Enhancing the Analysis of Error Propagation and Failure Modes in Cloud Systems

Domenico Cotroneo, Luigi De Simone, Alfonso Di Martino, Pietro Liguori, Roberto Natella
Università degli Studi di Napoli Federico II
{cotroneo, luigi.desimone, roberto.natella}@unina.it, {alf.dimartino, pi.liguori}@studenti.unina.it

***Abstract*—We argue for novel techniques to understand how cloud systems can fail, by enhancing fault injection with distributed tracing and anomaly detection techniques.**

I. Introduction

Cloud computing is becoming an attractive solution for running services with high-reliability requirements, such as in the telecom and healthcare domains. However, the reliability of cloud computing is hampered by the adoption of a complex stack of "off-the-shelf" components, including OSes, middleware, datastores, cloud management software, etc., which expose services to heterogeneous faults [1], [2]. Addressing these threats entails the systematic adoption of *fault injection*, but there are still open issues for its practical adoption.

II. Open issues in fault injection

Interpreting the outcome of fault injection experiments is a key step towards improving reliability. In particular, the analyst needs to assess the effects of the fault on the target system, and how they lead to a service failure, as they provide indications on where to improve fault tolerance. This assessment is typically pursued by **analyzing the spatial and temporal propagation of errors**. In the case of temporal propagation, the analysis identifies *latent* errors in the system, which manifest as a failure only after a period of time. Temporal propagation represents an opportunity for improving error handling: for example, by detecting the data affected by these errors with more thorough consistency checks, and by preventing that they turn into failures through software rejuvenation; or, if the error could not be recovered, by enforcing a *fail-stop* behavior, i.e., a service is stopped and a failure is notified to error handlers and/or to the users as soon as it occurs, in order to reduce its severity. In the case of spatial propagation, an error propagates across several components or layers of the cloud system, which increases the risk of cascading failures, and makes recovery more problematic (e.g., only recovering the last component in the propagation chain does not correct errors in the previous components). Spatial propagation can be prevented by blocking errors at components' interfaces, by looking at execution traces from fault injection experiments.

Error propagation analysis too often relies only on the knowledge, experience and intuition of human analysts, since existing fault injection solutions provide little support to the analyst for interpreting what happened during a fault injection experiment, which is especially cumbersome for systems with a large codebase and off-the-shelf software. Moreover, the analysis is made more difficult by the *non-determinism* of cloud systems. As in other types of distributed systems, the timing and the order of events is often not predictable, and can change even if there is no failure. Therefore, the analyst needs to distinguish between variations of the behavior that are caused by genuine errors, and variations that are caused by non-determinism and do not impact on the quality of service.

Identifying the failure modes of the system (that is, the impact of the fault on the service provided by the system) is another challenging aspect. Cloud systems can exhibit a large variety of failure modes, and not all these failure modes are necessarily known by the developers, due to the complexity of these systems. In the trivial case, the system fails by exhibiting an outage, but more subtle cases are also possible: for example, even if the system is available, it can still provide an incorrect service to the users, by returning wrong data, exhibiting poor performance, or corrupting the state of resources, leading to further service failures. Therefore, it is important to encompass a rich set of potential failure outcomes. Moreover, it is important to assess whether the system is able to detect these failures, since this is needed to trigger recovery. Ideally, the system should point out a failure by generating an explicit signal (e.g., an exception that can be handled by the service consumer, or by an automated recovery mechanism), and by logging as much useful information as possible about the failure, to let system administrators to diagnose the causes of the failure. Fault injection can reveal cases of missed detection that need to be covered.

Finally, **accelerating fault injection experiments** is a key challenge to make this approach useful in practice. From a technical perspective, it is very difficult to properly automate a large number of fault injection experiments, while assuring that the injected faults do not propagate from the target system to the fault injection system. Moreover, after an experiment, any residual effect must be cleaned-up, in order to prepare the target system for the next one. Unfortunately, the target system can corrupt its environment in subtle and unexpected ways (e.g., by propagating errors in a persistent datastore, which is the kind of error propagation issues that fault injection is intended to identify). On the one hand, it is important to clean-up any residual effect of an experiment (such as, to restore the state of a datastore, to release stale

This work has been supported by UniNA and Compagnia di San Paolo in the frame of "Programma STAR" (project "FIDASTE").

cloud computing resources, etc.). On the other hand, this clean-up can take a significant amount of time, thus increasing the duration of the experiments. If fault injection experiments take too long, they may be not affordable by developers.

III. Research Directions

Our current line of research is on supporting the analysis of error propagation and of failure modes in cloud systems, through a methodology that combines fault injection, black-box tracing, and anomaly detection (Fig. 1). The methodology handles the cloud system as a set of *black-box components*, which interact through *service-oriented* interfaces using messaging APIs (such as REST and message queues). The methodology collects *traces* of executions of the cloud system, that is, a sequence of *external service calls* from users to the cloud system, and of *internal service calls* from one internal component of the cloud system to another one.

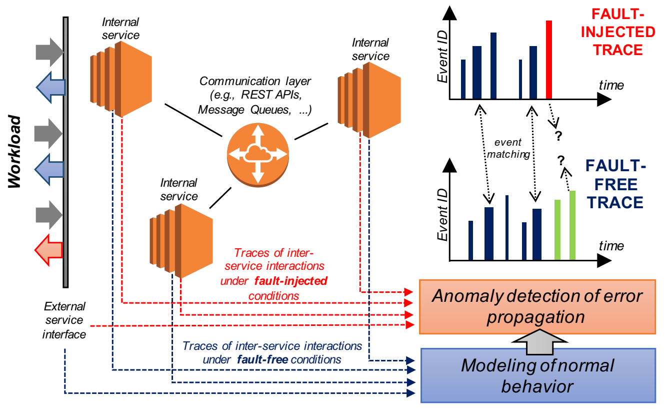
Workload
Internal service
Internal service
Internal service
Communication layer (e.g., REST APIs, Message Queues, ...)
External service interface
Traces of inter-service interactions under fault-injected conditions
Traces of inter-service interactions under fault-free conditions
Anomaly detection of error propagation
Modeling of normal behavior
Event ID
time
FAULT-INJECTED TRACE
event matching
?
?
FAULT-FREE TRACE

Fig. 1. Overview of error propagation and failure modes analysis.

Initially, the methodology executes the cloud system under fault-free conditions (i.e., without fault injection), by applying a workload with a sequence of service API requests. These service requests generate further requests among components inside the cloud system. During the execution of the workload, the methodology collects a set of *fault-free traces* that record all these requests. The traces are processed to create a model of the *normal behavior* of the cloud system. Due to non-determinism, the model generated by the methodology takes into account the "benign" variability of the interactions (e.g., different ordering, type, or duration of a subset of the events) that can occur under fault-free conditions. For this reason, the methodology adopts a probabilistic approach for modeling the variability of the training traces.

The methodology then performs a series of *fault injection* experiments on the cloud system. From every experiment, the methodology gets a *fault-injected trace*, which is individually analyzed by comparing it to the model of normal behavior, with the aim to point out *anomalies* with respect to this model. Many of the events in the fault-injected trace should not exhibit any "significant" difference with respect to the model of normal behavior, since all traces (both the fault-free and fault-injected ones) are collected under the same conditions (i.e., same software and hardware configuration, same workload, etc.); any deviation between a faulty trace and the model should be attributed to the injected fault, and reported to the analyst as part of the error propagation.

The results of anomaly detection are *visualized* by presenting the events both of the fault-injected and of the fault-free executions (e.g., the plots on the right side of Fig. 1). For visualization purposes, only one fault-free execution is presented to the analyst, by selecting the fault-free trace that is most similar to the fault-injected one. To pinpoint the propagation of errors through the interactions, the methodology matches the events of both traces, and highlights differences in terms of *missing* events (i.e., interactions that are not performed by the cloud system under faulty conditions) and *spurious* ones (i.e., interactions that do not happen under fault-free conditions). For every difference between the fault-free and fault-injected execution, the methodology uses the probabilistic model to estimate the likelihood of that difference, in order to hide noise and to highlight to the analyst only relevant differences. From these differences, the analyst can point out whether the error has been propagating across components (*spatial propagation*) and whether the errors keep propagating over a long period of time (*temporal propagation*).

Similarly, anomaly detection is applied to the interactions between the cloud system and its users. If there is an anomaly with respect to the sequence of services invoked by the workload, and with respect to the responses generated by these invocations, then the methodology can point out that the errors propagated into a failure of the cloud system. Moreover, the methodology correlates these failures with error signals raised by the cloud service interface, in order to identify cases where the error signal is delayed with respect to the failure (i.e., it is not raised in the same service call affected by the failure), or where the error is not signaled at all. To allow the methodology to detect the occurrence of service failures even in the cases when they are not explicitly signaled by the cloud service interface, the workload performs *assertion checks*, which assess the integrity of the state of the cloud system (for example, after requesting a resource update, the workload checks that the resource has consistently been updated).

Towards these goals, we foresee the need for addressing the following aspects: to develop accurate anomaly detection algorithms for modeling sequences of service interactions in a cloud system; to efficiently present fault injection experiments to human analysts, to guide them in the interpretation of error propagation and failure modes; to leverage anomaly detection to accelerate fault injection experiments, by identifying the components where errors were propagated, and selectively cleaning-up these components.

References

[1] D. Cotroneo, L. De Simone, AK Iannillo, A. Lanzaro, and R. Natella. Dependability evaluation and benchmarking of network function virtualization infrastructures. In *IEEE NetSoft 2015*.

[2] D. Cotroneo, L. De Simone, and R. Natella. NFV-Bench: a Dependability Benchmark for Network Function Virtualization Systems. *IEEE Trans. Net. Serv. Mgmt.*, 14(4), 2017.