Nonintrusive Monitoring of Microservice-based Systems

Fábio Pina, Jaime Correia, Ricardo Filipe, Filipe Araujo and Jorge Cardoso
CISUC, Dept. of Informatics Engineering
University of Coimbra
Coimbra, Portugal
fpina@student.dei.uc.pt, jaimec@dei.uc.pt, rafilipe@dei.uc.pt, filipius@uc.pt, jcardoso@dei.uc.pt

***Abstract*—Breaking large software systems into smaller functionally interconnected components is a trend on the rise. This architectural style, known as "microservices", simplifies development, deployment and management at the expense of complexity and observability. In fact, in large scale systems, it is particularly difficult to determine the set of microservices responsible for delaying a client's request, when one module impacts several other microservices in a cascading effect. Components cannot be analyzed in isolation, and without instrumenting their source code extensively, it is difficult to find the bottlenecks and trace their root causes. To mitigate this problem, we propose a much simpler approach: log gateway activity, to register all calls to and between microservices, as well as their responses, thus enabling the extraction of topology and performance metrics, without changing source code. For validation, we implemented the proposed platform, with a microservices-based application that we observe under load. Our results show that we can extract relevant performance information with a negligible effort, even in legacy systems, where instrumenting modules may be a very expensive task.**

***Index Terms*—Black-box monitoring; Gateway; Microservices**

I. Introduction

Microservice modules have become a trend in the development of distributed system. This new paradigm evolved due to a number of factors. First, standard monolithic systems were difficult to maintain, deploy, develop and scale. Hence, there was the need to decompose these vertical systems in modules that are function-oriented and that could be handled separately, in terms of development and management. Secondly, microservice architectures are better suited for deployment and operation in Docker [1] or other containers. Finally, methodologies in product development, such as Agile or DevOps, with smaller teams that work independently, are more aligned with microservice architectures. Therefore, microservices have tremendous benefits in term of development, operation, availability and scalability, and have thus become a standard for large-scale systems.

Despite the aforementioned benefits, there are some challenges to tackle. One of these challenges is monitoring. In old monolithic systems, monitoring was restricted to the system, with a stable infrastructure and little elasticity. In microservice systems, administrators have to pinpoint the root cause of the anomaly in hundreds or thousands of machines, with services that have high elasticity and communicate with each other. This huge increase in complexity, creates a herculean task for administrators.

978-1-5386-7659-2/18/$31.00 ©2018 IEEE

There are some monitoring tools, that due to their proven capabilities in standard systems, were also adopted in microservice architectures. These tools — e.g. `Nagios` or `Zabbix` —, normally monitor several infrastructure metrics, such as CPU or memory usage, and include dashboards to provide an overview of the system, with functionalities to notify administrators when some rule or threshold is violated.

Other monitoring platforms, such as New Relic [2] or Dynatrace [3] are intrinsically coupled to the programming language and system to monitor, but offer an overall overview of the infrastructure. Other interesting approaches are `Kibana`[4] or `Grafana` [5]. `Kibana` is used primarily to analyze logs and `Grafana` is more prepared to analyze and create visualizations of system metrics such as CPU or I/O utilization.

Since there is communication between several microservices, a more powerful monitoring technique consists of instrumenting all the modules, creating traceability for a particular request. Tracing normally propagates a correlation identifier that can be used to determine the flow through several microservices. In other words, tracing allows system administrators to determine the entire workflow of applications. There are some frameworks that help to implement tracing, such as ZipKin [6], Opentracing [7] or Dapper [8]. Despite the benefits, tracing brings two major drawbacks. First, all microservices must have tracing implemented and be responsible to send the data to a central point. This platform gathers, processes and aggregates the raw data. Therefore, developers have to focus, not only on the business algorithm, but also on monitoring and operation of the microservice. Secondly, the central point may be a system bottleneck, due to the large number of records. In fact, tracing systems normally purge older samples or save only a small percentage of data.

Bearing in mind the aforementioned solutions, one could think that administrators have all the tools to monitor systems. However, in reality, operators use a plethora of platforms and frameworks, some of them adopted from monolithic systems. These tools only give insights of what is happening in the system, and is the administrators' responsibility to endure the hard

task to navigate through several dashboards and notifications to identify the problem. Hence, microservices have introduced a new paradigm to develop distributed systems, with well defined functions and boundaries, but still use monitoring techniques similar to what we could find in older architectures.

The approach we propose here is completely “black-box”, decoupling monitoring functionalities from function-oriented microservices. It is a solution that is neither invasive, nor disruptive, as it requires no adaptations on the microservice level. As a consequence, it is a good solution to already implemented production systems. To achieve this, we used and adapted a gateway from Netflix, named Zuul [9], to collect metrics from the requests made by the microservices. Based in metrics such as response time, origin and destination of the requests, we aggregated the raw data in a concise output with relevant information, such as average response time, topology and overall service characterization.

Our results show that we can obtain relevant and useful information to system administration, even though we use a non intrusive approach methodology. We do not need to instrument microservices or add agents to the infrastructure, resorting only to components already needed by microservices modules. Therefore, the solution presented is useful, viable — specially in very dynamic and elastic systems —, and aligned with microservice methodology.

The rest of the paper is organized as follows. Section II describes the problem we tackle and the method we used to solve it. Section III describes the experimental settings. In Section IV we present and evaluate the meaning of the results, the strengths of this approach and its limitations. Section V presents the related work. Finally, Section VI concludes the paper and describes future directions.

II. Proposed Methodology

In this paper we tackle the problem of monitoring microservice architectures. In vertical solutions, monitoring is easier, because the application does not change that much over time. Microservice systems evolved from new development methodologies, such as Agile or DevOps and new deployment techniques, such as containers. System monitoring did not follow this evolution and is still based on the applications and techniques for monolithic systems. In Section V, we discuss how major worldwide technological companies are struggling with this fact, being forced to create customized platforms for their needs. Indeed, monitoring is a complex and difficult problem in highly dynamic systems.

We analyzed the monitoring problem from a different perspective. A typical approach for monitoring would consists of instrumenting or adding agents in as much layers as possible, from hosts to middleware, up to the application layer. Refer to Figure 1, which shows a sequence of three microservices, where some function in A invokes a function in B, which invokes a function in C. To bring the information of the interdependent invocations to the monitoring system, messages must carry some identifier that allows their

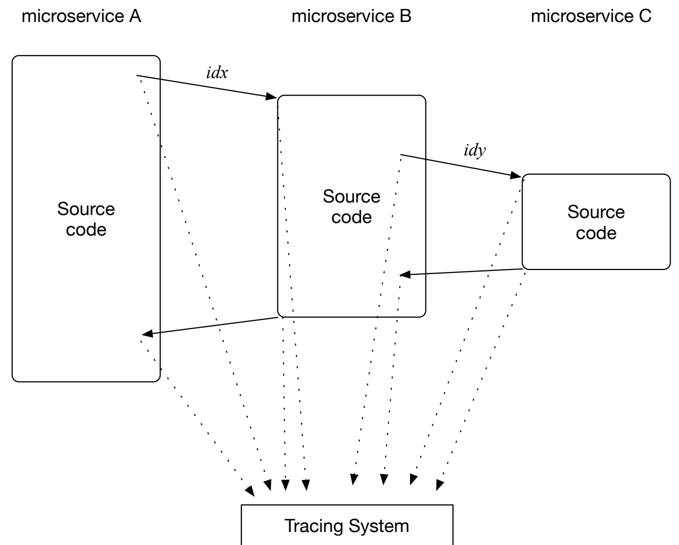


Fig. 1: Tracing of microservices application (optimizations to reduce tracing messages omitted).

correlation, for example, an HTTP header with the same identifier ($idx = idy$). Unfortunately, this involves changing the source code of the application. While this technique creates several monitoring points, these are also additional points of failure and maintenance that couple monitoring with business logic. This goes against the microservice methodology, which follows the premise of function-oriented fine-grained modules. To eliminate the need for instrumentation, we follow a non-traditional approach. Knowing that microservices resort to a gateway to make service discovery and redirect requests, we added the capability to collect some monitoring metrics to this gateway. The idea is to make the gateway gather information, such as response time, IP and port of origin and destination, and the identification of the function that was invoked. This approach brings advantages, such as decoupling the monitoring system from the application without hindering system scalability, because the gateway and associated services are horizontally scalable.

In the next subsections we describe our methodology in more detail. First, we present the architecture, and how we incorporate our solution in Netflix modules. Secondly, we go through the metrics we can collect in the gateway and discuss the information dashboards we can build with standard tools. Finally, we present how we implemented and distributed the tool.

A. Architecture

In a microservice infrastructure, it is a common approach to solve the service discovery problem with a gateway. Hence, it is very appealing to take advantage of this module to observe the system. We resorted to three Netflix components: Zuul, Ribbon and Eureka, responsible for gateway, load balancing and registry of scalable service, respectively. These services allow us to gather metrics outside of the critical path, being

TABLE I. COLLECTED METRICS

Metric	Type
Start Time	Long
End Time	Long
Duration	Long
Origin IP	String
Origin Port	Integer
Destination Service	String
Destination Instance	String
Destination IP	String
Destination Function	String

an advantage to monitor microservice systems. In Figure 2, we present the high level architecture.

Our methodology has four components, that are aligned with microservice best practices, such as service discovery or containerization. First, the module "Metric Storage" gathers metrics collected by the customized Zuul application. These metrics are response time — in requests between services or directly from the client —, IP and port of origin and destination of the request, and function invoked on the destination microservice. This module, also acts as a backend to the "Frontend" module, where we display relevant information such as response times, topology and characterization of services.

The other two modules are associated with "Service Registry" and "Failure Detection". In this paper, we focused on docker containers [1], given its popularity. By analyzing containerization, data description of containers and how the container manager works, it is possible to fully automate the process of registry and failure detection. This give us a tremendous advantage, since we do not need container instrumentation. In this case, our module is notified when a change occurs in the containers, such as creation, destruction or state change, through an agent associated to the container manager. Bearing in mind that we also observe HTTP results, it is also possible to implement a module responsible for failure detection. This module combines information from the HTTP results with container status, to add capabilities to our system of autonomous maintenance and recovery. When an instance fails, it is possible to remove or restart this instance, without needing administration supervision or microservice instrumentation. It is also relevant to mention that the components involved in monitoring are horizontally scalable, and therefore do not harm performance or availability of the application. Since we remove the instrumentation necessary of systems like the one of Figure 1, the processes responsible for extraction and processing of the metrics are outside of the critical path, and consequently do not create any sort of overhead.

B. Collected Metrics

We present in Table I the metrics collected in our "Metric Storage" module. For each request, regardless of the origin (either another microservice or a client), we save metrics associated to the origin and destination of the request.

Beside standard plots with averages and quartiles, e.g., as in box-plots, this raw information allows us to create high-level information about the system. For example, it is possible to dynamically extract topological information and characterize the level of interaction among different microservices. Additionally, we can also calculate response times and load of each microservice, inferring maximum capacity and quality-of-service of each module, to ensure correct dimensioning.

For the frontend layer of our monitoring system, we used `Grafana` [5], an open platform for analytics and monitoring, highly flexible and customizable. To display a few more complex plots, we used as a complement, graphics generated using the `R` language — a common standard in the academic field, for simulation and analysis, incorporating the output on `Grafana`.

C. Implementation

To validate our method, we made a fully nonintrusive implementation for the `docker swarm` container management platform. The source-code and deployment instructions are available on GitHub [10] as open-source. Additionally, it also contains the sample microservice application used in our experimental validation, of Section III. The tool is easily deployable in a system with `Docker` and `Swarm Manager` installed. Since the monitoring tool needs an overlay network [11], the system must have this network created and configured, to ensure the correct operation of our approach.

Afterwards, the only parametrization needed is the name of the `overlay` network. The remaining parameters may be defined with the default values without loss of functionality. To use our monitoring solution, one could download the repository, define the `overlay` network in our configuration file and run the installation script that will automatically generate and deploy a `docker-compose` manifest file. The Service Registry component, described in Subsection II-A, will subscribe to the `docker` event API and be notified of container creation, destruction and state change. As such, service registration on the gateway will be done automatically, requiring no collaboration from the services themselves. This is possible because each container already carries the relevant metadata, such as name and service port.

The monitoring solution includes the user-customizable frontend module, with `Grafana`. Furthermore, we developed and included a custom plug-in, written in `R`, to generate more complex visualizations, such as chord diagrams [12]. Our raw data storage module, uses influxDB and MySQL databases. In Table II we present the overall containers associated (and deployed) with the tool. Once installed in a `Docker Swarm` container manager, all other applications deployed on it will automatically use our gateway for service discovery and monitoring, as long as they are in the same `overlay` network.

Figure 3 shows an example dashboard, extracted during the experimental stage. In this case, we show the charts described in Section IV, such as histograms and chord diagrams.

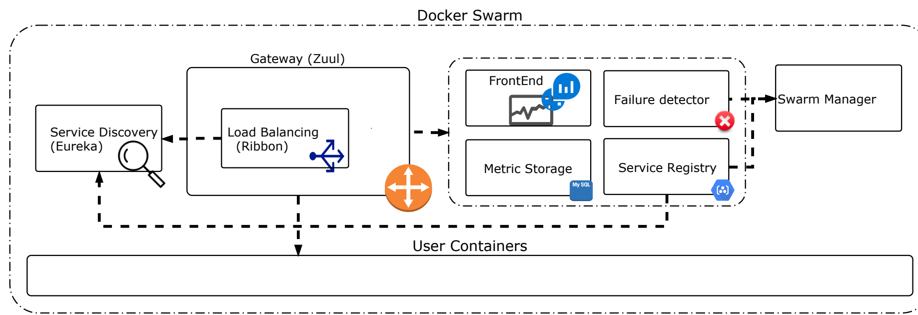


Fig. 2: System components

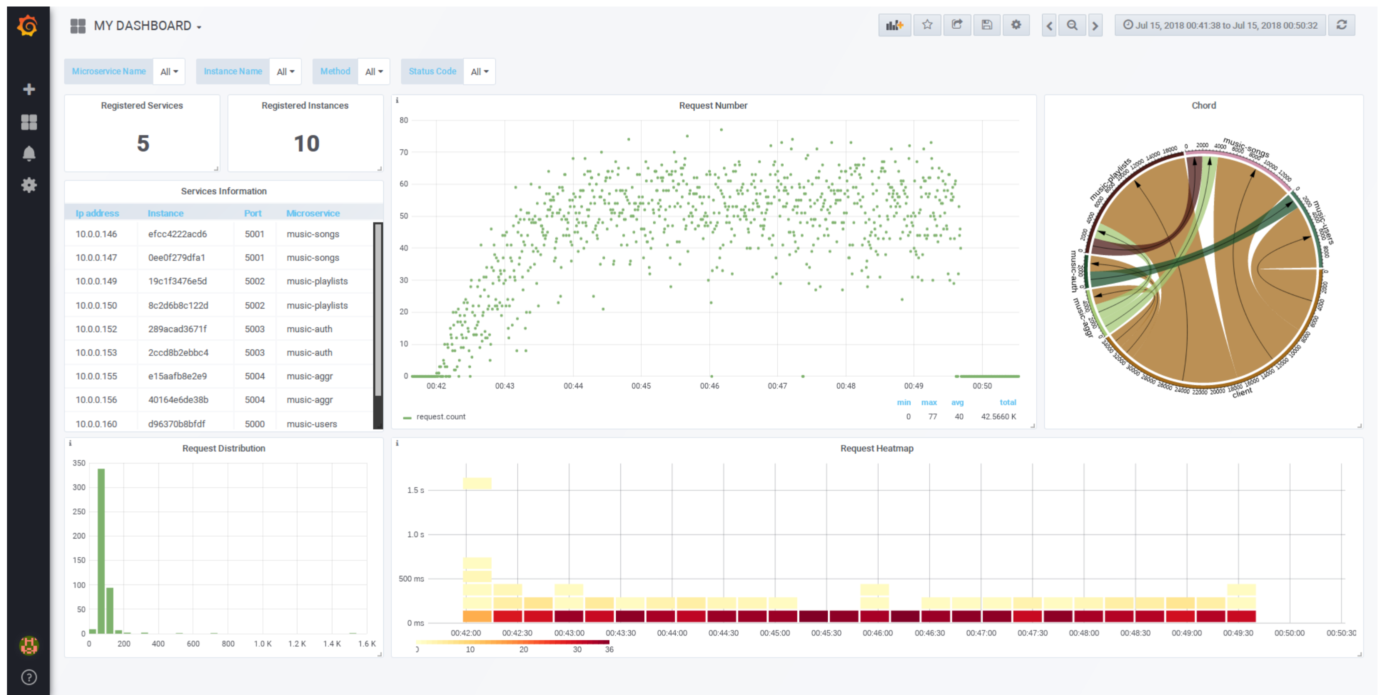


Fig. 3. Sample user-customizable frontend.

TABLE II. Tool Containers

Container	Description
Eureka	Service Discovery
Zuul	Gateway
Service Registry	Manages containers life-cycle, in association with Eureka
InfluxDB	Time-series scalable DB
MySql	DB
Grafana	Frontend
Chord Plugin	Generates chord visualizations

III. Experimental Setup

In this section we present the experimental setup used, the changes made to Netflix modules, and our microservice application. First, concerning the infrastructural modules, used by the test application, we rely on modules that are stable. To do load balancing, we used Ribbon. This module gives us several advantages, such as the available load balancing algorithms, the use of REST interfaces, but most importantly, an off-the-shelf integration with the remaining support modules from Netflix. Hence, integration with the discovery and registry module — Eureka —, is made, allowing a more agile instantiation and implementation of our methodology. Aligned with Ribbon and Eureka, we also used the Netflix gateway — Zuul —, that uses Ribbon internally. Zuul gets the service location through a query to Eureka, and then routing requests to the correct service. Since requests have to pass through Zuul, this module allow us to have a clear vision regarding traffic between microservices and gather monitoring information to a central point.

The other component of our experimental setup is the application that allows us to test the monitoring method. The application that we implemented is related to music and has five microservices with well defined functions. The application allows its clients to manage users, playlists and songs. On Table III, we identify the overall endpoints associated to each microservice, respective invocations methods and a brief

TABLE III. Microservice and functions available

microservice	functionality	request type	description
Authentication_MS	/	GET	System Healthcheck
	/login	POST	Validate user credentials and create token
Users_MS	/	GET	System Healthcheck
	/login	POST	Validate user credentials
	/users	GET	Get user info
	/users	POST	Create user
	/users/{id}	DELETE	Remove user
	/users/{id}	PUT	Update user
Playlists_MS	/	GET	System Healthcheck
	/playlists	GET	Get playlists associated to a user
	/playlists	POST	Create playlist
	/playlists/{id}	GET	Get playlist
	/playlists/{id}	PUT	Update playlist
	/playlists/songs/{id}	DELETE	Remove a specific music from a playlist
	/playlists/songs/{id}	GET	Get music info associated to a playlist
	/playlists/songs/{id}	POST	Add music to playlist
Songs_MS	/	GET	System Healthcheck
	/songs	GET	Get music info
	/songs	POST	Create music info
	/songs/convert/{id}	GET	Convert music from `mp3` extension to `wav`
	/songs/criteria	GET	Get music list based on some criteria
	/songs/{id}	DELETE	Remove music
	/songs/{id}	PUT	Update music
Aggregator_MS	/	GET	System Healthcheck
	/playlists/songs/{id}	GET	Get all music info associated to a playlist

description.

Since we wanted to collect raw information about the requests, but without instrumenting microservices, we changed the Zuul source code to register information concerning origin and destination of each request. We save the following information: microservice that made the request, start time, end time, IP and Port of the request origin, microservice instance that processed the request, and function that was invoked. With this information, we were able to extract relevant information about the system, such as topology or average response times, decomposed by microservice and function. As mentioned, we did not need any kind of instrumentation on the source code of the application (i.e., we only changed the infrastructure). The raw data is then pre-processed and redirected to a MySQL database that makes part of our "system metric" module.

The software was installed in a virtual environment running Ubuntu 16.04. The virtual machine had 8 vCores, with 22 GiB of RAM. All components were installed with standard parametrization, except the Zuul parameter `sensitive-headers`. This configuration allows us to propagate the authentication token through all microservices without any kind of manipulation from the gateway.

To simulate load on the system, we used Apache JMeter [13]. We configured this load tool with 10 threads, and a launching period of 120 seconds. Each thread ran during 10 minutes with the following loop: 1) Create User; 2) Authenticate; 3) Get user; 4) Update user; 5) Add song; 6) Get song; 7) Update song; 8) Convert song; 9) Add playlist; 10) Get playlist; 11) Update playlist; 12) Add music to playlist; 13) Get music from playlist; 14) Get all musics from playlist; 15) Delete music from playlist; 16) Delete playlist; 17) Delete song; 18) Delete user.

TABLE IV. Software used

Component	Observations	Version
Zuul	Gateway	1.4.4
Eureka	Service discovery	1.4.4
Ribbon	Load balancer	1.4.4
MariaDB	DB used by microservices	10.3.7
MySQL	DB used by the frontend	8.0
JMeter	Load testing tool	4.0

In Table IV we present the open-source components used in the experiment and respective versions.

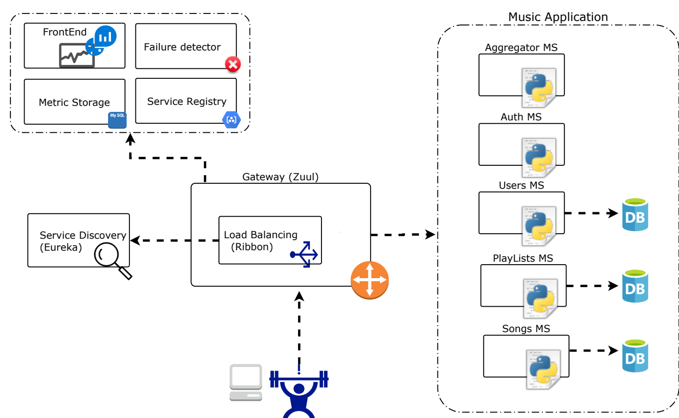


Fig. 4. System architecture

Our ultimate goal with this experiment is very simple: understand the limits, benefits and disadvantages of our "black-box" nonintrusive monitoring tool. Figure 4 summarizes the entire system, with application, infrastructure, including the monitoring tool and a load generator.

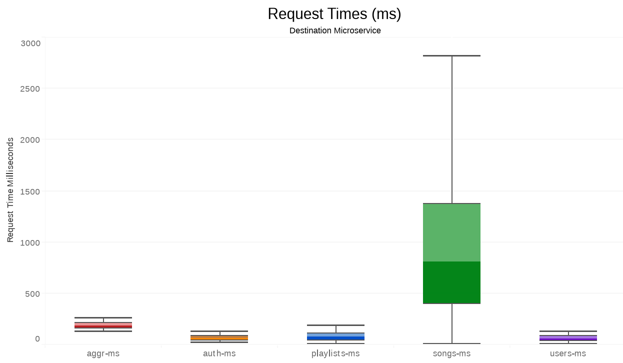


Fig. 5. Boxplot of response time by microservice

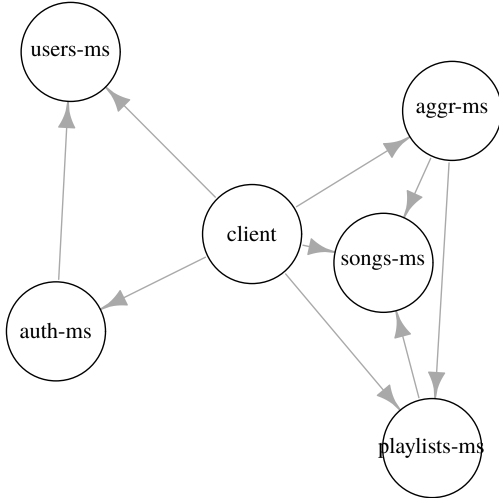


Fig. 6. Application graph

IV. Results

In this section, we present the results of gathering monitoring data from the API gateway. This technique allows us to extract raw data from microservice interactions and, therefore, create a set of metrics and charts with relevant information for administrators, without the need of instrumentation or agents at hosts level. In this paper, we present 5 visualizations that combined, give us a clear vision of the system.

Concerning the frontend application, we divided visualization into 3 distinct charts. First, we need to understand what microservices have a higher variance in the response time. To get this data — see Figure 5 —, we opted for a boxplot chart. This kind of graphic allows us to have compressed information in only one visualization. Figure 5 was created based on data extracted from our MySQL database. It is relevant to mention that although we are presenting response time distributions of microservices, it is possible for the user to drill-down, and visualize the same distribution by destination function inside each microservice.

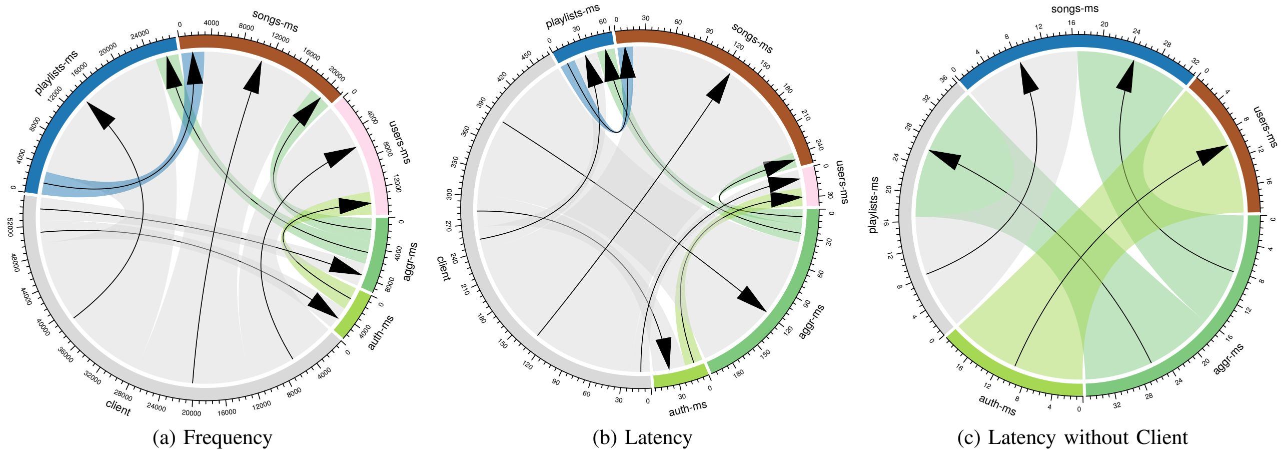
Regarding dependencies between microservices, we resort to a graph. This representation allows us to present topology and dependencies between modules. In Figure 6, we can easily see the relations between the different microservices, and the direct accesses from clients.

Having response times distribution in boxplot charts for each microservice (and function) and dependencies between microservices in graph visualization, there is still a crucial aspect missing, to understand the health of the system: microservice and function importance in the system. To achieve this, we resort to chord diagrams, based on the work of Gu *et al.* [12]. This kind of graphic allows us to see more complex relations between entities. Graph nodes are arranged along a circle, and the importance of their interactions is proportional to the width of the connecting arcs. We use arrows to provide information of which side receives the call, and colors to simplify interpretation. For instance, in Figure 7a, we can see the number of requests, and in Figure 7b latency. In a very large system, a chord graph comprising everything would probably be difficult to read. Hence, to improve diagram interpretation, administrators can select what microservices to display, as seen in Figure 3. For instance, in Figure 7c we see latency, without requests made by the clients, since these requests can have a huge impact on the graph and make other interactions less visible.

Taking into consideration Figure 7a, an administrator would verify that microservice `playlist-ms` would be the origin or destination of around $28,000$ requests. From these, around $3,000$ were requests from `playlist-ms` to `songs-ms`, $21,000$ requests made directly by clients and around $3,000$ from the `aggr-ms` microservice. This way, we have a vision of the `playlist-ms` microservice relevance in the overall system. Additionally, the same analysis could be made for latency. The box-plots, combined with the dependency graph and the chord diagram, give us a good idea of the system capacity, module importance and response time distribution by microservice or function.

An interaction of a system administrator with the monitoring system could go this way: the administrator would first look to Figure 5. This box-plot, provides a clear understanding of what services have higher response times. It is easily observable that the service `songs-ms` has the highest response times of all modules. The second module with higher response times is `aggr-ms`. Nevertheless, an administrator would try to understand the importance that the `songs-ms` microservice has in the system. Although it has a higher response time compared with other microservices, an administrator should look to the remaining Figures. With Figure 6, he can see that `songs-ms` receives invocations directly from the client, `aggr-ms` and `playlists-ms`, so, there is a large system dependency on the `songs-ms` microservice. Furthermore, we notice that the latency of `songs-ms` depends on who is invoking it, presenting a much higher latency for client-initiated invocations. This would show that either they are invoking different functions or there is some anomaly. An administrator - using our application - can then drill down and see granular invocation data for further analysis.

The last information that an administrator needs is the number of requests and latency in the calls between microservices.



(a) Frequency (b) Latency (c) Latency without Client

Fig. 7. Chord diagrams

Even though songs-ms has a high response time, and is a key module in terms of dependencies, we need to understand if the number of requests that go through songs-ms is relevant in the overall number of requests processed by the system. To have this information, we can look to Figure 7a and 7c. We can see that songs-ms is an important destination of requests, specially from client and aggr-ms. In fact, looking to Figure 7c, we can see that aggr-ms dependencies (songs-ms and playlists-ms) have low latencies, so they are not a bottleneck.

Given that Figure 7a shows the aggr-ms service makes roughly twice as many invocations as it gets, it leads to the conclusion that latency is a result of multiple requests, possibly serially or with low parallelism. Therefore, an administrator with these visualizations would have two possible solutions: drill-down the boxplot of songs-ms by function, to check if there is any offending function, and/or try to improve the way aggr-ms invokes dependencies. It is important to remember that all this information is achievable with no instrumentation or agents in the infrastructure.

When we compare our approach to current monitoring tools for microservices, we can see some benefits, as well as disadvantages. One of the disadvantages, is related to tracing. We do not have the granularity that tracing offers, to understand the workflow of specific requests. Hence, we may miss some information concerning causality between microservices. Nevertheless, if we have a widespread distribution of requests, we can still estimate the workflow. On the other hand, our module is far less intrusive, as it does not have the overhead to develop instrumentation or deploy agents in the system. Additionally, our solution could be implemented in legacy systems in a very agile way, something that is probably beyond reach of tracing-based solutions.

V. Related Work

Since our work is tangential to distinct research fields, and is a very active topic, we divided related work between industry and academic solutions. Additionally, we present some work that despite not being directly related to ours, is complementary.

First, concerning industrial approaches, we have software from Netflix. Netflix has several modules for monitoring and instrumentation. Vector [14], is a framework that allows a creation of dashboards with metrics, such as CPU or network utilization. The module forces the existence of an agent — named Performance Co-Pilot (PCP) —, on each host or application to monitor. Another system is Atlas [15]. This platform is focused on big data and time series. The goal is to apply prediction methods, to understand the evolution of metrics and real time analysis. Although powerful, this platform requires instrumentation of microservices. Another very similar approach is Prometheus [16], an open-source monitoring solution that also requires instrumentation.

Application Performance Monitoring (APM) tools, based in instrumentation or agents, allow the creation of dashboards and the definition of notifications to administrators, when some threshold is violated. For example, Dynatrace [3] and others [2], [4], [5], [17] have some features related to microservice infrastructures. However, all these tools are mostly focused on displaying information, and are much less concerned with intrusiveness than we are. Spotify uses a similar approach to Netflix. They had the need for a customized monitoring infrastructure that creates dashboards and time series. Once again, each machine runs an agent to send information to a central point [18]. Additionally, there are some open-source projects, like IOVisor [19], that detects performance problems in thousands of virtual machines.

Looking into academic contributions, in [20], the authors give some guidelines on how to build and monitor a microservice platform. The availability of instrumentation or agents to collect information from hosts and applications is assumed. In [21], the authors present a monitoring dashboard, but once again based on agents and service instrumentation.

In [22], the authors use a distinct approach, where each microservice is responsible for its own elasticity and scalability.

The modules save the information about CPU utilization and response times. Although this has some benefits, the authors do not focus on the "domino effect", or chain reaction that may occur as components interact.

In [23], the authors propose a methodology to create "monitoring as a Service", based on containers, where agents are associated with the microservice container. In this architecture, there is a one-on-one relationship between agent and container that may cause some overhead and scalability issues. Additionally, monitoring is associated with the container, in disfavor of the workflow that exists between modules. In [24], the access point of each container is changed to monitor the network. Hence, it has a kind of "man-in-the-middle" approach having no consideration about the application.

Our methodology is different from all the previous ones in at least two aspects. Some tools referred before aim to create dashboards and notifications, based on the premise that agents or tracing is available in the system. This kind of tools does not aim to decrease system intrusiveness, focusing only in visualizations and data presentation. Other approaches instrument the system, creating frameworks coupled to containers, microservices or the network, but they do not give information about application workflows.

Tracing, one of the most used approaches of this kind, gives the capability to understand the flow of a specific request. However, there are some disadvantages: developers have to instrument each microservice and focus not only on the business algorithms, but also on monitoring. Unfortunately, this merges source code with quite distinct goals.

Unlike previous work, our method is driven by simplicity. It gives the ability to monitor a system, without agents, instrumentation or development overhead. Nevertheless, results achieved in terms of data visualization are quite impressive, enabling system administrators to grasp crucial aspects of the system with minimal effort.

VI. Conclusions and Future Work

Monitoring and operating distributed systems is a difficult task for administrators. With microservice technologies this task has become more complex than ever, due to elasticity and system dynamics. The majority of monitoring solutions were designed for older architectures, therefore lacking any considerations regarding the new paradigm.

In this paper we proposed a new approach for monitoring, without any instrumentation or probes in the system. We aimed to analyze the limits of a "black-box" approach, using only some of the infrastructural modules already deployed in a microservice architecture. We recurred to Netflix modules, customizing the gateway, to gather raw data from microservice invocations. Results show that our solution involves minimal configuration efforts to be integrated in the system and to produce relevant information to administrators.

As future work, there are several directions that we want to further investigate. First, we want to improve our open-source tool in terms of automated analysis, such as critical path enumeration and anomaly detection, to give administrators more information about high level behavior. Finally, we think it would be very helpful to generate models capable of predicting system and microservice capacity to help administrators with dimensioning and SLA assurance.

Acknowledgment

This work was partially carried out under the project PTDC/EEI-ESS/1189/2014 — Data Science for Non-Programmers, supported by COMPETE 2020, Portugal 2020-POCI, UE-FEDER and FCT.

We would also like to express our gratitude to the INCD - *Infraestrutura Nacional de Computação Distribuída*, for providing access to their computational resources.

References

[1] Docker. https://www.docker.com/what-docker. Retrieved June, 2018.
[2] New Relic. https://newrelic.com. Retrieved May, 2018.
[3] Dynatrace. https://www.dynatrace.com/platform/. Retrieved May, 2018.
[4] Kibana. https://www.elastic.co/products/kibana/. Retrieved May, 2018.
[5] Grafana. https://grafana.com/. Retrieved May, 2018.
[6] Zipkin. http://zipkin.io/. Retrieved June, 2018.
[7] Opentracing. http://opentracing.io/. Retrieved May, 2018.
[8] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc, 2010.
[9] Zuul. https://github.com/Netflix/zuul. Retrieved May, 2018.
[10] Github – monitoring_ms. https://github.com/fabiopina/monitoring_ms. Retrieved June, 2018.
[11] Docker overlay network. https://docs.docker.com/network/overlay/. Retrieved June, 2018.
[12] Zuguang Gu, Lei Gu, Roland Eils, Matthias Schlesner, and Benedikt Brors. circlize implements and enhances circular visualization in r. *Bioinformatics*, 30(19):2811–2812, 2014.
[13] Papers — Apache JMeter^{TM}. http://jmeter.apache.org/. Retrieved: May, 2018.
[14] Vector. https://github.com/Netflix/vector. Retrieved May, 2018.
[15] Atlas. https://github.com/Netflix/atlas. Retrieved May, 2018.
[16] Prometheus. https://prometheus.io/. Retrieved June, 2018.
[17] Appdynamics. https://www.appdynamics.com. Retrieved May, 2018.
[18] Spotify. https://labs.spotify.com/2015/11/17/monitoring-at-spotify-introducing-heroic/. Retrieved June, 2018.
[19] Iovisor. https://www.iovisor.org/. Retrieved May, 2018.
[20] S. Haselböck and R. Weinreich. Decision guidance models for microservice monitoring. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 54–61, April 2017.
[21] 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.
[22] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, AIMC '15, pages 19–24, New York, NY, USA, 2015. ACM.
[23] Augusto Ciuffoletti. Automated deployment of a microservice-based monitoring infrastructure. *Procedia Computer Science*, 68:163 – 172, 2015. 1st International Conference on Cloud Forward: From Distributed to Complete Computing.
[24] 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.