Explainable Root Cause Analysis for Failing Microservices

Jacopo Soldani, Stefano Forti, and Antonio Brogi

University of Pisa, Pisa, Italy, {name.surname}@unipi.it

Abstract

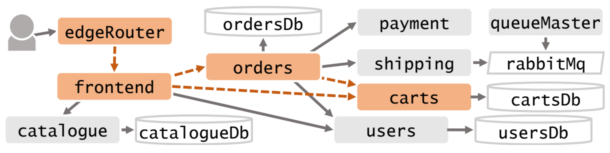
Determining the root causes of observed failures is a main issue in microservice-based applications. Unfortunately, available root cause analysis techniques do not focus on explaining *how* root failures actually caused the observed failure. On the other hand, the availability of such explanations would greatly help to pick adequate countermeasures, e.g., by introducing circuit breakers or bulkheads. We hence present a declarative root cause analysis technique, which can determine the cascading failures that possibly caused an observed failure, identifying also (or starting from) a root cause. We also introduce a prototype implementation of our technique, and briefly comment on how we used it to assess our technique by means of controlled experiments.

1 Introduction

Microservices gained momentum in enterprise IT, with Netflix and Spotify already being delivered as microservice-based applications, for instance [14]. Microservice-based applications are considered *cloud-native*, since they are composed by loosely coupled services, which can be independently deployed and scaled to fully exploit the potentials of cloud computing [4].

Microservice-based applications are often composed by hundreds of services, which are replicated by instantiating multiple instances of each service. Multiple instances of the various services in an application interact to respond to end users' requests, possibly resulting in thousands of interactions happening at the same time. Service instances can fail, e.g., by returning error responses to their invokers, or not answering at all since they suddenly crashed.

Understanding the possible root causes for a failing service instance is inherently complex. Did it fail on its own? Did it instead fail in cascade, since it interacted with another failing service instance? Did the latter fail on its own or in cascade to some other service instance? Answering such questions is not easy, when possibly thousands of interactions among different service instances happen at the same time [14]. At the same time, answering the above questions is crucial to enact countermeasures and avoid the same failure cascade to happen again [10].



Motivating Scenario. Consider *Sock Shop* [15], whose microservice-based architecture is displayed on the right (where darker nodes and dashed arrows highlight the portion considered in this motivating scenario). Suppose that we deployed two replicated instances of each microservice in *Sock Shop*, and that an instance of `carts` fails, e.g., by starting to return error responses to its clients because of an internal error. Suppose also that `frontend`'s instances can tolerate the failure of `carts`' instances, e.g., by caching carts. Suppose instead that `orders`' instances fail when `carts` replies with error responses, becoming unable to process the requests from `frontend`'s instances. When this happens, we have cascading failures in `frontend` as well, due to which *Sock Shop*'s end users cannot place orders.

For *Sock Shop* to get back fully working, application operators must identify the internal failure of an instance of `carts` as the root cause of the failures in `frontend`'s instances, as well as that such root causing failure propagated to `frontend` through `orders`. This would enable first recovering the failing instance of `carts`, e.g., by restarting it, which would then result in the instances of `orders` and `frontend` getting back fully working as well. Also, by identifying the failure cascades that made `frontend`'s instances unable to place orders, application operators could operate only on such cascades to avoid this to happen again. For instance, they could introduce a circuit breaker enabling `orders` to tolerate the failure of `carts`' instances, whilst not intervening on `frontend`, which can already tolerate the failure of `carts`.

Existing root cause analysis techniques can help determining the service instances that may have failed first [12]. This is typically done by correlating the performance of the different service instances or the events they log, so as to determine the set of possible root causes. Identified root causes are also sometimes ranked by returning first those having higher chances to have caused the observed failure. However, there is no explanation of *how* root causing failures propagated to other service instances, up to causing the failure observed on a service instance. Explanations —given as the possible failure cascades originating from an identified root causing failure— would enable intervening not only on the service that first failed, but also on those that failed in cascade [10]. They would enable, e.g., to equip intermediate services with circuit breakers enhancing the failure resilience of their instances [8], or to introduce bulkheads limiting failure propagations to only certain parts of an application [7].

Our Contribution. We propose an *explainable* root cause analysis technique, which automatically determines both the possible root causes for a failure observed on a service instance, and the failure cascades due to which the root causing failure possibly propagated up to that observed. It can also be used by restricting the possible root causes to a given set, hence enabling to explain the possible root causes identified with other existing techniques. In both cases, the explainable root cause analysis starts from the distributed logs of an application's service instances. Such logs are processed by means of declarative rules, which enable eliciting the interactions occurring among service instances, and determining whether a service instance failed on its own or in cascade, e.g., because it interacted with another failing service instance.

We also introduce a prototype implementation of our technique, and we briefly comment on its use in controlled experiments. The experiments' results show that our technique effectively determined the true root causes/explanations in 99.74% of the considered cases.

2 Related Work

Our survey [10] reviews existing techniques for automatically identifying the possible root causes of failures in microservice-based applications. Such techniques are classified based on the granularity of analysed failures, by distinguishing techniques analysing *application-level* failures from those analysing *service-level* failures. Application-level techniques, such as, e.g., [1, 3, 5], enable determining the possible root causes of failures observed on the frontend of an application. Service-level techniques, such as the one we propose and, e.g., [6, 13, 16], instead enable determining the possible root causes of failures observed on any service in an application. Service-level techiques hence enable analysing failures at a finer granularity [10].

The main difference between existing service-level techniques and the one we propose resides in *explainability*. Existing techniques can effectively determine the possible root causes for a service failure, often also ranking them based on the probability for an identified root cause to have caused the observed failure. This is done by correlating the performances or events happening on the services forming an application. For instance, [13] directly correlates metrics monitored on application services, whilst [6] and [16] exploit such correlation to drive a random walk on a graph representing the architecture of an application or causal relationships between the service therein. Similar approaches are adopted by other existing techniques, always resulting in the lack of explanations of how identified root causing failures propagated and caused that observed [10]. The latter is precisely the aim of our work, which can determine the possible root causes for observed failures or start from those identified with existing techniques, while providing explanations of how root causing failures propagated and caused the observed ones.

In this perspective, the technique in our previous work [11] is closer to the one in this paper, given that it explains how root causing failures propagated to cause an observed failure. Such technique however relies on a specification of the application architecture and of the failure behaviour of each of the service therein, which is used to drive the analysis of application logs. The technique in this paper hence differs from that in our previous work [11], given that we directly process the application logs, without requiring any specification of the application.

In summary, to the best of our knowledge, the service-level technique in this paper is the first enacting root cause analysis by also identifying the cascades due to which the root causing failures propagated and caused that observed. It is the first doing it without requiring any other inputs than the logs produced by the services forming an application. At the same time, our technique can complement the results obtained with other existing techniques, by explaining how the root causing failures they identify propagated and caused that observed.

3 Explainable Root Cause Analysis

Logs & Interactions. Our technique works on a simple representation of application logs modelled as Prolog facts of the form `log(SName, SInstance, Timestamp, Event, Message, Severity)`, where the name `SName` and the instance identifier `SInstance` of a logging service are followed by the log `Timestamp`, the type of the logged `Event`, the associated log `Message` (if any), and its `Severity` (expressed according to the Syslog standard [2]).

Our methodology currently handles the following types of events: (i) `internal`, which denotes logs related to the internal business logic of the considered service, (ii) `sendTo(DstService, SessionId)`, which denotes that a request was sent to an instance of `DstService` with an associated `SessionId`, (iii) `received(SessionId)`, which denotes reception of a message at a particular instance of the destination service within the interaction identified by `SessionId`, (iv) `timeout(DstService, SessionId)`, which denotes that the interaction started towards `DstService`, identified by `SessionId`, incurred in a timeout, and (v) `errorFrom(DstService, SessionId)`, which denotes that the the destination service replied with an error code within the interaction identified by `SessionId`.

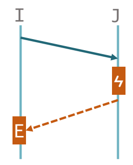
Based on this simple modelling and on a fistful of declarative Prolog rules,[1] our technique can identify: (a) *successfully completed* interactions between services, (b) service requests that incurred in a timeout event and were *not received*, and (c) interactions that *failed* either due to a logged error or to an expired timeout.

Explanations. Our technique recursively builds explanations for logs associated to a failure, so as to determine their root cause. It is worth noting that, thanks to Prolog resolution mechanisms, our technique permits instantiating a specific root cause service as an input parameter so to restrict the obtained explanations only to those events that have such a service as their

[1] https://github.com/di-unipi-socc/yRCA/blob/main/explainer/prolog/explain.pl.

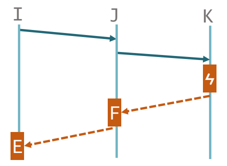
failure cascade root cause. If, conversely, no root cause is specified, our technique determines all explanations for all possible root causes. This enables using our technique both as an *explainer* in pipeline to other existing tools for root cause identification as well as a stand-alone tool.

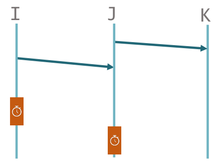
By recurring on logged events, our technique can explain 8 cases – 5 recursive and 3 base cases – corresponding to different possible cascading and root failures, respectively. For each case, we briefly summarise it offering graphical sketches to epitomise recursive cases.



1. Internal error of invoked service instance. Our technique explains that a failure or timeout event `E` at service `I`, happening at the end of a failed or timed-out interaction with service `J`, may have been caused by an internal failure (i.e., a logged event whose severity is at least *warning*) at service `J`. In this case, our technique recurs on the internal error of `J` to explain it.

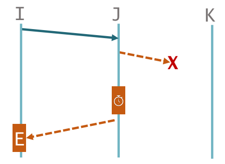
2. Failed interaction of invoked service instance. Our technique explains that a failure or timeout event `E` at service `I`, has been caused by a failure event `F` at service `J`, which – in turn – has been caused by a failed interaction of `J` with service `K`. In this case, after identifying the failure cascade `K` → `J` → `I`, our technique recurs on the log of `F` to explain it.

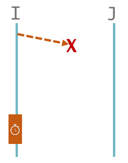




3. Timed-out interaction of invoked service instance. Our technique explains that a timeout event at service `I` has been caused by a timeout at service `J`, which – in turn – has been caused by a timeout related to an interaction of `J` with service `K`. In this case, after identifying the timeout cascade `K` → `J` → `I`, our technique recurs on the timeout at `J` to explain it.

4. Unreachability of a service called by invoked service instance. Our technique explains that a failure or timeout event `E` at service `I` has been caused by a timeout at service `J`, which – in turn – has been caused by a failed interaction of `J`. Differently from cases 2 and 3, our technique here considers the possibility of the request sent by `J` to have never been received by the target service `K`, then causing the interaction to fail because of a timeout. In this case, after identifying the failure cascade `K` → `J` → `I`, our technique recurs on the log of the timeout at `J` to explain it.





5. Unreachability of invoked service instance. Our technique explains that a timeout event at service `I` has been caused by a non-received request during an interaction of `I` with service `J`. In this case, our technique abducts a new piece of knowledge in the explanation, i.e. that `J` was unreachable, and recurs to explain it.

6. Internal service error. This case explains an `internal` failure event logged by a service, identifying the service itself as the root cause for such an event. Recursion ends.

7. Temporary service unreachability. This case explains abducted unreachability events for a service, identifying that such a service was temporarily unreachable because it previously logged some information. Recursion ends.

8. Unstarted service. This last case explains abducted unreachability events for a service, identifying that such a service never logged any information. Recursion ends, by abducting the fact that such a service was possibly never started.

Prototype. Our technique comes with an open source prototype, called yRCA.[2] yRCA encodes our explainable root cause analysis by means of a Prolog reasoner, which is embedded in a Python-based command-line tool. An example of output returned by yRCA is shown hereafter, with possible explanations for the failure mentioned in our motivating scenario (Section 1):

```
[0.615]: edgeRouter: Error response (code: 500) received from frontend
(request_id: [<requestId>])
   -> frontend: Error response (code: 500) received from orders (request_id: [<requestId>])
   -> orders: Failing to contact carts (request_id: [<requestId>]). Root cause: <exception>
   -> carts: unreachable
[0.385]: edgeRouter: Error response (code: 500) received from frontend (request_id: [<requestId>])
   -> frontend: Failing to contact carts (request_id: [<requestId>]). Root cause: <exception>
   -> carts: unreachable
```

By default, yRCA groups the possible explanations based on the structure of the failure cascade, and it ranks the different explanations based on the frequency with which they occur in all identified failure cascades (with such frequency indicated between square brackets at the beginning of each explanation). The idea is that the more frequent is an explanation, the higher is the probability that it corresponds to the true explanation for an observed failure. This is inspired by other existing analysis techniques, which rank the identified root causes by giving higher ranks to those found with a higher rate [10].

4 Conclusions

We presented a technique for explaining cascading failures in microservice-based applications. Our technique can determine the failure cascades that possibly caused an observed failure, either also eliciting the possible root causes or starting from given root causes, e.g., when another existing technique is used to identify the possible root causes for an observed failure.

Our technique comes with a prototype implementation (called yRCA), which we used to assess it based on controlled experiments run on an existing chaos testbed. In particular, we configured the Chaos Echo testbed [9] to obtain a reference application whose architecture mirrors that of *Sock Shop*, and whose services can be configured to control their interaction and failure behaviour. We then run the reference application by varying the configuration of its services so as to analyze the performances of yRCA when varying four different parameters, viz., (a) end-user load, (b) service interaction rate, (c) failure cascade length, and (d) service failure rate. For each case, we considered 200 generated failures, whose true root causes/explanations were effectively identified by yRCA in 99.74% of the cases. The effectiveness of our technique however not only comes from this, but also from the number of returned false positives, viz., failure cascades considered to have possibly caused the observed failure, even if this was not the case. False positives should be kept low, as they require application operators to waste time and resources in unnecessarily checking them [10]. We therefore measured the average number of failure cascades identified by yRCA, one being the right solution and the other being false positives. On average, we had at most 2 different root causes identified, for a total of at most 3 possible explanations for an observed failure. yRCA hence kept the number of false negatives low on average, thus limiting the efforts hat should be spent by application administrators in checking failure cascades that did not truly caused an observed failure.

Finally, it is worth noting that the failure cascades explaining an observed failure can help application administrators in identifying where to enact suitable countermeasures (e.g., circuit breakers and bulkheads [7]) to avoid the occurrence of those failure cascades. One natural

[2] https://github.com/di-unipi-socc/yRCA.

direction for future work is the prototyping of a tool supporting the visualization of failure cascades explaining observed failures, together with suggestions of possible countermeasures. Another interesting future work direction is extending the scope of our explainable root cause analysis to deal with incomplete logs, e.g., in case the logging driver fails or a service instance gets suddenly killed without flushing all its logs.

References

[1] Aggarwal, P., et al.: Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals. In: Service-Oriented Computing. LNCS, vol. 12632, pp. 137–149. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-76352-7_17

[2] IETF: The Syslog protocol. RFC 5424, Network Working Group (2009)

[3] Kim, M., et al.: Root cause detection in a service-oriented architecture. SIGMETRICS Perform. Eval. Rev. **41**(1), 93–104 (2013). https://doi.org/10.1145/2494232.2465753

[4] Kratzke, N., Quint, P.: Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. J. Syst. Soft. **126**, 1–16 (2017). https://doi.org/10.1016/j.jss.2017.01.001

[5] Liu, P., et al.: Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). pp. 48–58. IEEE (2020). https://doi.org/10.1109/ISSRE5003.2020.00014

[6] Meng, Y., et al.: Localizing failure root causes in a microservice through causality inference. In: 2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS). pp. 1–10. IEEE (2020). https://doi.org/10.1109/IWQoS49365.2020.9213058

[7] Newman, S.: Building Microservices. O'Reilly Media, 2 edn. (2021)

[8] Richardson, C.: Microservices Patterns. Manning Publications, 1 edn. (2018)

[9] Soldani, J., Brogi, A.: Automated generation of configurable cloud-native chaos testbeds. In: Dependable Computing - EDCC 2021 Workshops. pp. 101–108. Springer (2021). https://doi.org/10.1007/978-3-030-86507-8_10

[10] Soldani, J., Brogi, A.: Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. ACM Comput. Surv. **55**(3) (2022). https://doi.org/10.1145/3501297

[11] Soldani, J., et al.: What went wrong? Explaining cascading failures in microservice-based applications. In: Service-Oriented Computing. pp. 133–153. Springer (2021). https://doi.org/10.1007/978-3-030-87568-8_9

[12] Solé, M., et al.: Survey on models and techniques for root-cause analysis. CoRR abs/1701.08546 (2017)

[13] Wang, L., et al.: Root-cause metric location for microservice systems via log anomaly detection. In: 2020 IEEE International Conference on Web Services (ICWS). pp. 142–150. IEEE (2020). https://doi.org/10.1109/ICWS49710.2020.00026

[14] Waseem, M., et al.: Design, monitoring, and testing of microservices systems: The practitioners' perspective. J. Syst. Soft. **182**, 111061 (2021). https://doi.org/10.1016/j.jss.2021.111061

[15] Weaveworks, Container Solutions: Sock shop. https://microservices-demo.github.io (2017)

[16] Wu, L., et al.: MicroRCA: Root cause localization of performance issues in microservices. In: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium. pp. 1–9. IEEE (2020). https://doi.org/10.1109/NOMS47738.2020.9110353