FailViz: A Tool for Visualizing Fault Injection Experiments in Distributed Systems

Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella
Università degli Studi di Napoli Federico II
Naples, Italy,
{cotroneo, luigi.desimone, pietro.liguori, roberto.natella}@unina.it

Nematollah Bidokhti
Futurewei Technologies, Inc.
USA
nbidokht@futurewei.com

***Abstract*—The analysis of fault injection experiments can be a cumbersome task. These experiments can generate large volumes of data (e.g., message traces), which a human analyst needs to inspect to understand the behavior of the system under failure. This paper introduces the *FailViz* tool for visualizing fault injection experiments, which points out relevant events for interpreting the failures. We also present a motivating example in the context of *OpenStack*, and point out future research directions.**

***Index Terms*—Anomaly detection; Fault injection; OpenStack; Visualization**

I. Introduction

Fault injection is a fundamental technique to ascertain the fault-tolerance properties of distributed systems. This technique has been applied in the context of many safety- and business-critical domains, including railways [1], telecommunications [2], automotive [3], and air traffic control [4]. One key advantage of fault injection is that experiments can be automated and scale well. However, human intuition is still fundamental to turn the experimental results into insights to make a system more fault-tolerant.

In particular, understanding the behavior of distributed systems under failure can be a difficult task. The typical practice of looking for *failure predicates* over events and outputs [1] may not suffice to understand failures. If the distributed system is large and complex enough, it can fail in unexpected ways not anticipated at design time [5], [6], and the effects of the failure can be missed by the predicates. For example, even if the system is still available, it can provide an incorrect service to the users, by returning wrong data, exhibiting poor performance, or corrupting the internal state of resources. Moreover, human analysts need to scrutinize the experimental data to reconstruct the chain of events between the injected fault and the failure, so that they can break the chain to make the system more robust. Unfortunately, this can be a cumbersome task due to the large volume of data (logs, message traces, instruction traces, etc.) generated by complex systems. This issue is further exacerbated by the extensive use of *"off-the-shelf"* software components, either proprietary or open-source (such as application frameworks, middleware, datastores, etc.), whose events and protocols can be difficult to analyze and understand [7].

In this work, we develop a tool (*FailViz*) to aid human analysts at investigating failures in distributed systems under fault injection. The goal of the tool is to relieve the human analyst from manually inspecting thousands of events, by only pointing out events that are relevant for interpreting the failure. *FailViz* instruments the distributed system before fault injection, in order to be able to trace the messages exchanged within the system. Then, after the experiment, it analyzes the trace of messages during fault injection, by comparing the trace to a *reference fault-free run*, and by visualizing any divergent event to the human analyst. Therefore, the tool can be used by analysts to get a simplified overview of the experiments and to better understand the results. Moreover, we plan to use the tool as a basis for future research on automated debugging of failures in the distributed systems. *FailViz* complements previous state-of-the-art tools for the analysis of distributed systems (such as *Magpie* [8], *Pinpoint* [9], *Pip* [10], and *ShizViz* [11]) as it is specifically designed to analyze failures from fault injection, instead of failures occurred in production.

In the following of this paper, we provide the design rationale for the *FailViz* tool (Section II), and present a motivating example of failure analysis (Section III) in the context of *OpenStack*, a popular cloud management platform [12]. We conclude with directions for future research work (Section IV).

II. Design

The driving idea is to analyze the distributed system as a set of black-box components interacting through public service interfaces (e.g., REST APIs, message queues), in order to be applicable in large systems that include third-party software; and to use an anomaly detection algorithm on these interactions in order to pinpoint how the system behaves in the case of failures. The tool records and analyzes the *messages* that occur in the distributed system during fault injection. In general, messages are the key observation point for debugging and verification of distributed systems, as they reflect well the activity of the distributed system [13]. For example, nodes perform work when they receive messages to provide a service to another node (e.g., through remote procedure calls), and reply with messages to provide the response and results; moreover, nodes use messages to asynchronously notify a new state to other nodes in the distributed system. Therefore, developers need to look at messages to understand how the behavior of a system degenerated into a failure (e.g., it becomes unavailable to users), and how to mitigate the failure.

Fig. 1 shows the design of the tool. We first instrument communication APIs (step ①). Then, we exercise the system by applying a workload, and without fault injection (step ②). The tool records all messages exchanged among the components, and between the components and the clients. All these messages

978-1-7281-3929-6/19/$41.00 ©2019 IEEE
DOI 10.1109/EDCC.2019.00036

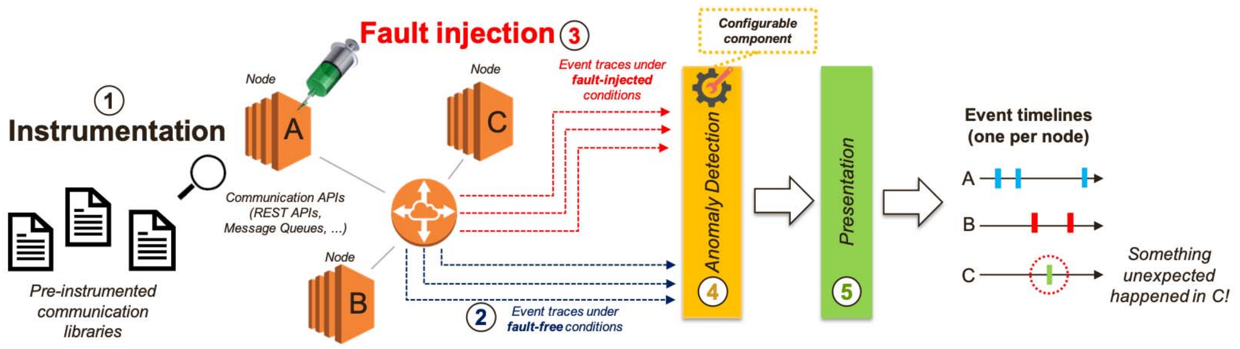

Fault injection ③
Configurable component
Event traces under fault-injected conditions
Node
Node
Node
① Instrumentation
A
B
C
Communication APIs (REST APIs, Message Queues, ...)
Pre-instrumented communication libraries
② Event traces under fault-free conditions
Anomaly Detection ④
Presentation ⑤
Event timelines (one per node)
A
B
C
Something unexpected happened in C!

Fig. 1. The workflow of the *FailViz* tool.

are recorded into a *fault-free trace*. Several fault-free traces are generated by repeating the workload several times.

After recording the fault-free traces, we perform a series of fault injection tests against the distributed system (step ③). This step will produce *fault-injected traces* (also *faulty traces*), one per test. The fault-injected traces are analyzed by looking for anomalies by using a configurable anomaly detection component (step ④). See II-C for details about anomaly detection component. Since all executions (both the fault-free and fault-injected ones) are performed under the same conditions (i.e., same software and hardware configuration, same workload, etc.), we look into deviations (*anomalies*) between the fault-injected trace and the fault-free trace, since these deviations include the effects of the injected fault. The results of anomaly detection are visualized by presenting to the human analyst the message trace of an experiment (step ⑤). The tool emphasizes messages that were omitted because of the injected fault (i.e. only happening in fault-free conditions), and new messages that were caused by the injected fault (i.e., only happening under faulty conditions). Moreover, *FailViz* supports the inspection of the anomalies by summarizing detailed information about the messages.

A. Instrumentation

The first step of *FailViz* consists of instrumenting the distributed system under test, in order to keep track of messages sent between nodes during a test. In particular, the tool records information by collecting traces of *communication API invocations* made by the distributed software. We designed the tool for providing a set of *pre-instrumented* communication libraries to trace messages. This design choice is motivated by the fact that developers seldom create new communication APIs from scratch, but they often re-use already available ones such as *REST frameworks* (e.g., Django [14], Spring [15]) and *message queueing* (e.g., AMQP [16], RabbitMQ [17]). Therefore, we collect message traces by installing pre-instrumented libraries (shipped with *FailViz*) in the target application. During the experiments, the libraries will send traces to a collector.

The tool adopts distributed tracing techniques to record every call to communication APIs, by using *probes* inserted in their source- or binary-code. This instrumentation is a form of "black-box tracing", since it does not require any knowledge about the internals of the distributed system under test, but only the knowledge about which communication APIs are used by the system. This approach is especially suitable when the testers may not have a full and detailed understanding of the entire distributed system. For example, this is the case of distributed systems developed by large teams (and in which testers and developers are distinct people), or distributed systems that embed components developed by third-parties. Moreover, distributed tracing is already familiar to developers for debugging, performance monitoring and optimization, root cause analysis, and service dependency analysis [18], [19].

In our implementation, we trace the application by using a context manager/decorator mechanism of the *Zipkin* distributed tracing system [20]. The instrumented application sends data via HTTP to the collector, which stores trace data. We record information about the exchanged messages, such as the sender of the message, the name of the service API that is being called, the timestamp of the message and its duration. We refer to the traced calls as *events*, and to the sets of events as *traces*. The tool assigns an identifier (*symbol*) to each recorded event, such that two events are identified by the same symbol if they have the same pair <*sender of the message, name of the called service API*>. Therefore, traces are represented as sequences of symbols.

B. Data collection

Once the distributed system has been instrumented, it is executed several times to perform fault injection tests. The distributed system is monitored during the execution; at the same time, the system is stimulated with a workload (e.g., by generating client requests), and a fault is injected into the system. Each test injects a different fault, and only one fault is injected per test. For each test, we collect a trace (**fault-injected trace**) of the messages that are generated in the system during the execution of the workload. Therefore, we obtain several traces, one per test.

In addition to fault-injected traces, we also execute the workload and collect traces without injecting any fault (**fault-free traces**). Such fault-free traces (also known as *golden runs* or *reference runs*) have been adopted in the past for fault injection experiments in small systems (e.g., embedded ones), by using the traces as a reference to understand how the fault-injected system derailed from a proper execution [21]–[23]. In order to collect fault-free traces, the tool executes the same workload (also used in fault injection tests) N times, but without injecting any fault. The messages sent in each execution are saved in a fault-free trace, with one fault-free trace per workload execution. We use more than one fault-free trace since we need to take into account the variability of the execution that can occur in distributed systems (e.g., the relative ordering of messages during execution).

C. Anomaly detection

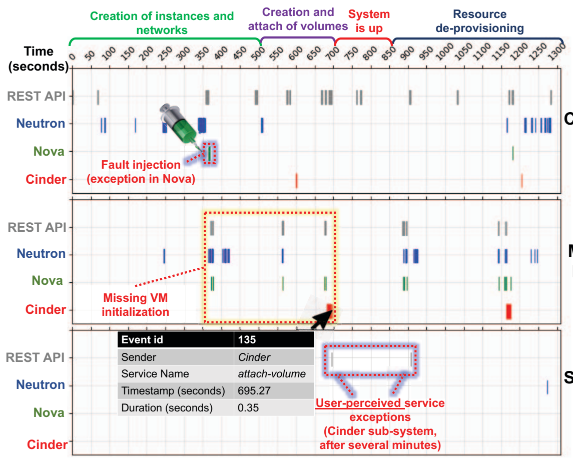
In order to analyze failures during a fault injection test, the tool looks for "anomalies" in the fault-injected execution, and points them out to the human analyst. The analysis compares the fault-injected execution with the reference execution (i.e., the "golden run") and identifies the differences between them. The ultimate result of the tool is a classification of the events of the fault-injected trace into:

- **Common events**: Events that occurred both in the fault-injected trace and in at least one of the fault-free traces, with the same type and order.
- **Anomalous events**: Differences between the fault-injected trace and all of the fault-free traces. These events are further classified into:
  - **Spurious events**: Events that would normally not happen under fault-free conditions.
  - **Missing events**: Events that happen in fault-free conditions, but do not happen under fault injection.

The component that deals with anomaly detection can be configured by the user, which may choose among different algorithms. In the initial version of *FailViz*, we identify anomalies by using a simple algorithm, which we will extend in future work. The anomaly detection component performs a string comparison of two sequences (respectively the fault-injected sequence, and one of the fault-free sequences), by looking for the *longest common subsequence* (LCS) of the strings [24]. The LCS is a subset of symbols that are present in both sequences in the same order, and that can be obtained by removing (a minimal number of) symbols from the original sequences. This kind of problem is recurrent in computer science, like in bioinformatics and in source code versioning (e.g., in the *diff* Unix tool), and can be solved with efficient algorithms [25], [26].

In order to perform the comparison, the tool selects one fault-free trace among the ones collected at the beginning of the workflow. In particular, the tool selects the fault-free trace *most similar* to the fault-injected trace, since we want to identify and to filter out from the failure analysis as much *common* events as possible (i.e., the tool aims to discard the subset of messages that also happen with the same type and order in at least one fault-free execution), in order to draw the attention of the human analyst on *anomalous* events.

The similarity between two strings x and y is measured by considering the length of the LCS ($|LCS(x,y)|$) [27], that is, the number of symbols that appear in both strings while preserving the order of symbols. In particular, we compute the normalized length of the LCS ($nLCS$), where $nLCS(x,y) = \frac{|LCS(x,y)|}{\sqrt{l_x \cdot l_y}}$, and l_x and l_y are the lengths of the individual strings x and y. The tool uses this metric to identify the fault-free trace of the training set most similar to the fault-injected trace.



Event id	135
Sender	*Cinder*
Service Name	*attach-volume*
Timestamp (seconds)	695.27
Duration (seconds)	0.35

Fig. 2. Graphical visualization of a fault injection experiment in OpenStack.

III. Example

In this section, we present an example of a fault injection experiment on the OpenStack cloud computing platform. We instrumented the following communication points:

- The *RESTful API libraries* of the OpenStack subsystems (e.g., Nova, Neutron, Cinder) used for communication between OpenStack and its clients. Each OpenStack subsystem includes a *client* component, which includes API bindings for communication.
- The *OSLO Messaging library*, which uses a message queue library, by exchanging messages with an intermediary queuing server (RabbitMQ) through RPC messages. These messages are used for communication among OpenStack subsystems.

We instrumented only 5 selected functions of these components, by adding a total of 20 lines of Python code.

This experiment injected a fault in the Nova subsystem, which manages VM instances in OpenStack. During the experiment, OpenStack was exercised by a workload, which emulated a system administrator or customer that deploys a new virtual infrastructure, by calling the OpenStack REST APIs. One of these API calls is an asynchronous request to create a new VM instance. After the API call ends, Nova takes a few minutes to create and initialize the instance. During these operations, we injected a Python exception to force a failure.

Fig. 2 shows the output provided by *FailViz* for the selected experiment. The graphical representation is oriented to a human analyst that needs to understand what happened during the experiment. This representation shows the events between the OpenStack clients and the OpenStack subsystems (labeled as *REST API*), and the inter- and intra- subsystems API calls events (labeled using the name of the subsystem).

In order to point out how the fault impacted on the system, this representation divides the events among *common*, *missing*, and *spurious* ones. The groups are obtained by applying an anomaly detection algorithm (§ II-C).

FailViz provides an interactive visualization of the experiment. A user can investigate a specific event by pointing the mouse on it: the tool displays a table with information about the event, which is important to facilitate the analysis of the failure. Our implementation uses *mpld3* [28], a library that brings together *Matplotlib*, the popular Python-based graphing library, and *D3js*, the popular JavaScript library for creating interactive data visualizations for the web. In the figure, we notice a large number of missing events. The failure affected several OpenStack subsystems over a relatively long time period. These events include several internal calls to initialize the instance and to attach it to its virtual resources (the "propagation chain" of the failure). The spurious events, instead, include the exceptions of two REST API calls to the client.

In our example, due to the injected fault, Nova did not complete the initialization of the VM instance, leaving it in an inactive state. Later on, after 5 minutes, the workload client experienced a service exception when calling the API of the Cinder subsystem, which manages storage volumes in OpenStack. By investigating on the event pointed by the mouse, we notice that the event <*cinder-volume, attach-volume*> did not occur in the faulty execution (i.e., a missing event). Thus, *FailViz* helps the analyst in understanding that the workload did not attach a volume to the VM instance during the faulty execution.

Moreover, the OpenStack Neutron subsystem was also unable to attach the VM instance to the virtual network. Both Nova and Neutron did not raise any API exception, but the failure only became apparent to the client when invoking the API of the Cinder subsystem. Therefore, the problem propagated both across subsystems (from Nova to Neutron and Cinder) and across time, since the client perceived the failure only after a relatively long time. This behavior is problematic from the point of view of high-availability, as the propagation delay also increases the time-to-detect and the time-to-recover the failure. Moreover, the longer the propagation chain, the more difficult will be for a developer to reason about how to best tolerate the fault, e.g., whether to manage the fault in Nova, Neutron and/or Cinder and at which time to manage the fault during the workflow. For example, the API could return a more timely notification of the failure to the client, either by introducing a callback mechanism in the Nova API that creates the instance or by returning an error from other API calls to Nova or Neutron.

IV. Conclusion and Future Work

We presented *FailViz*, a novel tool for analyzing fault injection experiments in distributed systems. We designed *FailViz* to be non-intrusive, highly portable, and configurable with several anomaly detection algorithms. In this paper, we applied *FailViz* with a basic anomaly detection algorithm in OpenStack. We aim to add new approaches for anomaly detection (e.g., approaches based on probabilistic models, neural networks, combined models, etc.), in order to enable the possibility to configure the anomaly detection component. This will allow us to compare several approaches for anomaly detection on different targets and with several workloads. In particular, we will evaluate the robustness of anomaly detection algorithms with respect to non-determinism effects in distributed systems.

Acknowledgment

This work has been partially supported by Critiware s.r.l. and by Huawei Technologies Co. Ltd.

References

[1] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE TC*, vol. 42, no. 8, pp. 913–923, 1993.

[2] K. R. Joshi, W. H. Sanders, M. A. Hiltunen, and R. D. Schlichting, "Automatic recovery using bounded partially observable markov decision processes," in *Proc. DSN*. IEEE, 2006, pp. 445–456.

[3] T. Piper, S. Winter, N. Suri, and T. E. Fuhrman, "On the effective use of fault injection for the assessment of AUTOSAR safety mechanisms," in *Proc. EDCC*. IEEE, 2015, pp. 85–96.

[4] R. Natella and D. Cotroneo, "Emulation of transient software faults for dependability assessment: A case study," in *Proc. EDCC*. IEEE, 2010, pp. 23–32.

[5] P. Garraghan, R. Yang, Z. Wen, A. Romanovsky, J. Xu, R. Buyya, and R. Ranjan, "Emergent failures: Rethinking cloud reliability at scale," *IEEE Cloud Computing*, vol. 5, no. 5, 2018.

[6] K. J. Hole and C. Otterstad, "Software systems with antifragility to downtime," *IEEE Computer*, vol. 52, no. 2, 2019.

[7] D. Cotroneo, L. De Simone, A. Di Martino, P. Liguori, and R. Natella, "Enhancing the analysis of error propagation and failure modes in cloud systems," in *Proc. ISSREW*. IEEE, 2018, pp. 140–141.

[8] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems." in *Proc. HotOS*, 2003, pp. 85–90.

[9] Y.-Y. M. Chen, A. J. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, "Path-based failure and evolution management," in *Proc. NSDI*, 2004, pp. 309–322.

[10] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *Proc. NSDI*, vol. 6, 2006, pp. 9–9.

[11] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Queue*, vol. 14, no. 2, p. 50, 2016.

[12] OpenStack project, "User stories showing how the world #RunsOnOpenStack," 2018. [Online]. Available: https://www.openstack.org/user-stories/

[13] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 517–530, 2016.

[14] Django. [Online]. Available: https://www.djangoproject.com

[15] Spring. [Online]. Available: https://spring.io/projects/spring-framework

[16] AMPQ. [Online]. Available: https://www.amqp.org/

[17] RabbitMQ. [Online]. Available: https://www.rabbitmq.com/

[18] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services." in *Proc. OSDI*, 2014, pp. 217–231.

[19] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. DSN*. IEEE, 2002, p. 595.

[20] Zipkin. [Online]. Available: https://zipkin.io

[21] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[22] M. Leeke and A. Jhumka, "Evaluating the use of reference run models in fault injection analysis," in *Proc. PRDC*. IEEE, 2009, pp. 121–124.

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

[24] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proc. SPIRE*. IEEE, 2000, pp. 39–48.

[25] J. W. Hunt and M. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[26] E. W. Myers, "An O (ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.

[27] S. Budalakoti, A. N. Srivastava, M. E. Otey *et al.*, "Anomaly detection and diagnosis algorithms for discrete symbol sequences with applications to airline safety," *IEEE Trans. on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 39, no. 1, p. 101, 2009.

[28] mpld3 Library. [Online]. Available: http://mpld3.github.io/