Z., Iyer, R.: NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In : Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International, pp.91-100 (2000)
12. Segall, Z., Vrsalovic, D., Siewiorek, D., Yaskin, D., Kownacki, J., Barton, J., Dancey, R., Robinson, A., Lin, T.: FIAT-fault injection based automated testing environment. In : Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on, pp.102-107 (June 1988)
13. Looker, N., Xu, J.: Dependability assessment of grid middleware. In : Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on, pp.125-130 (2007)
14. Ju, X., Soares, L., Shin, K., Ryu, K., Da Silva, D.: On Fault Resilience of OpenStack. In : Proceedings of the 4th Annual Symposium on Cloud Computing, New York, NY, USA, pp.2:1--2:16 (2013)
15. Yahoo: AnarchyApe. (Accessed 2012) Available at: https://github.com/yahoo/anarchyape
16. Natella, R., Cotroneo, D., Duraes, J. A., Madeira, H. S.: On Fault Representativeness of Software Fault Injection. Software Engineering, IEEE Transactions on 39(1), 80-96 (Jan 2013)
17. Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., Wong, M.-Y.: Orthogonal defect classification-a concept for in-process measurements. Software Engineering, IEEE Transactions on 18(11), 943-956 (1992)
18. Pretschner, A., Holling, D., Eschbach, R., Gemmar, M.: A generic fault model for quality assurance. In : Model-Driven Engineering Languages and Systems. Springer (2013) 87-103
19. Pecchia, A., Cotroneo, D., Kalbarczyk, Z., Iyer, R. K.: Improving Log-based Field Failure Data Analysis of multi-node computing systems. In : Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on, pp.97-108 (June 2011)
20. Grottke, M., Trivedi, K.: A classification of software faults. Journal of Reliability Engineering Association of Japan 27(7), 425-438 (2005)