Transparent Tracing of Microservice-based Applications*

Matheus Santana, Adalberto Sampaio Jr., Marcos Andrade, Nelson S. Rosa
Federal University of Pernambuco, Centre of Informatics
Recife, PE, Brazil
{embs,arsj2,mvaa,nsr}@cin.ufpe.br

ABSTRACT

Tracing has been applied to study and understand the behavior and performance of distributed systems. Despite the attention this topic has received, two important aspects are still challenges and especially harmful in the context of microservice-based applications: source code instrumentation and performance overhead. Existing attempts resort on working around overhead (e.g., sampling techniques) and do not address microservices architecture's high technological heterogeneity. Our main contribution is a novel approach for tracing microservices which joins proxies' usage (for handling tracing concerns) and operating system syscalls monitoring (for diagnosing causality between multiple requests). It makes advances on the field by completely separating instrumentation and application code while minimizing performance overhead. We carry out a performance evaluation to show the impact of our solution on the execution of microservice-based applications. Our proposal fosters developers' productivity by allowing them to focus on business logic instead of instrumentation and copes with the intrinsic heterogeneity of microservices by relying on deployment modifications and operating systems mechanisms solely.

CCS CONCEPTS

• **Applied computing** → **Service-oriented architectures**; • **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

Microservices, Monitoring, Tracing, Code Instrumentation, Debugging

ACM Reference Format:
Matheus Santana, Adalberto Sampaio Jr., Marcos Andrade, Nelson S. Rosa. 2019. Transparent Tracing of Microservice-based Applications. In *Proceedings of ACM SAC Conference (SAC'19)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3297403

*This work was supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), grant number 1699939.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC'19, April 8-12, 2019, Limassol, Cyprus
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5933-7/19/04...$15.00
https://doi.org/10.1145/3297280.3297403

1 INTRODUCTION

The microservices architectural style suggests the concept of autonomous, decoupled, and technologically heterogeneous software components which interact with each other through lightweight communication protocols in a choreographic fashion. This style has been widely adopted to face everyday challenges related to software design, development, maintenance and deployment [18]. It leverages collaboration between distributed teams which may independently develop and deploy software systems by fostering the definition of boundaries between components; it promotes system resiliency as services can be easily replicated, and failure is not feared anymore, but embraced; and it also makes easier to scale applications in and out by leveraging automation to a maximum degree.

These facilities come at their price: an ecosystem of highly heterogeneous, small and replicated services imposes new challenges, such as performance and services allocation issues [11]. One of them is the difficulty to debug tricky production problems. For instance, a subtle bug in one specific component of a microservice-based architecture might lead to poor user experience and may not be detectable through HTTP error codes nor any other means which rely solely on data provided by applications and protocols they leverage.

Another challenge is to arrange components in a manner that satisfies some given system's properties, such as availability and latency. For instance, components could be co-located to diminish network latency or spread to reduce competition for resources. Whatever the goals, awareness of the actual current arrangement, which may not necessarily reflect the projected one, is mandatory.

Both components' arrangement [22] and debugging [21, 23] problems have been approached with distributed tracing tools [7, 9, 14, 24, 25, 27]. However, they usually require application developers to change their source code or merge software dependencies onto their software stack to achieve monitoring goals. Moreover, tracing tends to incur in overhead on the system-under-monitoring. Overhead issues have been commonly worked around through sampling techniques, while library- and middleware-level instrumentation are broadly applied for mitigating the need for code changes. Despite their popularity, these strategies are not ideal solutions as the former may mask non-trivial bugs, and the latter still requires software changes.

The generally high costs of instrumentation-related efforts have motivated the proposal of strategies for cheaper instrumentation in multiple research fields [12, 26]. In this paper, we propose a strategy for enabling microservices' tracing which keeps developers away from the burden of code instrumentation and has a reduced impact on the application's performance. Our primary goal is to propose a strategy for tracing services' behavior without changing their source code (nor imposing dependencies onto the software stack) neither incurring in high overhead. We do this through separation of application- and instrumentation- related code which is accomplished by using proxies for handling tracing tasks along with the monitoring of syscalls for allowing proxies to keep track of requests' causality relationships.

This paper is organized as follows: Section 2 introduces basic concepts needed to understand the proposed solution; Section 3 presents the solution in details; Section 4 presents an evaluation of the proposed strategy, and related works are discussed in Section 5. Finally, Section 6 gives insights on what has been achieved to this point and the next steps towards a readily applicable and efficient strategy for tracing microservices.

2 BASIC CONCEPTS

This section introduces some basic concepts related to microservices architectural style and tracing of distributed systems.

2.1 Microservices Architectural Style

The microservices architectural style, or microservices architecture, is the next step succeeding Service-Oriented Architecture (SOA) [6]. It dictates the conception of small, independent, and specialized software components which interact with each other for fulfilling features provided by the system from which they are constitutive elements. What makes microservices particularly useful, and also challenging, is that they take fine-grained service-oriented architectures to the next level: each component must solve one single problem from the business domain and be independently developed, deployed and executed.

By promoting extensive decoupling between architectural elements, microservices foster flexible ways of collaborative software development, computing resources allocation, and design decision making. Such flexibility paves the ground for a highly complex ecosystem of communicating pieces of software [6]. This heterogeneity poses a challenge to understand the systems' behavior as they grow larger in complexity and hampers the debugging when problems arise. It also restricts the enforcement of traditional methods for understanding a system's behavior and debugging. For instance, logging has a high cost when the system is composed of hundreds of components, implemented in several different programming languages, frameworks, and libraries, and communicating through a multitude of protocols.

2.2 Distributed Tracing

Tracing is a strategy for understanding distributed systems' behavior, implementing optimizations and fixing problems which is not new [8, 15, 20]. In the context of microservices, it enables us to extract and expose the behavior of microservice-based applications by tying up messages exchanged by the microservices. Tracing systems like X-Trace [7] and Dapper [24] adopt this strategy to propagate tracing-specific data traveling within exchanged messages to precisely diagnose causal relationships between exchanged messages. Another characteristic of tracing is the instrumentation of systems' components and the reporting of data produced via instrumentation to a monitoring agent responsible for their later exposure through a convenient interface.

The traditional way for enabling tracing would require that all microservices be instrumented. The first one should be modified in order to generate identifying tags for each one of the incoming user requests and maybe other tracing-related data such as sampling decision. Moreover, all microservices must be changed in order to propagate these data. This enables tracing features such as diagnosis of causality relationship between requests and sampling. Finally, each one of the microservices should report tracing data to a trace server. This strategy is highly dependent on data generated, propagated and reported by components, which strictly targets tracing's enablement and regards two sets of infomation: one with data which must be propagated across application's messages and the other with data which is sent to the trace server.

The first one should be as small as possible as it will impact on application's messages size. It commonly includes the identifying tags and sampling decisions, and travels along with application-specific data, leveraging the protocols application use to exchange the messages between the microservices. The second one also includes identifying tags, which are required for establishing requests' causality, but may not include some data present in the first set, such as sampling decisions. It usually includes timing information but may also include protocol's version, application-specific data, message's attributes (e.g., size, contents), and so on. It is formatted and sent to the trace server respecting a communication protocol which does not need to be the same as the one used for exchanging application-specific messages. Beyond dependence upon specific data, tracing also depends upon code instrumentation for providing aforementioned modification needs. There are many ways how a microservice-based application developer can instrument the source code to enable tracing. She could, for instance, develop ad-hoc instrumentation logic for each microservice, which would certainly represent a cumbersome effort. It would also be possible to leverage libraries but this would still require representative effort, mainly due to the variety of programming languages in a microservices context. We strive for a more adequate approach for enabling tracing of microservices, which is presented in the next section.

3 TRANSPARENT TRACING OF MICROSERVICE-BASED APPLICATIONS

"Use the right tool for the right job", one of the guiding principles of the microservices architectural style, means using many different programming languages, frameworks, and libraries. The result is a highly complex ecosystem of small pieces of software, each one designed and implemented with well-scoped requirements in mind. Hence, these applications need better ways of behavior tracing and bug fixing [6]. Microservices developers and operators might benefit from distributed tracing tools as they allow a better understanding of systems' behavior and performance. However, their adoption in this context demands considerable time and effort as it implies performing changes in several software components, programming languages, and frameworks. Instrumenting them is impractical and using off-the-shelf solutions (when available) is not a good alternative as it still requires knowledge about APIs and configuration properties, and changes in application's source code.

Our proposal tackles the increased cost (i.e., time and money invested in instrumentation-related activities) of enabling distributed tracing for microservice-based applications and advances microservices' tracing by providing developers and operators with means for tracing their applications' behavior and performance without demanding any source code modifications.

We reduce the time spent by developers on instrumentation to a minimum by relying on modifications to deployment schema and monitoring of processes' activities instead of changing applications' source code. The deployment is modified to inject intermediate software components between application's microservices while the monitoring serves to track causalities between multiple requests. This plan is concretized twofold: first, we share guidelines for the deployment and usage of tracing-specialized proxies; then, we provide a strategy for monitoring specific Linux syscalls which allows tracking requests causality and an implementation for it.

Our approach is capable of effectively determining causality of messages generated by users' requests by combining the intermediate components' deployment and processes' monitoring. The general picture is that the proxies generate and report tracing-specific data while the monitoring propagates portions of these data necessary for diagnosing requests causality in order to free applications' code from any instrumentation-related modification.

3.1 Deployment and Usage of Proxies

Microservices are usually deployed through containerization technologies, e.g., Docker[1], in which principles like localization transparency along with its benefits are embodied, e.g., easiness of scaling, moving, and replacing components. This scenario helps the adoption of proxies to intercept all communications by changing the configuration related to microservices' deployment schema.

We use proxies for intervening all network interactions between microservices by deploying one proxy per microservice. One could argue that it would be possible to achieve our goals with less proxying units, but keeping 1:1 ratio helps to simplify and understand the deployment schema. In this way, automation becomes easier, which aligns quite well with microservices' principle of automating everything. In practice, there are no reasons for deploying more than one unit per microservice instance as this strategy increases the resource consumption due to the extra components. Although proxies' resources consumption is reduced, it is not to be neglected.

Proxies in our approach are aware of any message exchanged between microservices and use them for generating and reporting tracing information. Their responsibilities are the same as the ones attributed to instrumentation in the strategy presented in Section 2.2.

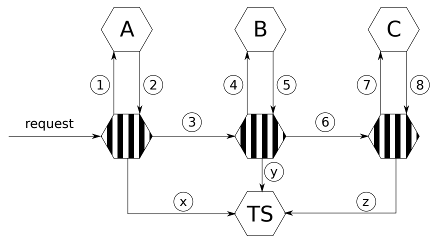

Figure 1: Overview of proxies deployment

Figure 1 shows a scenario in which three microservices, namely A, B, and C, interact with each other to serve a single request. By applying our strategy, we interpose three proxies (striped hexagons) between them and deploy a trace server (`TS`). Circled numbers indicate the order in which communications occur: A receives the user's requests, and requests something to B which in turn sends a request to C. Both incoming (①, ④ and ⑦) and outgoing (②, ⑤ and ⑧) requests are received and sent through proxies, which report them to `TS`. Tracing-related requests (ⓧ, ⓨ and ⓩ) do not need to occur in any particular or specific moment. Responses are omitted for simplicity. Note that proxies shall also inject meta-information used to determine requests causality. For instance, if `request` is an HTTP request lacking an `X-Request-Id` header, the first proxy (one which handles `request`) shall inject the missing header, which should then be propagated by further services and proxies.

This approach is protocol-agnostic as it neither relies upon any protocol-specific information nor structure. However, other protocols demand extra efforts like requiring specific ways for both extracting tracing information and tracking requests causalities. We also do not prescribe any particular protocol for communication between proxies and the trace server (`TS`). To the extent of what we are proposing here, tracing of other protocols would require proxies capable of supporting them which must be properly configured to do so.

The use of proxies for tracing microservices is recent and supported by tools like Istio[2]. However, it lacks transparent tracing as it demands application's code changes to propagate trace meta-information in order to track requests causality.

[1]https://docker.com

[2]https://istio.io

We leverage Linux system calls (syscalls) monitoring for eliminating this shortage.

3.2 Syscalls Monitoring

Syscalls are basic procedures exposed by Linux's kernel to application's processes to perform critical actions like hardware manipulation. By monitoring them, we can track process behavior related to the execution of requests paths and obtain and inject tracing-related meta-information. This approach is how we accomplish meta-information propagation without requiring any modification on application's code.

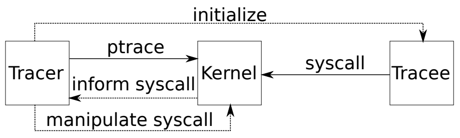

Figure 2: Syscalls interception overview

Figure 2 depicts the elements involved in the syscalls monitoring strategy and the main interactions between them. The Tracer (our monitor process) is responsible for the Tracee (microservices processes) initialization and setup of syscalls monitoring, which is done through the `ptrace`[3] system call. After setup, Linux's Kernel informs the Tracer whenever the Tracee performs a system call.

We identified the syscalls invoked by threads for receiving and sending HTTP requests. Syscalls' execution is always evaluated in the context of a performing thread because we use this association for diagnosing requests causality: we assume that any outgoing request performed by a thread while it is servicing an incoming request is caused by the incoming request. In practice, we intercept the syscall used to receive an HTTP request for extracting tracing meta-information from HTTP headers and syscalls used to execute outgoing requests for injecting tracing meta-information into them. Examples of such syscalls are `read`, which is used for reading an incoming request from a socket, and `sendto`, which is used to send requests.

Any new thread starts in the `idle` state and can receive incoming requests once it calls the `accept` syscall. When this happens, we register the socket by which the thread can receive requests and change its state to `Waiting requests`. In this state, three events may happen: new sockets may be open, open sockets may be closed or a request may arrive. When a new socket is open, we register it and keep the thread in `Waiting requests` state. When the socket is closed, we just remove it from our records and take the thread back to the `idle` state if it hasn't any more open sockets. When a request arrives, we extract and record its tracing headers and change thread's state to `Servicing`. In this state, a thread can `close` sockets, perform outgoing requests (`sendto`) or create new threads (`clone`). When the thread closes sockets, we remove them from our records and take it back to the `idle` state if there aren't any remaining open sockets. If an outgoing request is triggered, we inject tracing headers into it. If new threads are created, we just start monitoring them with the difference that they are already put in the `Waiting requests` state and associated with their parent's open sockets and incoming tracing headers. We use SECCOMP[4] filters to reduce the overhead posed by syscalls' interruptions. These interruptions may be numerous and expensive due to the context switch that happens between kernel- and user-space.

[3]http://man7.org/linux/man-pages/man2/ptrace.2.html

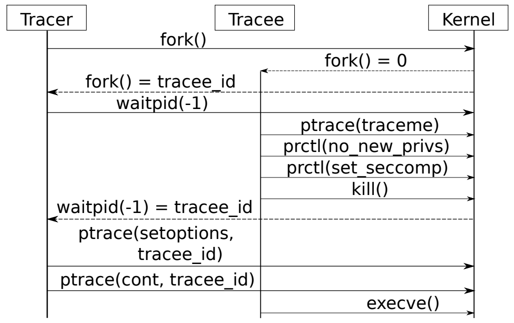

Figure 3: Tracee process initialization.

Figure 3 shows a Tracee setup. Tracer first forks a child process to run the microservice's code. Then, it waits (`waitpid`) for the Tracee to perform essential operations: inform the kernel that it must be traced by its parent (`ptrace(traceme)`); forbid ascent of privilege levels (`prctl(no_new_privs)`), which makes possible to run our monitoring without admin privileges; and install SECCOMP filters (`prctl(set_seccomp)`). The Tracee then stops (`kill`) to allow the Tracer to set necessary options (`ptrace(setoptions)`). These options include flags for making `ptrace` automatically trace further children processes and activating better handling of SECCOMP-related interruptions. Next, the Tracer proceeds to Tracee's execution (`ptrace(cont)`), that executes microservice's code (`execve`).

After Tracee initialization, Tracer keeps an infinite control loop established by `waitpid` syscall. This syscall interrupts the Tracer execution until some of its Tracees makes a syscall that triggers a notification, obeying the rules of installed SECCOMP filters.

Figure 4 depicts the `read` interception for headers extraction from incoming HTTP requests. Despite this is not being represented, `extractheaders` is subjected to some conditional verification before being performed: the `read` syscall which caused the interruption must consider an open TCP socket's file descriptor and `request` must be an HTTP request. `ptrace(peekdata)` must be performed *after* the syscall execution as `request` would not be available. Also, the original syscall is executed without having its parameters or result modified.

[4]https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html

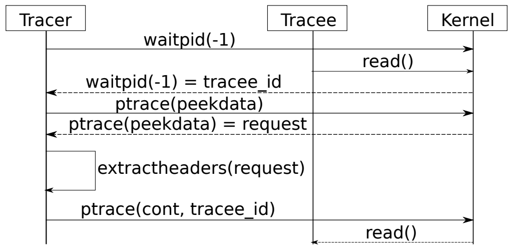

Figure 4: Headers extraction on `read` syscall interception.

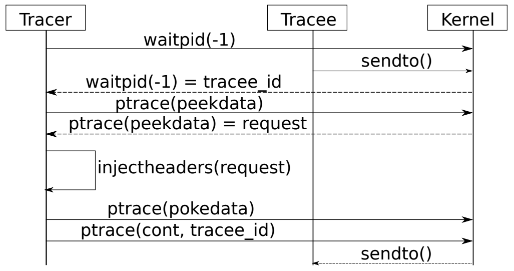

Figure 5: Headers injection on `sendto` syscall interception.

Figure 5 shows the `sendto` interception, which is similar to `read` interception, except for: the `request` retrieved through `peekdata` is an outgoing request instead of an incoming one; and `pokedata` is used to change syscall's paremeters by replacing the original request with the one that holds tracing headers. Hence, as opposed to `read`'s interception, parameters' manipulation occurs *before* syscall execution.

4 EVALUATION

This section presents an evaluation of the proposed monitoring solution.

4.1 Objectives

We carried out a performance evaluation to assess the performance overhead of the proposed solution on the system-under-monitoring. In practice, the performance evaluation targets at i) evaluating the overall overhead imposed by our implementation; ii) comparing it with the overhead imposed by other strategies; and iii) measuring metrics which impact on microservices development and operation.

4.2 Experiments

We used Sock Shop[5] app as experimental workload [3]. We chose applications' *response time* as the metric because it is a good measure of users' experience [16]. Table 1 summarizes the experiments' parameters. The workload counted with the execution of 1.000 POST HTTP requests targeted at the "checkout order operation", which is the application's operation that involves more microservices. For trying to resemble a realistic, low-load, scenario, we also wait for some seconds between each request.

[5]https://microservices-demo.github.io

Table 1: Experimental parameters.

Parameter	Value
Workload	Checkout order
Number of Requests	1.000
Time Between Requests	Randomly generated following gaussian distribution (mean: 5, standard deviation: 2)
Tracing Scenarios	Disabled (*No Tracing*) Conventional (*Instrumented Microservices*) Our strategy (*Rbinder*)

Our experiments considered three distinct scenarios to compare the proposed strategy with other monitoring possibilities: i) the non-traced application (*No Tracing*), ii) the application traced by microservices' instrumentation (*Instrumented Microservices*)[6], and iii) the application using our strategy (*Rbinder*[7]). The non-traced scenario is accomplished through the deployment of microservices with configuration options for disabling tracing. The instrumentation of the second scenario is achieved through third-party libraries and code modifications. Finally, in the third scenario, our strategy (which regards usage of proxies plus syscalls monitoring) comes into play: Envoy Proxy[8] proxies are used to intercept the microservices communication and handle tracing responsibilities; syscalls monitoring is enabled via modification of microservices' Docker images for starting the main process under our monitoring process. This is as simple as prefixing the command used for starting the main process with the command for starting the monitor process. For instance, if the command for starting the microservice is `java -jar orders.jar`, we would issue `rbinder java -jar orders.jar` instead for starting the main process under our monitor process. Both tracing-enabled scenarios used Zipkin[9] trace server. All services were deployed as Docker containers and orchestrated by using Docker Compose.

4.3 Results

Figure 6 shows the results of the experiments. The error bars indicate a 95% confidence interval. As can be observed, the response times of both tracing-enabled scenarios are very close. The mean response time when the application is not being monitored is 0.110s against 0.116s when tracing is enabled by traditional means (source code instrumentation) and 0.121s when our strategy (Rbinder) is in place. This means that the traditional approach is 1.052 times slower than the untraced application while ours is 1.097 times slower.

[6]https://github.com/gfads/microservices-demo/tree/gfads
[7]https://github.com/gfads/microservices-demo/tree/rbinder
[8]https://www.envoyproxy.io/
[9]https://zipkin.io

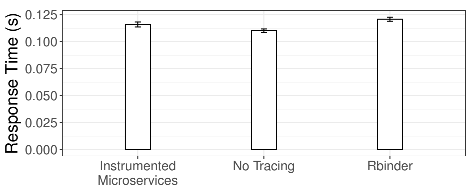

Figure 6: Response time of the three assessed scenarios.

Hence, considering this metric, the impacts of the tracing strategies over the application's performance are very similar.

Figure 7 depicts the CPU usage. Despite the similar usage pattern, it is possible to distinguish a slightly higher usage in the tracing-enabled scenarios. The average CPU usage when the application is not being traced is 17.44%. For instrumented microservices, this average increases to 18.58% (1.07x higher) and for Rbinder it increases to 21.52%(1.23x higher). This increase could be explained by the extra instructions executed for tracing purposes. For our strategy, this amount is still greater as per the syscalls monitoring. For instance, extra instructions are needed to copy buffers from the kernel- to user-space on each syscall interruption.

Figure 8 presents the RAM usage. Here again, consumption is higher in tracing-enabled scenarios, which shall also be explained by tracing-related memory allocation. For instance, the trace server demands a lot of memory. Also, memory usage in Rbinder is influenced by the deployment of numerous proxies.

Relying upon off-the-shelf libraries for applications' instrumentation may lead to some unintended consequences like a considerable increase in components binaries' size and time to start them. Figure 9 depicts the size of the files used for initializing Sock Shop's microservices. For Go and Java microservices, these are the binaries resulting of code compilation and JAR archives, respectively. Results show that instrumentation led to bigger binaries for all microservices, especially the Java ones, in which case using our solution for tracing produces binaries 25% smaller on average.

We also measured the time to microservices become operational, from an application's user point-of-view. This time is relevant because it has a direct impact on developers' and service operators' productivity. For instance, developers will spend much more time if they have to wait for longer initialization periods within their development environment. Also, operations' activities such as replicating instances for services' scalability are damaged if microservices take too long for starting up. Figure 10 presents the results. There is no statistically significant difference for the initialization time when the microservices written in Go (`user` and `payment` are considered. The `user` service presented an average initialization time of 2.21s (standard deviation = 0.27) for the instrumented scenario and 2.22s (std dev = 0.17) for Rbinder. `payment` took 2.49s on average (std dev = 0.10) to initialize in instrumented scenario and 2.51s (std dev = 0.09) in Rbinder. However, there is a great difference in initialization time when Java services are regarded. `carts` took 46.50s (std dev = 1.20) to initialize in instrumented scenario and 30.33s (std dev = 0.78) in Rbinder, `orders` needed 46.11s (std dev = 0.98) if instrumented and 32.07 (std dev = 1.90) when monitored by Rbinder, and `shipping` took 44.09 (std dev = 0.99) when instrumented versus 28.20s (std dev = 0.33) when Rbinder was active.

The evaluation shows that Rbinder poses slightly higher performance overhead and uses more RAM. Nevertheless, it does not require any intervention in the application's code to enable monitoring. It also avoids some common drawbacks of conventional instrumentation strategies such as larger binaries and slower microservices' initialization. Therefore, the benefits regarding easiness of adoption and reduction of costs related to instrumentation code development and maintenance might compensate for the drawbacks on performance and resources consumption.

5 RELATED WORK

Existing work leverages distributed tracing for supporting microservice-based applications' evolution [22]. The experience sheds light over the need of a more lightweight way for enabling tracing of fine-grained architectures like microservices. This need is further confirmated by works such as the one by Neves and Pereira [17], which intend to use tracing for modelling the behaviour of systems architected this way. Having this in mind, we organized existing solutions around the two pillars of our solution: absence of code instrumentation and use of syscalls.

5.1 Code Instrumentation Strategies

Carosi [4] correlates tracing and profiling data to better support debugging activities. Similarly to our approach, this work uses proxies for handling tracing-related data generation and report. However, Carosi leverages web server's modifications for tracing headers propagation while we use syscalls monitoring and interception.

Monere [27] and Pinpoint [5] use code instrumentation to collect information that helps in service discovery and fault detection, respectively. The former statically extracts dependency structures from the documentation, and the latter relies on middleware-level instrumentation for logging which components processed each request. Our strategy is more generic in the sense that it traces information regarding generic systems' behaviours. Moreover, we rely on platform-level instrumentation solely for tackling the highly heterogeneous environment of microservice-based applications.

More similar to our strategy, X-Trace [7] and Dapper [24] trace requests path to understanding the systems' behaviour and aid in debugging of performance issues. They rely upon application-, library- and middleware-level instrumentation for propagating tracing metadata used for tying up related requests. Our proposal also depends on metadata for keeping

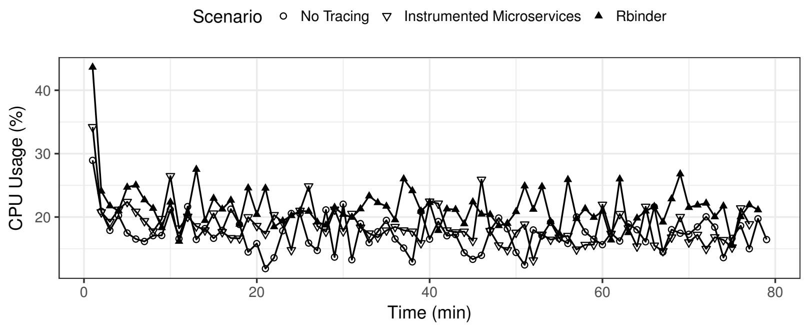

Figure 7: CPU Usage.

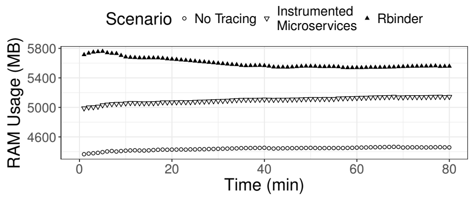

Figure 8: RAM usage.

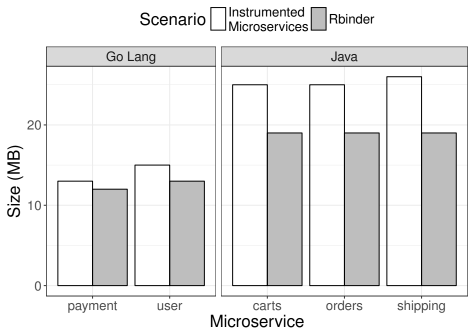

Figure 9: Size of microservices' binaries.

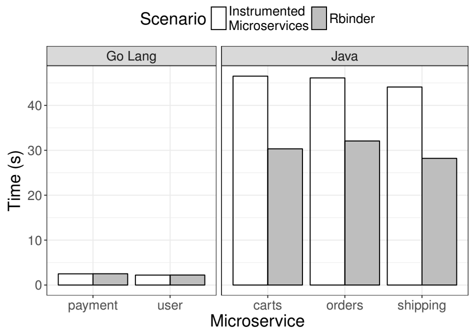

Figure 10: Microservices initialization time.

track of requests causality. However, we avoid code instrumentation by using syscalls monitoring for metadata propagation and proxies for metadata generation and reporting.

Mace and Fonseca [13] propose a strategy for decoupling tracing's enablement and usage so that the same instrumentation efforts applied for enabling a system's tracing can be used for multiple tools relying on tracing data. Their work relates to ours in a sense it also intends to mitigate the pain of tracing fine-grained distributed systems, but it still relies upon instrumentation.

Kitajima and Matsuoka [10] and Aguilera et al. [1] also propose strategies for tracing requests paths. The former uses an imprecise heuristic while the latter proposes two algorithms for diagnosing causality between requests. These approaches have the benefit of grasping systems' behaviour without requiring propagation of tracing information. Meanwhile, we pay the price of precise request tracing by propagating such information via the operating system's syscalls monitoring. Moreover, Aguilera relies upon (middleware-level) instrumentation to some degree, which we also avoid as a general principle due to the potential it has to increase the strategy's adoption difficulty in a microservices-based environment.

5.2 Use of Syscalls Strategies

Ardelean et al. [2] uses a tracing strategy to collect data in a given period to understand the system's load variations. They also work with kernel-level calls for diagnosing causalities between messages exchanged by components at different levels of the system's stack. Unlike our proposal, they need to instrument the code to inject kernel-level calls.

Xu et al. [28] apply a learning-based algorithm to diagnose causality between multiple requests. Similarly, as we do, they also leverage kernel's syscalls but do so in a more passive manner: kernel's event traces are used to feed the learning algorithm. Such strategy results in an imprecise means for causality detection (as opposed to the one we seek here, which is a precise one).

Finally, Callahan et al. [19] use a similar approach to solving a different problem. They leverage syscalls monitoring and interception to allow re-execution of monitored programs.

6 CONCLUSION

This paper presented a strategy for tracing microservice-based applications which has transparency as its ultimate design goal. The solution was motivated by the constraints imposed by fine-grained service-oriented architectures: instrumentation cost is prohibitively expensive in a scenario of such great technological heterogeneity. Our strategy is grounded on two main ideas: the usage of proxies for relieving the burden of tracing-related activities from applications' code and the complimentary, and necessary, approach for propagating tracing-related meta-information through monitoring Linux's system calls. The results show that our strategy imposes overhead similar to other tracing solutions without requiring source code instrumentation and favors microservices operations because it promotes smaller binaries and faster services initialization.

Future work shall strive for performance improvements of the proposed strategy. To this end, we need more experiments regarding a broader range of workloads and parameters evaluation. For example, it would be beneficial to have a grasp on network impact due to communication overhead. Also, there are some traits presented in related works that could be pursued. For instance, it is an open question how the introduced strategy could be extended to support tracing of application-level information. Finally, we suggested our approach to be protocol-agnostic. With the increasing adoption of protocols other than HTTP for implementing communication between services, it would be useful to explore how our strategy could be used for tracing microservices leveraging further protocols.

REFERENCES

[1] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 74–89.

[2] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation NSDI 18)*. USENIX Association.

[3] Antonio Brogi, Andrea Canciani, Davide Neri, Luca Rinaldi, and Jacopo Soldani. 2017. Towards a reference dataset of microservice-based applications. In *Proceedings of the 1th Workshop on Microservices: Science and Engineering (MSE 2017). Springer.*

[4] Robert Carosi. 2018. Protractor: Leveraging distributed tracing in service meshes for application profiling at scale.

[5] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on.* IEEE, 595–604.

[6] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216.

[7] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association, 20–20.

[8] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. 1987. Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)* 5, 2 (1987), 121–150.

[9] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 34–50.

[10] Shinya Kitajima and Naoki Matsuoka. 2017. Inferring Calling Relationship Based on External Observation for Microservice Architecture. In *International Conference on Service-Oriented Computing*. Springer, 229–237.

[11] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. *MartinFowler. com* 25 (2014).

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[13] Jonathan Mace and Rodrigo Fonseca. 2018. Universal context propagation for distributed system instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 8.

[14] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 378–393.

[15] Friedemann Mattern et al. 1989. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms* 1, 23 (1989), 215–226.

[16] Daniel A Menascé. 2002. QoS issues in web services. *IEEE internet computing* 6, 6 (2002), 72–75.

[17] Francisco Neves and José Pereira. [n. d.]. Holistic performance analysis for large-scale distributed systems. ([n. d.]).

[18] S Newman and Building Microservices. 2015. O'Reilly Media Inc. *Building Microservices* (2015).

[19] Robert OCallahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC17). USENIX Association, Berkeley, CA, USA*. 377–389.

[20] Michel Raynal and Mukesh Singhal. 1996. Logical time: Capturing causality in distributed systems. *Computer* 29, 2 (1996), 49–56.

[21] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. 2011. Diagnosing Performance Changes by Comparing Request Flows.. In *NSDI*, Vol. 5. 1–1.

[22] Adalberto R Sampaio, Harshavardhan Kadiyala, Bo Hu, John Steinbacher, Tony Erwin, Nelson Rosa, Ivan Beschastnikh, and Julia Rubin. 2017. Supporting Microservice Evolution. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on.* IEEE, 539–543.

[23] Dhruv Sharma, Rishabh Poddar, Kshiteej Mahajan, Mohan Dhawan, and Vijay Mann. 2015. H ansel: diagnosing faults in openStack. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 23.

[24] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report. Technical report, Google, Inc.

[25] Byung-Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N Chang. 2009. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities.. In *USENIX Annual technical conference*.

[26] Zhonglei Wang, Antonio Sanchez, and Andreas Herkersdorf. 2008. Scisim: a software performance estimation framework using source code instrumentation. In *Proceedings of the 7th international workshop on Software and performance*. ACM, 33–42.

[27] Bruno Wassermann and Wolfgang Emmerich. 2011. Monere: Monitoring of service compositions for failure diagnosis. In *International Conference on Service-Oriented Computing*. Springer, 344–358.

[28] Hongteng Xu, Xia Ning, Hui Zhang, Junghwan Rhee, and Guofei Jiang. 2016. Pinfer: Learning to infer concurrent request paths from system kernel events. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on.* IEEE, 199–208.