Microservices Monitoring with Event Logs and Black Box Execution Tracing

Marcello Cinque, *Member, IEEE,* Raffaele Della Corte, *Member, IEEE,* and Antonio Pecchia, *Member, IEEE,*

Abstract—Monitoring is a core practice in any software system. Trends in microservices systems exacerbate the role of monitoring and pose novel challenges to data sources being used for monitoring, such as event logs. Current deployments create a distinct log per microservice; moreover, composing microservices by different vendors exacerbates format and semantic heterogeneity of logs. Understanding and traversing the logs from different microservices demands for substantial cognitive work by human experts. This paper proposes a novel approach to accompany microservices logs with black box tracing to help practitioners in making informed decisions for troubleshooting. Our approach is based on the passive tracing of request-response messages of the REpresentational State Transfer (REST) communication model. Differently from many existing tools for microservices, our tracing is application transparent and non-intrusive. We present an implementation called MetroFunnel and conduct an assessment in the context of two case studies: a Clearwater IP Multimedia Subsystem (IMS) setup consisting of Docker microservices and a Kubernetes orchestrator deployment hosting tens of microservices. MetroFunnel allows making useful attributions in traversing the logs; more important, it reduces the size of collected monitoring data at negligible performance overhead with respect to traditional logs.

Index Terms—monitoring; microservices; REST; Docker; Clearwater; Kubernetes; log analysis.

1 Introduction

MONITORING entails gathering a variety of **data sources** that pertain the execution of a given system, such as resource usage metrics, network statistics, traces and logs: it is a *core* engineering practice in any software system for assuring service continuity and downtime reduction [1]. Trends in **microservices systems** exacerbate the role of monitoring. Microservices put forth *reduced size*, *independency*, *flexibility* and *modularity* principles [2], which well cope with ever-changing business environments. However, as real-world applications are decomposed, they can easily reach *hundreds* of microservices (e.g., the deployments of Netflix or Hailo). This inherent complexity determines an increasing difficulty in debugging, monitoring and forensics [3].

Recent work started to address monitoring challenges in deploying Virtual Machines (VMs), which –similarly to *containers*– can be used to host microservices. Given the large availably of Cloud providers, Fadda *et al.* [4] propose a multi-objective mixed integer linear optimization approach to maximize the *quality* of monitoring, which is usually neglected in favor of other characteristics, such as *size* and *availability* of the VMs. While [4] addresses **indirect monitoring** (i.e., the collection of metrics, such as *CPU usage*, *free memory* and *power consumption*, that do not require *active* participation by the software under-monitoring[1]), microservices are posing even more challenges in the area of **direct monitoring**, which rely on source code *instrumentation* –e.g., by means of calls to logging functions or tracing APIs– for obtaining monitoring data at runtime.

Event logs are the *primary source* of data for direct monitoring. Event logs are sequences of text lines –typically stored in *log files*– reporting on the runtime behavior of a computer system; the use of logs is known since the early days of computers [5]. We claim that architectural *advantages* brought by microservices **clash** with event logs practices. For example, microservices systems (i) result from the composition of software developed by differently-skilled teams; (ii) are distributed across heterogeneous platforms and technologies; (iii) are strongly *dynamic*, with microservices being frequently added, updated or replicated for scalability and fault tolerance. Current deployments create a distinct log per microservice; moreover, centralization and collection of event logs require an infrastructure available at each node of the system.

In spite of the technical advances achieved by up-to-date log management tools like Splunk [6] or Logstash [7], practitioners face compelling challenges in maintaining catalogues of *regular expressions* and *keywords* for parsing and monitoring runtime logs. The composition of microservices exacerbates format and semantic heterogeneity of logs caused by the lack of standard coding practices across developers and vendors. Even more important, understanding and traversing the logs from different microservices demands for substantial *cognitive work* by human experts in forensics and troubleshooting.

This paper proposes to accompany microservices' logs with **black box tracing** in order to help practitioners to make **informed decisions** when traversing heterogeneous logs for troubleshooting. Tracing accounts for various data, such as source-destination of invocations, response codes and runtime checks on timeout expirations. We present a

• *Marcello Cinque, Raffaele Della Corte and Antonio Pecchia are with the Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione (DIETI), Università degli Studi di Napoli Federico II, via Claudio 21, 80125, Napoli, Italy.*
E-mail: {macinque, raffaele.dellacorte2, antonio.pecchia}@unina.it

1. Indirect monitoring capitalizes on non-intrusive operating system-level probes or kernel modules to collect the metrics.

specific approach to this goal, which builds on **passive tracing** (also known as *sniffing*) of request-response messages that underlie the REpresentational State Transfer (REST) communication model to transparently build microservices execution traces. We stem from general requirements in microservices systems and our field experience in log analysis to pursue several novelties. Differently from many existing tracing tools, such as [8], [9], [10] and [11], our approach is application transparent, non-intrusive and it requires no microservices modifications; moreover, it is resilient to *changes* in the computing environment, such as addition and deletion of microservices.

We present an implementation of our tracing approach called **MetroFunnel**, which is available –both to practitioners and the research community– as a ready-to-use tool on GitHub[2]. Our approach aims to increase the efficiency of both of experts and modestly-skilled practitioners in traversing builtin microservices' logs by means of a supplementary run-time tracing and checking mechanism.

The assessment of MetroFunnel is done in the context of two case studes: a Clearwater IP Multimedia Subsystem (IMS) setup consisting of Docker microservices [12] and a Kubernetes orchestrator deployment managing tens of microservices. In both cases, we investigate some practical scenarios to demonstrate that traces gathered with no knowledge of the application design allow making useful attributions on microservices by means of **actionable evidence**. In both the case studies, we assess the performance overhead and the log file size of MetroFunnel and compared them with builtin logs, under increasing load conditions. Results highlight that, in the worst case (our Clearwater setup), MetroFunnel induces a 3.3% higher performance overhead when compared to the collection of builtin logs, which is reasonably negligible. Results also highlight that the size of MetroFunnel's trace is significantly smaller than traditional logs. In the worst case (again, Clearwater setup), the size reduction accounts for about 59%. Overall, results suggest that a good strategy to reduce the overhead of the on-line monitoring is to use MetroFunnel at runtime, for tracing and early warning, while keeping builtin logs on their respective nodes, to be accessed only when needed for troubleshooting.

The paper is organized as follows. Section 2 presents related work in the area and positions the contribution of our research. Section 3 provides a real-life example to motivate our approach. Section 4 discusses the research statement, requirements, goals and *non*-goals of our work. Section 5 discusses the design choices underlying MetroFunnel and its implementation. Section 6 describes the case studies, while our investigation of the considered practical scenarios is provided in Section 7. Section 8 presents the overhead assessment on both case studies. Section 9 discusses the threats to validity, and Section 10 concludes the work.

2 Related work

Many solutions have emerged for monitoring *microservices*- and *containers*-based systems. We divide them in the areas of **indirect** and **direct** approaches, and we position our research in the context of these areas.

2. https://github.com/dessertlab/MetroFunnel

2.1 Indirect monitoring

Indirect monitoring approaches typically aim to collect *OS*- and *network*-level metrics. One trend has consisted in **porting** to microservices consolidated tools that were originally conceived for networked and Cloud systems. For example, monitoring frameworks, such as the open-source **Nagios** [13] and **Ganglia** [14], and the commercial **Amazon CloudWatch** [15], are now used to collect and evaluate metrics (e.g., CPU utilization, system load and free memory) in clusters *hosting* microservices. It is worth noting that these frameworks were not designed to collect fine-grain monitoring data in the granularity of microservices; as such, dedicated tools have now emerged for indirect monitoring of microservices and containers.

CAdvisor [16] is an open source agent –implemented as a monitoring container– that automatically discovers all the containers running in a system. It collects, aggregates, processes, and exports information about the containers, such as resource usage and network statistics.

Elascale [17] is an approach for auto-scaling and monitoring of Cloud software systems based on Docker [18] microservices. Monitoring is performed through the **ELKB stack**, i.e., Elasticsearch, Logstash, Kibana and Beats [19], which allows collecting performance metrics, e.g., CPU, memory and network usage, for each container. Elascale supports only Docker technology; moreover, it does not provide indications about runtime errors. **Sieve** [20] is a platform to infer insights from monitored metrics in microservices systems. Sieve analyzes the communication between containers hosting the microservices, in order to obtain a call graph and records of all the metrics exposed, such as CPU and memory usage. Sieve leverages **sysdig** [10] to obtain/analyze the communication between components, which requires a kernel module to observe the system calls used by microservices. **ConMon** is a distributed and automated monitoring solution for observing network metrics in container environments [21]. ConMon requires monitoring containers to be deployed inside the target system and needs intervention on the virtual switch to forward packets to the monitoring containers.

2.2 Direct monitoring

Many commercial tools for Application Performance Management (APM) allow monitoring microservices by means of **service instrumentation**, such as Dynatrace [11], AppDynamics [22], CA [23] and New Relic [24].

A specific commercial APM solution for microservices applications is **Instana** [8]. It leverages the **span data** model [25], which is based on *trace trees* where each node represents a service (named the *span*). Instana traces all the requests generated by *properly instrumented* microservices. For each request, the tool provides the total time of the trace and the number of errors occurred during the trace. Errors are captured when it is detected a logging call at `ERROR` severity or upon a bad response by a service. Instana requires application instrumentation to trace the requests.

Among open-source solutions we can find **Zipkin** [9], i.e., the distributed tracer for microservices by Twitter. It is based on services instrumentation (supporting a variety of programming languages) and on the *span data* model.

Zipkin uses collectors on the nodes of the target system to trace and store data. Stored data are accessed by a query service, which provides the completion time of services. Zipkin requires service instrumentation.

Sysdig [10] is a container-native solution, which allows collecting resource usage, network statistics, as well as capturing system calls and tracing applications within containers, such as microservices. System calls from containers are captured by means of a kernel module, while tracing is done by *instrumenting* the applications. Instrumentation consists in writing formatted text strings to `/dev/null`.converts the strings into events that allow obtaining completion times for the instrumented object. [26] presents a dashboard for monitoring and managing microservices. The dashboard is characterized by a Spring-based infrastructure that uses **Dynatrace** [11]. The infrastructure allows collecting both failure rate and response time of each microservice; however, they are collected by means of Dynatrace, which requires the instrumentation of the service to be monitored.

A run-time verification approach for microservices-based applications is presented in [27]. The approach is built on the top of Conductor, i.e., the Netflix microservices orchestrator [28]. It leverages the execution flow of the target application provided by Conductor to generate Time Basic Petri (TBP) nets. TBP nets are leveraged by a run-time verification module based on the MahaRAJA tool [29], which collects the execution trace of the target application *instrumented* with Java annotations.

Netflix **Hystrix** [30] is a latency and fault tolerance Java library designed to prevent cascading failures. It allows near-realtime monitoring of services by means of source-code instrumentation. Hystrix is able to measure successes, failures and timeouts of calls; however, it requires each call to external systems and dependencies of services to be wrapped inside an *HystrixCommand* object, which generates metrics on outcomes and latency.

2.3 Our contribution

Indirect approaches presented in Section 2.1 typically aggregate resource usage statistics at *container*- or *host*-level: as such, differently from our contribution, they cannot support inferences in the *granularity* of microservices. Similar to our contribution, the work discussed in Section 2.2 proposes tracing approaches for microservices; however, services under-monitoring are *actively* involved in the generation of the traces by means of code instrumentation. **Active tracing** is not application transparent and it needs for executing additional code or running extra processes: this entails a degree of intervention on the target system, which might be not suited for microservices developed at different times, by different teams and vendors.

Different from this literature, we contribute with a **passive tracing** approach that requires no microservices instrumentation. *We achieve application transparency of indirect approaches at the data granularity of active tracing*.

With respect to the literature on passive tracing in distributed systems, it is worthy to mention [31] and [32]: this research dates back early 2000 and resulted into *Project 5* and *WAP5*, respectively. The former focuses on local-area distributed systems and capitalizes on network traces; the latter focuses on wide-area distributed systems and uses an *interposition library* for process-level tracing. While we share some technical similarities with this work due to passive tracing, [31] and [32] do not address microservices.

To the best of our knowledge there are few contributions on non-intrusive tracing of microservices. **MONAD** [33] is a self-adaptive microservice infrastructure for heterogeneous scientific workflows. It leverages the subscription model of Advanced Message Queuing Protocol (AMQP), and collects special *Start*/*End* messages that represent invocation/completion of workflows. The approach requires that microservices communicate by means of AMQP; moreover, differently from our proposal, it provides monitoring data in the granularity of *workflows*. The non-intrusive approach in [34] records the calls between microservices and responses with a modified version of the *Zuul* Netflix gateway [35]. We observe that this approach depends on the functionalities of the gateway and its availability in the target network. For example, in a Docker environment, the correct operation of the approach would require an overlay network. Differently from this work, our proposal does not rely on any specific external tool. **Gremlin** [36] is a framework for systematically testing the failure-handling capabilities of microservices, with fault injection. With Gremlin, we share the "touch the network, not the app" principle, as the framework is based on the key insight that failures can be staged by observing the network interactions. However the paper pursues the different objective of microservices testing, that entails the use of traces in controlled and repeatable environments. Furthermore, it does not explore the use of traces to better analyze application logs, as we do. **Hansel** [37] and **Gretel** [38] leverage non-intrusive network monitoring for root-cause analysis in OpenStack. Differently from our work, both papers are tied on the monitored system, namely OpenStack. Hansel analyzes network messages to mine OpenStack's unique identifiers and it leverages them to construct execution trails of events. Gretel builds upon the observation that OpenStack components interact using a finite set of API interfaces. Authors then leverage OpenStack integration tests to create fingerprints for all such APIs, to be used to uncover problems at runtime, in case of deviations from fingerprints.

These studies share with us a similar use of passive tracing of REST messages, but, differently from them, we discuss the implications of such use in the more general context of monitoring and event log analysis in microservices. Our final aim is to show that traces gathered with no knowledge of the application design are helpful to gather actionable evidence that then can be used by practitioners to better traverse and analyze microservices' builtin and heterogeneous logs.

3 Motivating Example

We use a real-life example to motivate our work. The example aims to illustrate (i) ancillary events available across various logs in a microservice-based system and (ii) manual forensics and *guessings* done by human experts in traversing the logs. Data snippets are from a Clearwater IP Multimedia Subsystem (IMS) installation, which consists of Docker

```
 1 (6505550350, 6505550359) Failed
 2 Exception in quaff_cleanup_blk:
 3 — Expected 401|200, got 504
 4  (call ID fc05dd9e8527018fafff19e30b213af1)
 5 Leaked sip:6505550359@example.com, DELETE returned 502
 6 Leaked sip:6505550350@example.com, DELETE returned 502
 7 Basic Call — Unknown number (TCP) at 2018—09—27 12:50:14
 8 Endpoint on 34861 received:
 9 SIP/2.0 504 Server Timeout
10 Via: SIP/2.0/TCP 172.17.0.2:34861;rport=53976;
11    received=10.0.30.88;branch=z9hG4bK1538052594.4184752
12 CSeq: 4 REGISTER
13 from 10.0.30.85:5060 (TCP)
```

Fig. 1: Snippet from the client log.

microservices and will be used later on to assess our implementation. Clearwater provides voice, video and messaging services. The end goal of this example is troubleshooting a failure experienced by a Clearwater client while attempting a *voice telephone call*.

Fig. 1 shows a snippet from the client log, which contains various errors, i.e., (i) a timeout of the server –code `504`– occurred at `12:50:14`, reported at *lines 3*, *7* and *9*; (ii) two `502` responses triggered upon attempting the deletion of the telephone accounts `sip:6505550359@example.com` and `sip:6505550350@example.com` (*lines 5-6*). The most interesting observation is that the log *trivially* indicates at *line 13* that the potential origin of the failure is the server machine `10.0.30.85` (i.e., the IP address of the machine that hosts Clearwater's microservices in our testbed); however, no error message provides specific evidence or context that might help practitioner to make *informed* decisions on how to progress the inspection.

Regarding the timeout error, either an *expert* or a *modestly-skilled* practitioner would now opt to look at the log of the *anchor* microservice, i.e., *bono* in Clearwater IMS, serving as the fronted for the client's connections. Consequently, we manually scrutinized the log of *bono*: surprisingly, beside normative messages (such as status of incoming requests and recycling of TCP slots) we found no error messages that revealed the cause of the timeout.

After having looked at the client and *bono*, we end up with no evidence to move on with the inspection. A hypothetical practitioner is now expected to guess the next steps and formulating hypotheses. After having inspected the architecture of Clearwater, we found out that a microservice called *sprout* is closely related to *bono* and, indirectly, to the client. We manually reviewed the log of *sprout* to search for errors, which led to uncovering the following:

```
27—09—2018 12:49:54 UTC Error hssconnection.cpp:131: Failed
to get Authentication Vector for 6505550350@example.com
```

The timestamp of the message is compliant with the occurrence of the timeout; moreover, the message contains one of the above-mentioned telephone accounts `6505550350@example.com`. Differently from *bono*, the analysis of *sprout* is more fruitful; however, still we found no clear indication on the cause of the `Failed to get Authentication Vector` statement. Again, the practitioner is forced to even more in-depth thinking to progress the analysis. In this respect, it can be noted that the message above is recorded by the source file `hssconnection.cpp`. We went through the system's documentation and inspected the source code to discover that the *hssconnection* component

```
INFO  [StorageServiceShutdownHook] 2018—09—27 12:49:53,673
Gossiper.java:1454 — Announcing shutdown
[...]
INFO  [ACCEPT—/172.18.0.4] 2018—09—27 12:49:55,687
MessagingService.java:1020 — MessagingService has
terminated the accept() thread
INFO  [main] 2018—09—27 12:49:57,386 CassandraDaemon.java:
155 — Hostname: 5a4e42449b0c
INFO  [main] 2018—09—27 12:49:57,423
YamlConfigurationLoader.java:92 — Loading settings from
file:/etc/cassandra/
[...]
INFO  [Thread—2] 2018—09—27 12:50:13,124
ThriftServer.java:136 — Listening for thrift clients...
```

Fig. 2: Snippet from the log of cassandra.

of *sprout* is in charge of interacting with a further microservice of Clearwater called *homestead*. It is worth noting that this step would had been *not* so straight in lack of accurate documentation or the source code.

We thus analyzed the log from *homestead*, which pointed out the following error message:

```
27—09—2018 12:49:54 UTC Error cassandra_store.cpp:541:
Cassandra request failed: rc=3, Exception: socket open()
error: Connection refused [1]
```

that provides a final indication of the error, i.e., a `Connection refused` by *cassandra*, which is used to store authentication credentials and profile information in Clearwater. An extract of the log from *cassandra* shown in Fig. 2, confirms that the microservice has been unavailable in close time proximity to the timeout experienced by the client.

Due to space limitations, the work done in investigating the `502` responses obtained upon the deletion of the telephone accounts is presented more briefly. Again, we hypothesize a set of candidate microservices for deeper inspection based on the system's architecture, i.e., *ellis*, *homer* and *homestead-prov*. While *homer* contains no error messages, *ellis* reports several anomalies relating to the telephone accounts. Examples of the most noticeable are shown hereinafter:

```
27—09—2018 12:49:54.618 UTC WARNING utils.py:53: Non—OK HTTP
response. HTTPResponse(code=500,... effective_url='http://
homestead—prov:8889/public/sip%3A6505550359%40example.com/
error=HTTPError('HTTP 500: Internal Server Error',))
27—09—2018 12:50:14.643 UTC WARNING utils.py:53: Non—OK HTTP
response. HTTPResponse(code=599,... effective_url='http://
homestead—prov:8889/public/sip%3A6505550350%40example.com/
error=HTTPError('HTTP 599: Timeout',))
```

where the accounts `sip:6505550359@example.com` and `sip:6505550350@example.com` experience a `500` (*Internal Server Error*) and a `599` (*Timeout*) form *homestead-prov*. A closer look into *homestead-prov* points out further issues in interacting with *cassandra*, which further confirms the finding of the investigation above.

4 Research Statement and Goals

4.1 Discussion and statement

Traversing logs is not an easy task. It underlies substantial cognitive work by humans in guessing the most reasonable logs to scrutinize at first, hypothesizing step-by-step forensics, finding *pivots* (e.g., timestamps, keywords and values) to pinpoint and correlate relevant messages, and connecting the dots for obtaining the *big picture*. Overall, this process demands for fusing knowledge at different levels.

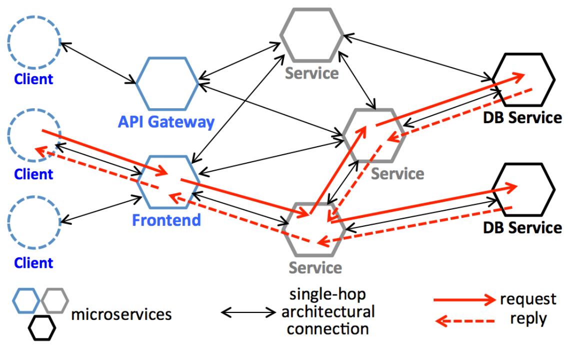


Fig. 3: A fictitious microservice system.

We observe that it is hard to infer the *context* of a service invocation from logs. For example, by context we mean the invoked service and source-destination of the invocation. The context provides valuable hints on where to find problems' symptoms. The manual inspection in Section 3 encompassed 7 out of 15 log files available in our Clearwater setup; however –after having manually checked all the logs– we noticed that only 5 out of the 15 logs reported useful information pertaining the errors. Only the *a-priori* knowledge of the system and interactions among microservices prevented us from digging into many useless logs. It is important to note that the knowledge of the system's internals might not be enough for efficiently traversing the logs. For example, at some point in the motivating example we were able to progress the analysis only by making hypothesis on the source code file. Overall, these issues are exacerbated in microservices systems.

Our **research goal** is to support practitioners in dealing with event logs and their analysis in the context of microservices systems. We present a *black box* approach to this goal, based on passive tracing of the request-response protocol between microservices. To this end, we model a system as a set of nodes and edges, where nodes denote microservices and edges represent single-hop architectural connections between microservices. Fig. 3 shows a fictitious three-tier microservice system, where *API Gateway* and *Frontend* provide anchor points for various clients (e.g., browsers or mobile devices). A potential **execution path** triggered by a service required by a client is superimposed as thick arrows: we aim to collect a trace for such paths, in order to accompany traditional logs.

The availability of a supplementary tracing mechanism can significantly increase the usefulness of logs and the efficiency of both expert and modestly-skilled practitioners in traversing the logs. We will demonstrate that traces gathered at negligible overhead and by means of a technique that embeds no knowledge of the application design allow making useful attributions on microservices requiring deeper inspection in troubleshooting.

4.2 Requirements

We aim to avoid the need to deploy a specific tracing/logging infrastructure; more important, our approach *does not* require any direct support from the microservices level. In consequence, vendors are not expected to spend efforts in supporting a certain methodology. We treat microservices as *black boxes* in order to pursue a ready-to-use implementation. Microservices systems are very flexible and dynamic; as such, a tracing technique should be responsive to changes of the computing environment by providing the following capabilities:

- *uncover new services* as they become available, i.e., tracing interactions from newly-added microservices with no human intervention;
- *application-transparency*, i.e., tracing should progress without any instrumentation of the source code of the microservices;
- *zero configuration (zeroconf)*, i.e., tracing should not rely on heavy configurations, , e.g., the list of services under-monitoring or their physical location, and should not incorporate the knowledge of the application design.

Towards these requirements, passive tracing –that requires no microservices modifications– is aimed to make our approach generally applicable; in fact, a tool that requires architecture-specific knowledge, or microservices modifications, is much less likely to be used.

The requirements above define what makes our approach potentially applicable to a wide range of systems. Another key requirement is to make the approach useful to practitioners in efficiently traversing the logs. Based on our field experience in log analysis, we list some of the context data that a tracing technique should collect for supporting troubleshooting. For example, there should be a clear indication of *invoked services*, accompanied by the *methods* used for the invocations; moreover, the trace should account for *source* and *destination IP addresses/ports*, which allow to pinpoint the nodes hosting microservices, and to discriminate among microservices hosted by the same node or replicas of the same service. Such an information allows practitioners to deal with limited knowledge of the system behavior and deployment. Moreover, the trace should support practitioners by providing evidence about problems occurred in the target system. This is achieved by collecting *response codes* and *completion time* of service requests, to allow faster diagnosis and to narrow the scope of the log analysis.

4.3 Non-goals of our research

We point out some *non*-goals of our research. The approach proposed in this paper:

- *Is not an error detector*. We make inferences based on the *request-response* protocol. Although data collected by our trace –e.g., HTTP response codes, server timeouts and service completion times– cover many error types that might occur in practice, we are not proposing a *one-fits-all* error detector.
- *Does not aim to measure microservices' performance*. Our trace includes an approximation of the time taken by invocations to complete. Although –under some assumptions- this information is reasonably accurate, we can pinpoint abnormal services durations outright, rather than providing precise measurements.
- *Is not meant to replace traditional event logs*. **Log messages are necessary to debug runtime problems**. However, as shown by the motivating example

above, troubleshooting is hard and underlies substantial cognitive work by humans: our goal is to make it easier and faster, not to automate it.

- *Is not a log management tool*. There are plenty of tools for managing and collecting logs; thus, this paper is not the proposal for one more tool. Rather, we generate a lightweight execution trace that accompanies microservices logs. The trace is a sequence of structured records, which makes it straightforward to fit into up-to-date log management frameworks, such as we did in this study.

5 Proposed approach

We present the design space that has been set for instantiating our approach, the format of the trace and the monitoring algorithms for inferring high-level insights on microservices from network packets.

5.1 Design space

The **REpresentational State Transfer** (REST)ful style is the most used for APIs development according to inferences that can be made with the data available at Programmable Web [39].The style is centered on the concept of **resource** on which *create*, *read*, *update* and *delete* operations (also known as CRUD operations) can be performed; network-accessible resources are addressed by means of Uniform Resource Locators (URL). In spite of the long-standing debate on the maturity of REST over SOAP for enterprises, as early as 2010 REST accounted for around 75% in terms of distribution of API protocols and styles [40]. Consequently, we focus here on REST due to its wide adoption over other styles for the communication, such as *asynchronous* communication mechanisms based on queues and topics.

The REST architectural style is very often implemented via HTTP. The different types of HTTP requests, i.e., POST, GET, PUT and DELETE, perfectly match CRUD operations. Example are shown in the following:

```
GET /test/users/1             (read user 1)
GET /test/users               (read all users)
POST /test/users/2            (create user 2)
PUT /test/users/3             (update user 3)
DELETE /test/users/4          (delete user 4)
```

where a test application manages the resource user and allows consumers to create, update, read and delete users data, such as explained for each URL.

Microservices communicate with other microservices and client applications through HTTP requests-responses of above types, which we monitor by *passively* tracing the network packets[3]. **Passive tracing**, i.e., the interception of packets passing through a network, requires no changes to the target software; as such, it is *application-transparent*.

A network packet requires some processing to be useful for inferring high-level information on microservices, such as described in Section 5.3; moreover, our initial target are LAN-based systems: in other cases, packet sniffing can be done at each participant host and traces are merged later on. In this paper we do not focus on these technical aspects because they do not pose novel challenges and have been addressed by previous work, such as [31], [41].

3. At this stage of our work we are not addressing HTTP over SSL.

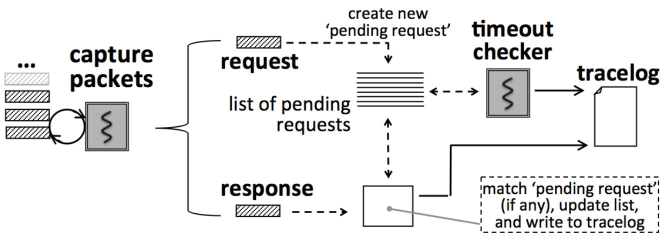

...
capture packets
request
create new 'pending request'
list of pending requests
timeout checker
tracelog
response
match 'pending request' (if any), update list, and write to tracelog

Fig. 4: MetroFunnel's main loop.

5.2 Format of the trace

A trace is a sequence of records that account for the outcome of microservices' invocations, such as timeout expirations, response codes and completion times. The format of the record is:

```
Timestamp, Method, URL, Src_IP, Src_Port, Dest_IP,
Dest_Port, Response_Code, Completion_Time, Info
```

Fields are briefly defined in the following, while Section 5.3 describes the algorithms that generate the records during the progression of the sniffing.

Each record is marked with the *Timestamp*, i.e., the time of the creation of the record, which is followed by *Method* (e.g., GET and PUT), and *URL* of the request. According to the requirements drafted in Section 4.2, we accompany the record with source and destination IPs-ports of the microservices involved in the interaction, i.e., *Src_IP-Src_Port* and *Dest_IP-Dest_Port*.

The record also contains **diagnostic fields** that pertain to the invocation. *Response_code* is the HTTP response code, while *Completion_time* represents the time –measured in milliseconds (*ms*)– between the service request and the corresponding response. Finally, *Info* is a label about the termination of the invocation; it assumes one of three values:

- *Request-Response*: both the HTTP request and response packets are captured for a given invocation within a *reasonable* timeframe. It is noteworthy that a record marked as *Request-Response* does not necessarily indicate that the invocation was correct, because –although the response is received– *Response_code* might be an erroneous HTTP code.
- *Request-TIMEOUT*: an HTTP *request* packet is captured at network level; however, no response is observed for that request within a *reasonable* timeframe.
- *NO_REQUEST-Response*: an HTTP *response* packet is captured at network level much more later than the corresponding *request*.

The following record shows a GET request for the resource /test/users/1, with source 172.18.0.10, 57170 and destination 172.18.0.6, 8888:

```
27-09-2018 09:42:15 UTC, GET, /test/users/1, 172.18.0.10,
57170, 172.18.0.6, 8888, 200, 2.31, Request-Response
```

both the request and response are captured for the invocation (i.e., *Request-Response*); moreover, the invocation is correctly accomplished with response code 200 within 2.31*ms*.

5.3 Tracing algorithms

We describe the algorithms that allow inferring a high-level trace from network packets. Fig. 4 provides an overview of the tracing approach, which consists of two threads: *capture packets* and *timeout checker*.

Capture packets continuously accepts new incoming network packets. For each packet, the algorithm run by *capture packets* checks whether it contains an HTTP header, beforehand: *if not*, the packet is discarded. In case of an HTTP packet, the algorithm proceeds by extracting low-level fields, such as source-destination IP addresses (IP header) and source-destination TCP ports (TCP header). The HTTP header is also scanned to verify if the packet denotes a *request* or *response*. This is done by accessing the first bytes of the header: if it starts with one of the protocol methods, it means it is a request (e.g., `GET /test/users/1 HTTP/1.1` or `POST /test/users/2 HTTP/1.1`); otherwise, if it starts with "HTTP" it means it is a response (a typical response header is: `HTTP/1.1 200 OK`, where 1.1 is the version of the protocol, 200 is the response code and `OK` the corresponding message) [42].

In case of a **request**, the algorithm appends an entry to an **internal list** –depicted in Fig. 4– that contains the *pending requests*, i.e, requests *to be decisioned* by our algorithms. The entry in the list is marked with the *current time*, which denotes the **arrival time** of the request. Please note that each pending request in the list is eventually decisioned by our algorithms to be (i) *expired* or (ii) *terminated* (either correctly or not) as explained hereinafter: once a decision is taken, a new record is appended to the trace.

5.3.1 Expiration of a request

The **timeout checker** thread runs the following algorithm. It periodically scans the list of *pending requests*: at each scan, for each entry that is stored in the list, the thread computes the difference between (i) *current time* and (ii) the time the entry was *inserted* in the list (i.e., the *arrival time* mentioned above). If the difference is higher than a timeout[4] the request is decisioned to be **expired** and it is deleted from the list. Upon the deletion, the algorithm generates a record that is appended to the trace, such as the following example:

```
26−09−2018 10:32:56 UTC, GET, /test/users/1, 172.18.0.10,
57170, 172.18.0.6, 8888, 999, 60010.60, Request−TIMEOUT
```

where the *Info* field is *Request-TIMEOUT*; moreover, the HTTP response code is assigned a fictitious `999` value to denote that the request is expired.

5.3.2 Termination of a request

Upon the receipt of a **response**, the algorithm executed by **capture packets** accesses the list and attempts to match the corresponding request, *if any*, based on the IP address and TCP port pair. Matching is based on the following assumptions: for the HTTP 1.0 standard each request corresponds to a response before a new request is initiated; starting from HTTP 1.1 –although it is allowed to *pipeline* multiple requests– the order of responses is the same as the requests. If the match is fruitful, the request is deleted from the list and a new record is appended to the trace. For example, if the response is correct the record will resemble the following, with response code 200 (correct service):

4. The timeout is an input parameter of the algorithm, and it set by the user at the boot of the capture.

```
26−09−2018 15:42:16 UTC, GET, /test/users/1, 172.18.0.10,
57170, 172.18.0.6, 8888, 200, 51.47, Request−Response
```

A response might account for an incorrect service, which means that it is accompanied by an erroneous HTTP code. As such, the record in the trace will appear as the following example (error code `502`):

```
26−09−2018 16:13:26 UTC, GET, /test/users/1, 72.18.0.10,
57170, 172.18.0.6, 8888, 502, 7564.33,  Request−Response
```

It is important to note that the matching step might be occasionally unfruitful, i.e., although it is captured a response, there is no corresponding request in the list. This case is observed when a request is declared *expired* by the *timeout checker* (and thus deleted from the list) but its corresponding response shows up later than the timeout. In this case, the algorithm generates a record marked as *NO_REQUEST-Response*, such as the following:

```
NULL, NULL, 127.0.0.1, 46594, 127.0.0.1,
8080, 502, NULL, NO_REQUEST−Response
```

which is usually preceded by a *Request-TIMEOUT* record in the trace, such as the one shown in Section 5.3.1; if needed, *reconciliation* of *NO_REQUEST-Response*s can be done when post-processing the trace.

Whenever either (i) a response is received (with a correct or erroneous response code) and the matching is fruitful, or (ii) the timeout is expired, we compute the **completion time** of the invocation as the difference between the *current time* and the *arrival time* of the related request (the arrival time is taken at the insertion of a request in the "pending list" as stated above). The completion time is included in the records of the trace; otherwise, if the matching step is unfruitful the completion time is set to `NULL`.

It should be noted that in the first case –i.e., the response is received and the match is fruitful– the completion time provides an approximation of the execution time of the service invocation. While this information is useful under certain hypothesis (e.g., negligible transmission delay or network congestion, such as within LAN-based environments) to pinpoint abnormal services durations, our aim is not to develop a performance evaluation tool, such as clearly stated in Section 4.3.

5.4 Implementation

We describe an implementation of our tracing approach called **MetroFunnel**, which is a multithreaded Java program aimed at capturing the network packets and generating the trace. Due to space limitations, we briefly introduce the key components and main capabilities of MetroFunnel, which is available at the GitHub link provided in Sect. 1.

The **PacketSniffer** component is responsible for *capturing, filtering*, and *inferring* HTTP messages' fields. The sniffer operates in promiscuous mode. We reviewed a number of Java sniffing frameworks, such as *Pcap4J*, *jpcap* and *jNetPcap*, which wrap the libpcap/Winpcap library. We use **jNetPcap** [43], which is based on Java Native Interface rather than Java Native Access, such as the other solutions, for better performability. The **LogManagement** package is responsible for writing the trace to a file; moreover, it implements the tracing algorithms, including request-response matching and timeout checking. The 1.4r1425 version for 64-bit Linux systems and JDK Java SE 8u151 have been used for jNetPcap and Java, respectively.

As mentioned above, MetroFunnel is *ready-to-use* and requires no system-specific configuration. At the boot of the program, the user is simply required to customize the capture by setting the *network interface(s)* and the *expiration timeout*. It should be noted that despite the expiration timeout is an application-dependent parameter, its value can be easily obtained through MetroFunnel itself. In fact, MetroFunnel allows obtaining an approximation of the time taken by service invocations to complete, which can be used to set the timeout parameter. MetroFunnel is available both as *standalone* Java application and as Docker image.

6 Case Studies

We test MetroFunnel in the context of two case studies, i.e., the **Clearwater** IMS setup consisting of Docker microservices and the **Kubernetes** setup running tens of replicas of a webserver microservice, presented in the following.

6.1 Clearwater IMS

Clearwater is an open-source IMS core. IMS is the standard architecture adopted by large telcos for IP-based *voice*, *video* and *messaging* services. We use a version of Clearwater consisting of 11 microservices distributed in as many Docker containers. We describe some of the microservices closely referenced by this paper, while a comprehensive architectural view of Clearwater can be found at [12]:

- **bono** is a Session Initiation Protocol (SIP) edge proxy. It provides a WebRTC interface for handling multimedia sessions. It represents the *anchor* point of the clients' connection to Clearwater;
- **sprout** is the SIP router of Clearwater, which handles client authentication;
- **cassandra** is the database to store authentication credentials and profile information;
- **homestead** is a C++ RESTful CRUD server on the top of *cassandra* and provides an interface for retrieving credentials and user profile information;
- **homestead-prov** exposes an interface to allow provisioning subscriber data in *cassandra*;
- **homer** is a standard XDMS (XML Document Management Server) used to store MMTEL (MultiMedia TELephony) service settings documents;
- **ellis** is a provisioning portal for self sign-up, line management and control of MMTEL service settings.

Above microservices expose HTTP-based REST interfaces. We use **Clearwater-live-test** [44] to generate a load for exercising Clearwater. *Clearwater-live-test* is a well-consolidated suite of *Ruby* programs, meant to check that a deployment is working correctly and used by the Project Clearwater team to validate that newly added functions work end-to-end. The tests that we used for our experiments share a similar pattern: (i) *test setup*, i.e., registering a certain number of telephone accounts; (ii) actual sequence of service invocations to Clearwater (number/type of invocations vary across the tests, such as call and waiting call between endpoints, call cancelation); (iii) *test finalization*, which deletes the telephone

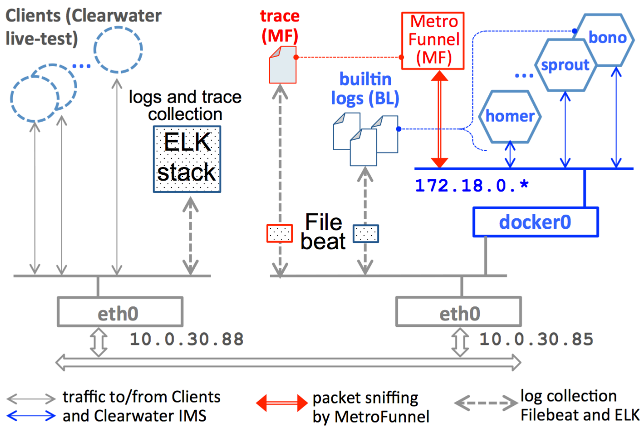

Clients (Clearwater live-test)
logs and trace collection
ELK stack
trace (MF)
Metro Funnel (MF)
builtin logs (BL)
bono
sprout
homer
172.18.0.*
docker0
File beat
eth0
10.0.30.88
eth0
10.0.30.85
traffic to/from Clients and Clearwater IMS
packet sniffing by MetroFunnel
log collection Filebeat and ELK

Fig. 5: Clearwater experimental setup.

accounts. The test suite emulates realistic usage scenarios, including corner cases such as: call rejected by an end-point, call to an unreachable user, call with no response, etc.

Clearwater microservices are hosted by an Ubuntu 16.04.03 LTS OS, Intel i3-2100 3.1GHz (with 2 cores) server. Each microservice is charactered by a unique IP address; microservices are connected through the *docker* LAN `172.18.0.*` shown in Fig. 5. The *docker* LAN is bridged with the physical LAN `10.0.30.*` through the `eth0` interface of the server: external clients –hosted by a different node– run instances of *Clearwater-live-test* and are allowed to reach the microservices. **MetroFunnel** *sniffs* the packets traversing the network `172.18.0.*`, again, the LAN hosting the microservices.

Clearwater's **builtin logs** (BL) and the **trace** by MetroFunnel (MF) are stored in a disk partition at the server side. We use *Filebeat* [45] to (i) read, on-line, new lines of log/trace from the files as they are generated during system operations, and (ii) forward the lines to a *Logstash-Elasticsearch-Kibana* (ELK) stack [19] –shown in Fig. 5– for archiving the logs at a central location. The stack encompasses Logstash for parsing, Elasticsearch for indexing and saving the logs and Kibana for visualization purposes.

Although many of the considerations made in this paper go beyond the availability of a specific *log management* infrastructure, we emulate a typical setup in distributed systems management based on the Filebeat-ELK infrastructure that encompasses a centralized log server. Please note that ELK is used in other container-based monitoring solutions, such as [17]; however, any other log management framework would have fit the aim of the assessment.

6.2 Kubernetes orchestrator

Kubernetes (aka k8s) [46] is a portable, extensible and open-source orchestrator for managing containerized workloads and services, which facilitates both declarative configuration and automation. Kubernetes provides a framework to run distributed systems resiliently, allowing the management of scaling requirements, failover, deployment patterns. Applications are run inside *pods*, i.e., the basic execution unit of Kubernetes, which can be composed by one or multiple containers. Pods can be easily replicated through the usage of *ReplicationControllers*, which make sure that a pod or a homogeneous set of pods is always up and available. Applications running on a set of pods are usually deployed

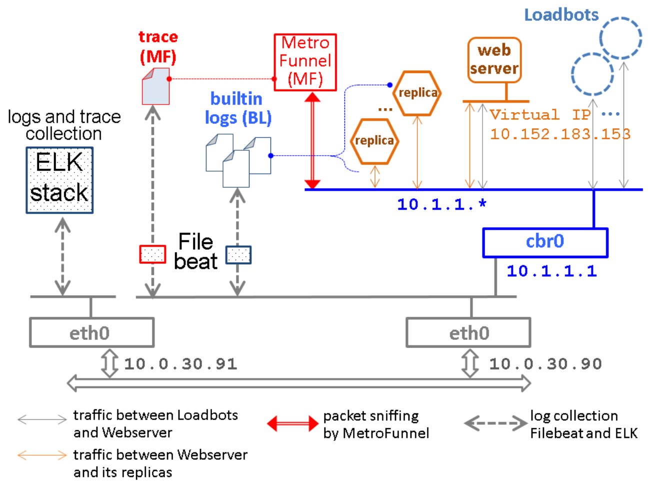

trace (MF)
Metro Funnel (MF)
Loadbots
web server
replica
...
replica
builtin logs (BL)
logs and trace collection
ELK stack
Virtual IP 10.152.183.153
...
10.1.1.*
cbr0
10.1.1.1
File beat
eth0
10.0.30.91
eth0
10.0.30.90
traffic between Loadbots and Webserver
traffic between Webserver and its replicas
packet sniffing by MetroFunnel
log collection Filebeat and ELK

Fig. 6: Kubernetes experimental setup.

as *services* in Kubernetes, which allows masking the set of pods to other applications; each service usually receives a virtual IP that can be used by clients. Services are expected to create a *proxy* on the bridge interface of the node where they are running; the proxy - hereinafter named *load balancer* – balances the traffic towards the service.

We use a lightweight and local kubernetes cluster, named *Microk8s* [47]. MicroK8s is a CNCF (Cloud Native Computing Foundation) certified Kubernetes deployment that can run locally on a single workstation. We deploy a test suite [48] on our setup, made of three types of microservices:

- **loadbot**, a client generating requests to a service, based on the Vegeta HTTP load testing tool [49];
- **webserver**, a microservice that responds to loadbots. It is deployed as a service in Kubernetes, along with a replication controller; the service masks a number of replicas running in different pods;
- **aggregator**, the orchestrator of the test suite; it scales up and down the number of replicas/loadbots and collects statistics about the test suite execution, such as request success rate and mean latency.

The test suite can be configured to run different load scenarios, by varying both the number of *webserver* replicas and *loadbots*. The requests generated by *loadbots* are received by the *webserver*; then, the *load balancer* forwards each request to one of the replicas. It should be noted that the testbed allows to stress different aspects of Kubernetes, i.e., applications deployed as services, replication controllers, load balancer, etc., under different load conditions. More important, it allows to test MetroFunnel at scale, in scenarios with several microservices under load balancing and replication.

Fig. 6 depicts the experimental setup of the Kubernetes case study. The Kubernates cluster has been deployed on a Xubuntu 16.04.03 LTS OS, 2 x Intel Xeon E5-2630L v3 2.9GHz (with overall 16 cores), 16 Gb RAM server. Each microservice is reachable through a unique IP address, e.g. `10.1.1.*`, on the *cbr0* network interface, i.e., `10.1.1.1`. The *webserver* has a Virtual IP address, i.e., `10.152.183.153`. The *cbr0* LAN is bridged with the physical LAN `10.0.30.*` through the `eth0` interface of the server, which is used to reach the node hosting the ELK stack. **MetroFunnel** *sniffs* the packets traversing the network `10.1.1.*`, i.e., the LAN hosting the microservices.

```
ID Record
1  27−09−2018 12:49:55 UTC,GET,/impi/6505550350%40example.com
   /av?impu=sip3A6505550350%40example.com&server−name=sip%3A
   sprout%3A5054%3Btransport%3Dtcp,172.18.0.10,57170,
   172.18.0.6,8888,504,0.68,Request−Response
2  27−09−2018 12:49:55 UTC,GET,/public/sip%3A6505550359%40
   example.com/associated_private_ids,172.18.0.12,34610,
   172.18.0.7,8889,500,0.79,Request−Response
3  27−09−2018 12:49:55 UTC,DELETE,/accounts/live.tests@
   example.com/numbers/sip%3A6505550359%40example.com,
   10.0.30.88,58980,172.18.0.12,80,502,2.49,Request−Response
4  27−09−2018 12:49:57 UTC,GET,/public/sip%3A6505550350%40
   example.com/associated_private_ids,172.18.0.12,34612,
   172.18.0.7,8889,999,2035.91,Request−TIMEOUT
5  27−09−2018 12:49:57 UTC,DELETE,/accounts/live.tests@
   example.com/numbers/sip%3A6505550350%40example.com,
   10.0.30.88,58982,172.18.0.12,80,999,2037.43,
   Request−TIMEOUT
```

Fig. 7: Records from the motivating example.

Both **builtin logs** (BL) generated by the *webserver* replicas running on the top of the Kubernetes cluster and the **trace** by MetroFunnel (MF) are stored in a disk partition on the cluster. As in the Clearwater setup, we use *Filebeat* to collect log lines and forward them to the ELK stack hosted on the `10.0.30.91` machine with the aim to archive logs at a central location.

7 Informed Decisions with MetroFunnel

Assessment focuses on two aspects: (i) given that our approach can reveal anomalies that reflect into the *request-response* protocol, this information is useful to practitioners for traversing the logs of microservices; (ii) the approach implementation does not impact performance significantly.

The first aspect is an assertion about *what* a human expert would find it useful in carrying out a cognitive task: while we stem from our experience in log analysis and provide practical examples of log decisions informed by MetroFunnel in the following sections, its validity is left to the reader's intuition because it is hard to find an *objective* method to quantify it. Regarding the second point, we use traditional metrics in computer systems evaluation, such as *performance* and *log size overhead*, which will be evaluated in Section 8.

7.1 Reflections on the motivating example

We start by discussing how MetroFunnel could have helped with the motivating example presented in Section 3, which refers to our Clearwater setup. As a reminder, the example encompassed a timeout of the server side (`504` response code) and two `502` errors upon attempting the deletion of telephone accounts. Fig. 7 shows some of the records generated by MetroFunnel during the same error scenario. *Record-ID 1* relates to the timeout, where a request to `172.18.0.6` (i.e., the IP address of the *homestead*) terminates with a `504` response. This record suggests practitioners that *homestead* is a reasonable candidate for deeper inspection. Please note that in Section 3 we ended up to the log of *homestead* after (i) several fruitful and unfruitful log investigations, which involved *client*, *bono* and *sprout*, and (ii) strong hypothesis on the system architecture and source code. On the contrary, MetroFunnel provides *ready-to-use* evidence –i.e., requiring no peculiar expertise– to start the analysis.

```
ID Record
1  28−09−2018 10:42:24 UTC, PUT,/private/6505550742%40example
   .com 172.18.0.12,48260, 172.18.0.7,8889,502,0.43,
   Request−Response
2  28−09−2018 10:42:24 UTC GET,/private/6505550742%40example
   .com/associated_implicit_registration_sets 172.18.0.12,
   48262,172.18.0.7,8889,502,0.44,Request−Response
3  28−09−2018 10:42:25 UTC PUT,/org.etsi.ngn.simservs/users/
   sip%3A6505550742%40example.com/simservs.xml, 172.18.0.12,
   35254,172.18.0.8,7888,200,0.26,Request−Response
4  28−09−2018 10:42:26 UTC GET,/public/sip%3A6505550742%
   40example.com/associated_private_ids,172.18.0.12,48266,
   172.18.0.7,8889,502,0.23,Request−Response
```

Fig. 8: Records generated by MetroFunnel during the crash.

Client errors in deleting the telephone accounts are reported by MetroFunnel at *record-ID 3* and *5*, respectively in Fig. 7. It should be noted that the latter is a timeout error by MetroFunnel (*record-ID 5*). This is consistent with the error notifications that we found in the log of *ellis* in Section 3, where the accounts `sip:6505550359@example.com` and `sip:6505550350@example.com` had a `500` (*Internal Server Error*) and a `599` (*Timeout*) error, respectively. In both *record-ID 3* and *5*, the service destination is `172.18.0.12`, i.e., the *ellis* microservice; more important, these records are preceded by *record-ID 2* and *4*, respectively, which further point out to errors involving the destination `172.18.0.7`, i.e., the *homestead-prov* microservice. Again, differently from the time-consuming work that we did in Section 3, MetroFunnel provides a clear picture on the microservices potentially related to errors and, in turn, evidence to drive informed decisions for forensics.

7.2 Crash of a microservice in Clearwater

This scenario encompasses the crash of the process *homestead-prov* –belonging to homonymous microservice container in our Clearwater setup– during the execution of the test suite. The following error is displayed by the client:

```
Account creation failed with HTTP code 502, body {"status":
502, "message": "Bad Gateway", "reason": "Upstream request
failed", "detail": {"Upstream error": "502","Upstream URL":
"http://homestead−prov:8889/private/6505550742%
40example.com"}, "error": true}
```

where it can be found an `Upstream request failed` message and response code `502`. Fig. 8 shows some of the records produced by MetroFunnel during the same setting. It can be noted that for the *record-ID 1*, *2* and *4* the response code is `502`; moreover, in all these records the interaction is between `172.18.0.12` and `172.18.0.7`, i.e., *ellis* and *homestead-prov*. This information can be leveraged by practitioners to go straight to potential candidates for deeper inspection without error-prone reasoning or specific hypothesis, such as with traditional logs. Although we do not list all the events from the logs due to space limitations, the inspection of *ellis* pointed out to several *Non-OK HTTP response* and *Bad Gateway* error messages upon interacting with *homestead-prov*, which thus confirms the usefulness of the attribution done with MetroFunnel.

7.3 System overload in Clearwater

In establishing the factors used for the assessment conducted in Section 8.1, we observe that –in our setup– Clearwater becomes unavailable with a load of 15 *concurrent* clients. This failure is reported by the clients with several error lines, such as:

```
ID Record
1  03−09−2018 14:34:20, PUT, /org.etsi.ngn.simservs/users/
   sip%3A6505550490%40example.com/simservs.xml, 172.18.0.12,
   52844, 172.18.0.8, 7888, 999, 9210.06, Request−TIMEOUT
2  03−09−2018 14:34:20, DELETE, /private/6505550086%40
   example.com, 172.18.0.12, 38582, 172.18.0.7, 8889, 999,
   9209.54, Request−TIMEOUT
3  03−09−2018 14:34:21, DELETE, /private/6505550036%
   40example.com,172.18.0.12, 38586, 172.18.0.7, 8889, 999,
   9209.06, Request−TIMEOUT
4  03−09−2018 14:34:26 UTC, DELETE, /accounts/live.tests@
   example.com/numbers/sip%3A6505550664%40example.com,
   10.0.30.88,50094,172.18.0.12,80,999,8971.87,Request−TIMEOUT
5  03−09−2018 14:34:28, POST, /accounts/live.tests@example
   .com/numbers/,10.0.30.88, 33282, 172.18.0.12, 80, 999,
   8192.98, Request−TIMEOUT
```

Fig. 9: Records from the Clearwater overload scenario.

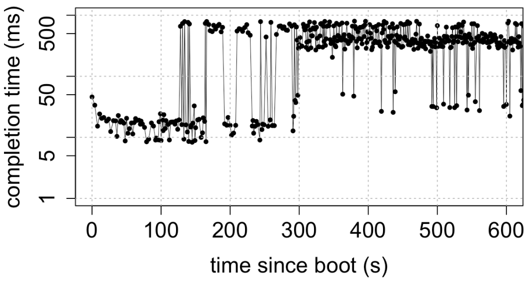


Fig. 10: Completion time of one example service by *ellis*.

```
RuntimeError thrown: Account creation failed with HTTP code
504, body <head><title>504 Gateway Time−out</title></head>
```

This is a *spontaneous* failure, which can be reproduced systematically with our testbed. Different from the previous case, the error notification by the clients says almost nothing about the location of the failure, which makes it hard to come up with effective hypothesizes for troubleshooting.

During the overload, MetroFunnel produces several *timeout expiration* records, such as the ones shown in Fig. 9. Timeouts are the outcome of a progressive performance depletion that affect Clearwater's services. For example, Fig. 10 shows the **completion time** (y-axis, log scale) –collected by MetroFunnel– of the service `POST /accounts/live.tests@example.com/numbers/` offered by *ellis* since the boot of the system (x-axis).

Records in Fig. 9 provide two clear indications: (i) some errors pertain the interactions between `10.0.30.88` and `172.18.0.12`, i.e., the client machine and *ellis*, such as *record-ID 4* and *5*; (ii) these records are preceded by timeout notices pertaining the interactions between *ellis* and either *homer* (`172.18.0.8` - *record-ID 1*) or *homestead-prov* (`172.18.0.7` - *record-ID 2* and *record-ID 3*).

We thus looked at *ellis*, *homer* and *homestead-prov* as candidates for deeper inspection. For example, by following the indication by MetroFunnel, we found out that *ellis*'s log contains several *Non-OK HTTP response* messages, such as:

```
03−09−2018 14:34:20, WARNING utils.py:53: Non−OK HTTP
response. HTTPResponse(code=599,request_time=30.03533101
081848,buffer=None,_body=None,time_info={}, request=
<tornado.httpclient.HTTPRequest object at 0x7fbb69be0710>,
effective_url='http://homesteadprov:8889/public/
sip3A650555099640example.com/associated_private_ids',
headers={},error=HTTPError('HTTP 599: Timeout',))
```

where it can be noted that the HTTP response code received from *homestead-prov* is 599, typically used to notify a network connection timeout. Again, MetroFunnel allowed pointing to a relevant log for the analysis without the need for formulating elaborated hypothesis.

7.4 System overload in Kubernetes

During the setup of the test suite used as load generator for our Kubernetes case study, we notice that the system exhibits an overload –in our deployment– when the test suite is configured with 50 *loadbots* and 5 *webserver* replicas. The overload can be noted by looking at the statistics reported by the *aggregator* (the test suite orchestrator):

```
Success: 93.97%    Latency (mean): 520.20ms
```

These results highlight that the *webserver* running on our Kubernetes deployment has not been able to manage all the requests from the *loadbots*, with about 6.03% of failed requests. The analysis of the logs generated by the *webserver* replicas did not allow pinpointing the cause of such behavior. The replicas only report the received requests and the responses, as shown in Fig. 11, which do not allow to spot the loss in the request success rate. On the other hand, MetroFunnel reported several *timeout expiration* records, such as the ones in Fig. 12. As in the Clearwater overload scenario, timeout records indicate that there is a performance depletion affecting the *webserver*; however, they also indicate that a number of requests did not receive any response, motivating the loss in the request success rate. It is important to note that MetroFunnel reports requests/responses from both the *loadbots* to the *webserver* and from the *load balancer* to the *webserver* replicas. A deeper inspection of the MetroFunnel records allows understanding that the timeouts did not affect only the *webserver* replicas, but also the *load balancer*. For instance, from *record-ID 1* and *2* in Fig. 12, it can be seen that the timeout of the request from the *load balancer* (`10.1.1.1`)[5] to the *webserver* replica (`10.1.1.123`) led to the timeout of the request from the *loadbots* (`10.1.1.143`) to the *webserver* (`10.152.183.153`). However, some of the requests received by the *load balancer* do not reflect into requests to replicas, as it can be seen in *record-ID 3*, *4* and *5*, where only timeouts from *loadbots* requests are reported. Differently from *record-ID 1*, *record-IDs 3-5* have no counterparts at the replicas' side, highlighting the loss of requests by the *load balancer*, which contributes to the 6.03% of failed requests. It is important to note that this is a valuable piece of information for practitioners, which would have not been discovered without the trace of MetroFunnel, since: (i) no logs are generated by the *load balancer*, and (ii) *webserver* replicas do not log anything about requests lost by the *load balancer*.

To mitigate the overload in our setting, we scale up the number of *webserver* replicas to 50, thus achieving a success rate of 99.39%. In this experiment we run MetroFunnel at scale by recording the communication involving around 100 Kubernetes pods/containers, with load balancing and replication capabilities.

5. Please note that the *load balancer* has the same IP address of the *cbr0* interface (see Fig. 6) since it is a service proxy deployed on that interface, as described in Section 6.

```
2019/06/27 12:45:56 Request:
2019/06/27 12:45:56 GET / HTTP/1.1
Host: 10.152.183.253
Accept—Encoding: gzip
User—Agent: Go—http—client/1.1

2019/06/27 12:45:56 Response:
2019/06/27 12:45:56 {
    ”name”: ”simple—webserver”,
    ”date”: ”27 Jun 19 12:45 UTC”,
    ”response”: ”OK”}
```

Fig. 11: Webserver logs from the K8s overload scenario.

```
ID Record
1  27—06—2019 12:45:09 UTC,GET,/,10.1.1.143,55365,
   10.152.183.153,80,999,2000.239896,26,36,Request — TIMEOUT
2  27—06—2019 12:45:09 UTC,GET,/,10.1.1.1,60892,
   10.1.1.123,80,999,2000.559095,27,34,Request — TIMEOUT
   ...
3  27—06—2019 12:47:06 UTC,GET,/,10.1.1.140,48813,
   10.152.183.153,80,999,2103.534883,86,20,Request — TIMEOUT
4  27—06—2019 12:47:06 UTC,GET,/,10.1.1.172,48823,
   10.152.183.153,80,999,2096.521841,92,19,Request — TIMEOUT
5  27—06—2019 12:47:06 UTC,GET,/,10.1.1.178,53935,
   10.152.183.153,80,999,2095.147769,90,18,Request — TIMEOUT
```

Fig. 12: MetroFunnel records from K8s overload scenario.

8 Overhead Assessment

In this section we assess the overhead introduced by Metrofunnel and compare it with the one caused by the collection of microservices' builtin logs. We focus on the overhead observable on the target system in terms of *performance* penalty and *size* of the produced log, which then reflects on the amount of information to be stored or transmitted over the network.

8.1 Performance overhead

To measure the performance overhead of MetroFunnel we setup a rigorous *experimental design* approach. In particular, for each case study, we identify a *response variable* and *controllable factors*, and conduct a set of experiments based on the well-consolidated procedures and recommendations available in [50]. In the Clearwater assessment, the response variable is the **test suite duration** (TSD), i.e., the time (measured in seconds) taken by the tests within *Clearwater-live-test* to complete. TSD summarizes the capability of our Clearwater setup at handling service requests. In the Kubernetes study, the response variable is the **mean request latency** (MRL), i.e., the mean time between a request and a response, which is provided by the *aggregator* of the used Kubernetes test suite. In both studies, the response variable is assessed with respect to the following controllable factors: (i) **load** (LD), and (ii) **type of log** (TL).

LD is the number of *concurrent* clients generating requests to exercise the target system. In the Clearwater system LD represents the number of clients executing the *Clearwater-live-test* independently, while in Kubernetes it represents the number of *loadbots* generating requests toward the *webserver* microservice. The *levels* of LD –i.e., the values LD can take according to the experimental design terminology– are established by conducting a **stress test**, which consists in running the system with an increasing number of clients. We observe that in our setup Clearwater becomes unavailable as we approach 15 clients. As such,

TABLE 1: Clearwater performance overhead results.

	NL	BL	MF	BL+MF	O_{BL}	O_{MF}	Δ	O_{BL+MF}
	avg. test suite duration (s)				%	%	%	%
LOW	665.4	677.4	664.8	679.4	1.8	≈0	-1.8	2.1
MEDIUM	853.2	868.8	881.4	895.9	1.8	3.3	1.5	5.0
HIGH	1,554.5	1,596.7	1,648.2	1,699.1	2.7	6.0	3.3	9.3

TABLE 2: Kubernetes performance overhead results.

	NL	BL	MF	BL+MF	O_{BL}	O_{MF}	Δ	O_{BL+MF}
	avg. request latency (μs)				%	%	%	%
LOW	536.2	565.1	560.9	615.5	5.4	4.6	-0.8	14.8
MED.	534.9	622.1	565.0	642.9	16.3	5.6	-10.7	20.2
HIGH	626.2	741.6	677.4	868.3	18.4	8.2	-10.2	38.7

we conduct our assessment within 1-12 clients[6]. Moreover, as usually done in many empirical assessments, we categorize LD through a smaller number of *actionable* classes -i.e., LOW, $MEDIUM$ and $HIGH$, denoting 2, 6, and 12 clients, respectively- which can be easily applied and understood by practitioners [50]. Similarly, we observe that our Kubernetes setup is overloaded when using around 50 *loadbots*. Therefore, we conduct our assessment considering 30 as maximum number of *loadbots*, and use the LOW, $MEDIUM$ and $HIGH$ classes to denote the scenarios encompassing, 1, 3 and 30 *loadbots*.

The levels assumed by TL vary within $\{NL, BL, MF, BL + MF\}$, which denote the collection of *no log* (NL), builtin logs (BL), i.e., the logs generated by Clearwater or by the *webserver* replicas running on Kubernetes, trace by MetroFunnel (MF), and collection of both builtin logs and MetroFunnel's trace at the same time ($BL + MF$). As mentioned above, logs are handled with Filebeat-ELK; we start/stop the Filebeat components shown in both Fig. 5 and Fig. 6 to enabled/disable the transmission of either BL or MF to the ELK node during the experiments.

In the following, we discuss the experiments done with the standalone Java console version of MetroFunnel because there was no statistically significant difference in the performance metrics with respect to the Docker version. Please note that, as shown in both Fig. 5 and Fig. 6, MetroFunnel and the target system microservices are hosted by the *same* machine, otherwise the assessment would had been too favorable to our proposal. As such, measurements presented hereinafter represent an *upper bound* for what it can be expected in practice because MetroFunnel can be conveniently deployed at a dedicated machine. We adopt a **full factorial design** with repetitions, which means that we run experiments for all the combinations in $LD \times TL$.

Clearwater. TABLE 1 reports the average *test suite duration* (TSD) with LOW, $MEDIUM$ and $HIGH$ load for NL, BL and MF, as well as the **overhead** caused by the type of log. Overhead is computed as follows. Given a level of LD, let us denote by (i) TSD_{NL} the test suite duration measured when collecting no log (NL), and (ii) TSD_{BL} the test suite duration obtained under the collection of Clearwater's builtin logs (BL). The percentage overhead induced by BL is given by: $O_{BL} = \frac{TSD_{BL} - TSD_{NL}}{TSD_{NL}} \cdot 100$. The overhead of MetroFunnel O_{MF} is computed by using TSD_{MF} (i.e., the test suite duration with the collection of MetroFunnel's trace), instead of TSD_{BL} in the overhead equation. Similar considerations apply to the computation of the overhead obtained by combining both builtin logs and MetroFunnel, i.e., O_{BL+MF}, for which TSD_{BL+MF} is used in the equation.

We observe that the collection of builtin logs induces an overhead of 2.7% at *HIGH* load. There is no appreciable overhead of MetroFunnel at LOW load. Overhead is 6.0% at $HIGH$ load and MF. The Δ column of TABLE 1 shows the difference between the overhead of MF and BL. It can be noted that at the maximum load $HIGH$, O_{MF} is around 3.3% more than O_{BL}, which means that –in the worst case– the overhead of supplementing logs with our *black box* execution trace is reasonably negligible. This is confirmed also by the overhead obtained collecting both BL and MF, shown in the right most column of TABLE 1, which is quite around the sum of O_{BL} and O_{MF}.

Kubernetes. TABLE 2 reports the MRL, i.e., the *mean request latency*, obtained with different combination of LD and TL in our Kubernetes deployment, along with the **overhead** caused by the type of log (which has been computed as in the Clearwater study). It can be noted that in our Kubernetes setup the collection of MetroFunnel trace induces an overhead (O_{MF}) that is lower than the one caused by the collection of builtin logs (O_{BL}), i.e., the logs generated by the *webserver* replicas. While their overhead is quite comparable with LOW load, i.e., 5.4% and 4.6% for BL and MF, respectively, the difference becomes more evident when considering MEDIUM and HIGH load. In fact, at maximum load O_{BL} is around 10.2% more than O_{MF}, as shown in the Δ column of TABLE 2. This is also confirmed by the overhead obtained by collecting both data sources, i.e., O_{BL+MF}, which exhibits values quite around the sum of the single overhead, as in the Clearwater case study, except for HIGH load, where the overhead is around 39%. We argue that, in this case, the performance is also affected by Filebeat, that has to cope with multiple data sources at high load. The combined overhead suggests that a good strategy to improve the performance is to process and collect only MF logs at runtime, for monitoring and early warning. BL logs could be kept on their respective nodes (managed with log rotation, as usual) and they would be accessed only when needed, for troubleshooting, guided by the initial attributions made through MF logs.

Comparing the results obtained in the Kubernetes case study with the Clearwater one, we can note that the overhead introduced by MetroFunnel is lower, with respect to BL logs, in the Kubernetes case. This can be explained by both considering the differences in the hardware configuration and the log size. About the configuration, Clearwater runs on a two-core processor with Solid-State Drive (SSD), while Kubernetes runs on a 16-core processor with magnetic disks. Since MetroFunnel is cpu-bound, it takes advantage of the multiple cores available on the Kubernetes deployment; on the other hand, the presence of an SSD in the Clearwater setup gives an advantage to process verbose logs, i.e., the Clearwater microservices. About the file size, it can be noted that, in the Kubernates case, BL logs tend to be bigger than MF logs, if compared to the Clearwater case (as discussed in Section 8.2), hence their collection introduces a higher performance penalty.

8.2 Log Size overhead

We measure the average **log size** of BL and MF logs produced during experiments for all LDs and for both the

6. We limit our experiments to 12 concurrent clients in order to avoid approaching the system's unavailability.

TABLE 3: Log size overhead in Clearwater.

	BL	MF	$O_{MF_{BL}}$
	average log size (KB)		%
2 (LOW)	15.82	6.52	-58.79
6 (MED.)	46.90	19.32	-58,81
12 (HIGH)	93.52	38.72	-58,60

TABLE 4: Log size overhead in Kubernetes.

	BL	MF	$O_{MF_{BL}}$
	average log size (KB)		%
1 (LOW)	5,125.18	1,173.39	-77.11
3 (MED.)	15,558.59	3,549.93	-77.18
30 (HIGH)	18,1720.55	39,504.47	-78.26

case studies. Results are reported in TABLE 3 for Clearwater and TABLE 4 for Kubernetes, along with the overhead of MetroFunnel with respect to the builtin logs in terms of log size. Overhead is computed as follows. Given a level of LD, let us denote by (i) LS_{BL} the size in KB of the either Clearwater's builtin logs or the logs from *webserver* replicas running on Kubernetes and (ii) LS_{MF} the size of the log trace generated by MetroFunnel. The percentage overhead induced by MF with respect to BL is given by: $O_{MF_{BL}} = \frac{LS_{MF} - LS_{BF}}{LS_{BF}} \cdot 100$.

It can be seen that in both case studies MF trace is significantly smaller than overall BL logs; in particular, it is less than half of BL in the case of Clearwater, and around one quarter of BL in the case of Kubernetes. As expected, the difference in size depends on the target system and on the verbosity of builtin logs that, in our case studies, is higher for Clearwater logs. This reflects in the overhead reported in the rightmost column of both TABLE 3 and TABLE 4, which can be seen as **a potential reduction**, in terms of log files to be collected and transmitted over the network, if one decides to collect and centralize MF logs only. It can be noted that MetroFunnel achieves a reduction of the log files accounting for about 59% and 78% for Clearwater and Kunbernetes, respectively, in the worst case setting.

9 Threats to validity

As for any study proposing a new approach, there may be concerns regarding the validity and generalizability of the proposal and results. We discuss them, based on the aspects of validity listed in [51].

Construct validity. Our study builds around the intuition that traces gathered at negligible overhead and with no knowledge of the application design, can support useful attributions on microservices. This is pursued by instantiating the proposal in the context of synchronous protocols with REST, which is the most used style for APIs development. Our implementation leverages a consolidated Java library for tracing. We rely on log analysis methods and our past expertise in the area. The analysis is based on two case studies, namely Clearwater IMS and Kubernetes orchestrator, which are two representative examples of microservices systems. It can be observed that asynchronous protocols, such as the Advanced Message Queuing Protocol (AMQP), are used too for microservices communication. MetroFunnel currently does not target these protocols, however, monitoring solutions are already available for most adopted implementations. Examples are MONAD [33], discussed in section 2, and the monitoring layer of RabbitMQ [52]. For instance, the latter provides a wide set of useful metrics for black box analysis, such as, number of unacknowledged messages, number of messages ready for delivery, publishing and delivery rate. As future work, we will investigate the integration of MetroFunnel with monitoring solutions for asynchronous protocols, with the same aim: to accompany microservices logs with black box tracing.

Internal and conclusion validity. Findings have been inferred by means of a rigorous approach encompassing design of experiments. We replicated the experiments under different configurations of the key factors. We opted for a conservative system deployment with the aim of avoiding any favorable setting to our proposal. We evaluated different metrics; moreover, analysis has been supplemented by manual investigations of both the logs in hand and the trace by MetroFunnel. Overall, this mitigates internal validity threats and provides a reasonable level of confidence on the conclusions.

External validity. Our proposal should be easily applicable to other similar LAN systems. Passive tracing requires no microservices modifications; practitioners are not expected to spend any efforts in using our proposal or supporting a certain methodology. MetroFunnel does not incorporate the knowledge of the application design. We do not require a specific log management framework for collecting the execution traces. The details provided can reasonable support the replication of our study by other researchers and practitioners. Most notably, we made MetroFunnel available to the community.

10 Conclusion

This paper presented an approach to accompany microservices logs with *black box* tracing in order to support informed decisions by practitioners. Our solution is based on passive tracing and aims to cope with the flexibility requirements of microservices systems. We achieve an average reduction of around 59% in log size, at 3.3% higher performance overhead when compared to the collection of buitlin logs in the worst case, i.e., our Clearwater setup. Better results have been obtained for Kubernetes, where MetroFunnel exhibited a performance overhead of around 10% lower than the one obtained collecting the webserver replicas logs, with an average reduction of around 78% in log size.

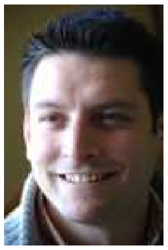
Our work highlighted new challenges underlying log analysis in this domain; as such, we believe that the findings of our study should be extremely useful to practitioners and to drive future research directions. We will devote future work to engineering a *full-fledged* framework on the top of tracing, which will encompass –for example– interactive dashboards, filters, and automated tools for archiving and browsing the traces. While these aspects do not pose specific research challenges, they are meant to boost the usability of our approach for production environments. We also plan to conduct an experimental campaign aiming at understanding the types of errors that can be caught with our proposal.

Acknowledgments

This research was carried out in the frame of Programme STAR, financially supported by UniNA and Compagnia di San Paolo under project "Towards Cognitive Security Information and Event Management" (COSIEM).

REFERENCES

[1] M. Cinque, D. Cotroneo, R. Della Corte, and A. Pecchia. Characterizing direct monitoring techniques in software systems. *IEEE Transactions on Reliability*, 65(4):1665–1681, Dec 2016.
[2] E. Wolff. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016.
[3] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. *Microservices: Yesterday, Today, and Tomorrow*. Springer International Publishing, 2017.
[4] E. Fadda, P. Plebani, and M. Vitali. Optimizing monitorability of multi-cloud applications. In *Advanced Information Systems Engineering*. Springer International Publishing, 2016.
[5] A. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, February 2012.
[6] Splunk. http://www.splunk.com.
[7] Logstash. https://www.elastic.co/products/logstash.
[8] Instana. https://www.instana.com/.
[9] Zipkin. https://github.com/openzipkin/zipkin.
[10] Sysdig. https://sysdig.com/opensource/.
[11] Dynatrace. https://www.dynatrace.com/.
[12] Clearwater. http://www.projectclearwater.org/.
[13] Nagios. http://www.nagios.org/.
[14] M. L Massie, B. N Chun, and D. E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.
[15] Amazon Cloudwatch. https://aws.amazon.com/it/cloudwatch/.
[16] cAdvisor. https://github.com/google/cadvisor.
[17] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia. Elascale: Autoscaling and monitoring as a service. *CoRR*, 1711.03204, 2017.
[18] Docker. https://www.docker.com/.
[19] Elastic. https://www.elastic.co/.
[20] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: Actionable insights from monitored metrics in microservices. *CoRR*, 1709.06686, 2017.
[21] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu. Conmon: An automated container based network performance monitoring system. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 54–62, May 2017.
[22] AppDynamics. https://www.appdynamics.com/.
[23] CA APM. https://www.ca.com/us/products/application-performance-monitoring.html?intcmp=headernav.
[24] New Relic. https://newrelic.com/.
[25] H. S. Benjamin et al. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.
[26] B. Mayer and R. Weinreich. A dashboard for microservice monitoring and management. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 66–69, April 2017.
[27] M. Camilli, C. Bellettini, and L. Capra. Design-time to run-time verification of microservices based applications. In *Software Engineering and Formal Methods*, pages 168–173, Cham, 2018. Springer International Publishing.
[28] Netflix Conductor. https://netflix.github.io/conductor/.
[29] M. Camilli, A. Gargantini, P. Scandurra, and C. Bellettini. Event-based runtime verification of temporal properties using time basic petri nets. In *NASA Formal Methods*, pages 115–130, Cham, 2017. Springer International Publishing.
[30] Netflix Hystrix. https://github.com/netflix/hystrix.
[31] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89, USA, 2003. ACM.
[32] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: Black-box performance debugging for wide-area systems. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 347–356, USA, 2006. ACM.
[33] P. Nguyen and K. Nahrstedt. Monad: Self-adaptive micro-service infrastructure for heterogeneous scientific workflows. In *2017 IEEE Int. Conference on Autonomic Computing (ICAC)*, pages 187–196, July 2017.
[34] F. Pina, J. Correia, R. Filipe, F. Araujo, and J. Cardroom. Non-intrusive monitoring of microservice-based systems. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8, Nov 2018.
[35] Zuul. https://github.com/netflix/zuul.
[36] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66, June 2016.
[37] Dhruv Sharma, Rishabh Poddar, Kshiteej Mahajan, Mohan Dhawan, and Vijay Mann. Hansel: Diagnosing faults in openstack. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 23:1–23:13, New York, NY, USA, 2015. ACM.
[38] Ayush Goel, Sukrit Kalra, and Mohan Dhawan. Gretel: Lightweight fault localization for openstack. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 413–426, New York, NY, USA, 2016. ACM.
[39] Programmable Web. www.programmableweb.com.
[40] Is REST Successful in the Enterprise? www.infoq.com/news/2011/06/is-rest-successful.
[41] Anja Feldmann. Blt: Bi-layer tracing of http and tcp/ip1an earlier version of this paper was presented as a position paper at the w3c web characterisation workshop, november 1998, cambridge, ma.1. *Computer Networks*, 33(1):321 – 335, 2000.
[42] RFC7231. https://tools.ietf.org/html/rfc7231.
[43] jNetPcap. http://jnetpcap.com/.
[44] Clearwater live test. https://github.com/metaswitch/clearwater-live-test/.
[45] Filebeat. www.elastic.co/products/beats/filebeat.
[46] Kubernetes. https://kubernetes.io.
[47] Microk8s. https://microk8s.io.
[48] Kubernetes test suite. https://github.com/mrahbar/k8s-testsuite.
[49] Vegeta. https://github.com/tsenart/vegeta.
[50] R. Jain. *The Art of Computer Systems Performance Analysis*. J. Wiley & Sons New York, 1991.
[51] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic, 2000.
[52] RabbitMQ. https://www.rabbitmq.com/monitoring.html.



Marcello Cinque (S'04-M'06) graduated with honors from University of Naples, Italy, in 2003, where he received the PhD degree in computer engineering in 2006. Currently, he is Associate Professor at the Department of Electrical Engineering and Information Technology of the University of Naples Federico II. Cinque is chair and/or TPC member of several technical conferences and workshops on dependable systems, including IEEE DSN, EDCC, and DEPEND. His interests include dependability assessment of critical systems and log-based failure analysis.

Raffaele Della Corte (S'15-M'17) received the B.S., M.S. and Ph.D. degrees in computer engineering from the Federico II University of Naples, Italy in 2009, 2012 and 2016, respectively. He is a Postdoctoral Researcher at the Department of Electrical Engineering and Information Technologies, Federico II University of Naples. His research interests include data-driven failure analysis, on-line monitoring of software systems, and security. Dr. Della Corte serves as reviewer in several dependability conferences and he is involved in industrial projects on the analysis and monitoring of critical systems.

Antonio Pecchia (S'09-M'13) received the B.S. (2005), M.S. (2008) and Ph.D. (2011) in Computer Engineering from the Federico II University of Naples, where he is now an Assistant Professor. He is a co-founder of the Critiware spin-off company (www.critiware.com). He was a post-doc at the National Interuniversity Consortium for Informatics (CINI) in European projects. He serves as TPC member and reviewer in conferences and workshops on software engineering and dependability. His research interests include data analytics, log analysis, empirical software engineering, dependable and secure distributed systems. He is a member of the IEEE.