Software Fault Injection Campaign Generation for Cloud Infrastructures

Lena Feinbube*, Lukas Pirl*, Peter Tröger[†], Andreas Polze*
* Operating Systems and Middleware Group
Hasso Plattner Institute at University of Potsdam
{firstname.lastname}@hpi.uni-potsdam.de
[†] Operating Systems Group
Chemnitz University of Technology
peter.troeger@informatik.tu-chemnitz.de

***Abstract*—A justifiably trustworthy provisioning of cloud services can only be ensured if reliability, availability, and other dependability attributes are assessed accordingly.**

We present a structured approach for deriving fault injection campaigns from a failure space model of the system. Fault injection experiments are selected based on criteria of coverage, efficiency and maximality of the faultload. The resulting campaign is enacted automatically and shows the performance impact of the tested worst case non-failure scenarios.

We demonstrate the feasibility of our approach with a fault tolerant deployment of an OpenStack cloud infrastructure.

***Index Terms*—fault injection, dependability modelling, testing, fault tolerance, OpenStack**

I. Introduction

Software fault injection (SFI) is a versatile tool for dependability assessment. In this approach, various types of system failure causes, namely faults or defects [1], are artificially inserted ("injected") into a running instance of the system. An behavioural investigation can show if the fault tolerance mechanisms of the system reacts in the intended way.

While numerous SFI approaches have been proposed and implemented in the past decades [2], SFI seems to remain more of a research topic rather than a commonplace software development tool. Reasons for SFI's lack of practical application may be usability issues, the challenge of finding an adequate and representative fault load [3], or the difficulty of answering the question of *when* and *where* to inject faults in order to achieve meaningful results.

We present a methodology for generating fault injection campaigns, which comprises multiple steps: First, a dependability model of the system is constructed (see Section II). Second, a fault injection campaign satisfying desirable criteria is generated from it (see Section III). Third, the fault injection campaign is conducted in an automated and orchestrated way. If the campaign succeeds, this asserts that the system is as dependable as specified in the initial model. If not, the experiment results can help to pin-point the weak parts of the architecture. We demonstrate the applicability of our approach with an OpenStack-based scenario, which is described in Section IV.

II. Dependability modelling for fault injection

Prior to any type of software evaluation, an understanding of what behaviour is expected under different circumstances must be formulated. Since software is so complex [4], there are manifold ways of creating such a description [5].

We chose to base our approach on the *fault tree* modelling language [6], which expresses a systematic deduction of failure causes starting with an undesired top event. The leaves of the tree, *basic events*, represent conceivable failure root causes, connected hierarchically with Boolean logic gates until the undesired top event is reached. If the system behaviour under fault injection and the fault tree model do not fit to each other, (at least) one of them must be flawed. Therefore, it is necessary to start with a trustworthy dependability model, f.e., by deriving it directly from architectural descriptions [7].

For the following sections, we assume that the system under test is represented in a fault tree model. For distributed cloud infrastructures, is seems reasonable to model on the level of machines and network connections. A more fine-grained investigation, e.g., the level of machine parts or software packages, is possible as well.

III. Generation of fault injection campaigns

A *fault injection experiment* is one execution of the system with a certain amount of faults being injected.A *fault injection campaign* is a set of fault injection experiments. Campaigns in our approach are designed for:

- High **coverage** of the space of possible faults;
- **Maximality** of experiments – the fault tolerance mechanisms should be exercised as intensively as possible;
- **Efficiency** of the overall campaign – all experiments must be conductible within a reasonable time frame.

The overall goal is to reduce the number of experiments within a campaign while still maintaining fault space coverage.The steps described subsequently yield such a fault injection campaign based on a given model.

A *fault injection point* (FIP) is an internal state change or external event which can contribute to fault activation, or directly causes a detectable error state. FIPs are the smallest possible target for fault injection, reflecting granularity at

TABLE I

AVERAGE PERFORMANCE DEGRADATION PER FIP AFTER 10 RUNS OF THE ENTIRE CAMPAIGN ON THE OPENSTACK TEST SYSTEM.

fault injection point	controller 1	controller 2	controller 3	storage 1	storage 2	storage 3	storage nw. at node 1	storage nw. at node 2	storage nw. at node 3
fail-stop	1.74	1.88	2.04	1.78	1.80	1.98	1.99	1.74	2.03
fail-silent	1.98	1.88	1.96	1.92	1.87	1.97	2.10	1.81	1.99
average	1.86	1.88	2.00	1.85	1.83	1.97	2.04	1.78	2.01

which the system is modelled and tested. Extracting all FIPs from a fault tree model simply means listing all basic events.

An *experiment* is a set of FIPs. Each experiment is carried out during exactly one program execution. This may be a test case or a period of normal system operation, in an artificial or ideally actual production environment. An experiment needs to have an externally measurable, binary outcome: It either succeeds, or it fails. This does not rule out the possibility of applying metrics at a continuous scale to the results. For example, performance under fault load can be analyzed by defining runtime thresholds, which must not be exceeded for "success". We assume that larger experiments – i.e., experiments containing more FIPs – achieve higher coverage and efficiency. Our approach therefore maximizes experiment size, relying on the concept of *minimal cut sets* (mincuts). The success of maximal experiments is assumed to imply the success of contained, smaller experiments.

IV. CASE STUDY: OPENSTACK

We have conducted a case study of our approach on a fault tolerant setup of the OpenStack cloud management system. For our purpose, OpenStack provides a suitable example: It is a complex, distributed, real-world software system, where fault tolerance is a key aspect and considered in all layers. The OpenStack community is pushing for frequent releases, which makes manual integration testing cumbersome.

Based on the description from the last section, we developed a tool chain automating the approach outlined above. It relies on scripting languages and the IT configuration management tool Ansible[1], which allows us to automate the entire process.

Our OpenStack setup comprises a master node, three controller instances, three Ceph storage nodes, and two compute nodes. The fully virtualized setup is distributed across five physical hosts. In the dependability model of the system, there are 11 basic events in total, hence 11 FIPs. In the naïve approach of running all conceivable permutations, this would amount to $2^{11} = 2048$ different possible experiments. We applied our proposed algorithm on this model and obtained 36 suggested experiments.

We implemented two failure cause types for the injection:

- *Fail-Stop*: The component either works correctly or detectably crashes. Specifically, a crash is achieved by terminating a VM (i.e., "pulling the power cord") or by removing the link of a virtual network interface (i.e., "pulling the network cable").
- *Fail-Silent*: The component either works correctly or does not respond. Specifically, this is achieved by freezing a VM or by silently dropping outgoing network packets.

Within our test bed, other fault classes are also conceivable: Byzantine faults can be injected by modifying, and not just dropping, packet content. To inject timing faults, we could also artificially delay packet transmission. The second step is the orchestrated execution of generated campaigns. Our implementation relies on exchangeable executables which inject particular fault types and prepare their setup.

Our experiments show that the system does not crash or freeze in any of the experiments – which indicates that the OpenStack infrastructure works as promised. As expected, we observed (varying) performance degradation under faultload. The average performance degradation per FIP is summarized in table I. Such observations, potentially caused by delays due to error detection via heartbeats, can be starting points for further detailed investigations.

V. CONCLUSION

We presented an approach for reproducible and automated fault injection campaigns in an distributed software environment. Our case study of an OpenStack deployment demonstrates the feasibility and real-world applicability of the approach. Compared to existing fault injection strategies, we see the capability for scalable automation as a major benefit of our approach. The presented approach could become part of continuous integration efforts that automatically run upon each software modification. The recently trending *Infrastructure as Code* paradigm provides the necessary machine-readable configuration and infrastructure information, which could be directly exploited for this purpose.

[1]https://www.ansible.com/, June 2, 2017

REFERENCES

[1] P. Tröger, L. Feinbube, and M. Werner, "What activates a bug? A refinement of the Laprie terminology model," in *26th IEEE International Symposium on Software Reliability Engineering*, 2015.

[2] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 44, 2016.

[3] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[4] B. Beizer, "Software is different," *Annals of Software Engineering*, vol. 10, no. 1, pp. 293–310, 2000.

[5] L. Feinbube, P. Tröger, and A. Polze, "The landscape of software failure cause models," *arXiv preprint arXiv:1603.04335*, 2016.

[6] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," 1981.

[7] C. Lauer, R. German, and J. Pollmer, "Fault tree synthesis from uml models for reliability analysis at early design stages," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 1, pp. 1–8, 2011.