Profiling Microservices

John Humphries, Konstantinos Kaffes
Stanford University

***Abstract*—We present a method for tracing the execution of RPC calls between microservices to understand the call tree and determine where bottlenecks arise. We show how the metadata propagated through our microservices system can later be reconciled and automatically rendered into a clear and digestible visual of the system. In our setup, we use Nameko - a popular, open-source microservices framework written in Python - to run microservices and propagate metadata throughout the system. We then show that our technique results in little additional overhead on the system and on administrators using our tool.**

Keywords: microservices, profiling, tracing, nameko

I. Introduction

In recent years cloud computing has been moving towards finer granularity. Initially, providers offered bare metal machines that clients had to reserve for long periods of time. However, this model was not economically viable for both users and providers. The former might want to use processing power intermittently or for short periods of time but had to pay as if they used it continuously while the latter would have tons of unused capacity they could not resell to other clients. That is why the cloud providers started offering Virtual Machines (VMs). This allowed them to multiplex different clients on the same hardware and increase utilization while clients could power off their VMs and not pay while they are not using them. The next step was the introduction of containers: enclaves of software within the same operating system that use various operating system features to ensure isolation and security. Containers are more lightweight than virtual machines and boot much faster, allowing faster scaling of services. The most recent trend in cloud computing is serverless. While technically it still uses containers, the programming model is simpler and more developer-friendly. Users upload functions written in some high level language which are triggered by events, such as the upload of a photo to a social network. Even more recently, the immense scaling cloud providers offer for these functions has been used for scientific and data analytics workloads [7] that require immense parallelism. However, under the hood, each function still runs in its own container.

Similar changes took place in the software design domain. Initially, most code development was monolithic. In monolithic applications, all components of a complicated system are developed, compiled, and deployed together. The more complex an application becomes and the more developers work on it, the harder it is to keep it under control. Developers are bound to the same stack, e.g. .NET, while different languages/frameworks might be more suitable for different components of the application. It is also impossible for a single team to have an understanding of the entire application. The aforementioned reasons are why big enterprises started moving towards tiered architectures. In applications following this type of architecture, user interface, request processing and data management are both physically and logically separated. Hence, each of these layers can be reused among different applications and modified separately. The natural extension of this design choice is to break each application layer even further into its individual components. This architectural style - applications consisting of loosely coupled independent services - is called microservices.

The emergence of tiered and microservices architectures raised issues regarding monitoring, tracing and root cause analysis that did not even exist in traditional monolithic applications. The fact that microservices can run on different machines, and even in different datacenters, makes these problems more challenging. In this paper, we propose a tracing mechanism for a particular microservices framework, Nameko, measure the overhead it incurs, and present several cases that highlight its use and functionality.

In the next section, we present microservices in general, and Nameko in particular, in more detail. Then, we analyze the tracing mechanism we implemented and analyze its overhead. In section 4 we show how it can be used in several cases. Finally, we compare against related work and draw some conclusions.

II. Microservices

According to Netflix Cloud Architect Adrian Cockroft, microservices architecture is service-oriented architecture composed of loosely coupled elements that have bounded contexts [14]. Each component, called a microservice, fulfills a simple and self-contained role. ”Loosely coupled” means that services are developed and updated independently. We say that microservices have bounded context if they are self-contained and use strict APIs to interact with each other without sharing object representations and data stores. Having separate data stores for each microservice is important as making a schema update might break compatibility with other services that share the same schema. Also, using microservices allows developers to avoid some of the worst complications of distributed systems. Servers are stateless and therefore interchangeable within the same service. State is pushed down to the data store, which handles replication and availability automatically. Hence, the only concern of system administrators is scaling the number of servers associated with each microservice according to the load.

Microservices have been used extensively in recent years, in fields ranging from e-commerce [4] to video streaming. One of the most prominent and successful adopters of microservices is Netflix. Struggling to scale its monolithic architecture to cope with its immense growth rate, Netflix switched to microservices hosted in a public cloud (AWS). Uber, having to cope with similar growth and requiring high development velocity, has distributed its functionality across more than 500 microservices. Seeking efficient and secure ways to implement inter-service communication, Uber engineers are using Apache Thrift [6]. Thrift is a software library and code generation tools developed at Facebook with the goal of enabling efficient and reliable communication across programming languages and frameworks. It allows developers to define data types and interfaces in a simple, declarative language.

However, not all companies have the resources to develop their own microservices substrate. Therefore, many frameworks have been developed that allow programmers to focus on the logic of their applications instead of having to handle low-level distributed systems problems such as communication, replication, and reliability. There is a large variety of such frameworks implemented in many different languages. Micro [12] is a microservices ecosystem that provides support for RPC, service discovery, load balancing, and synchronous and asynchronous communication. Gizmo [11] is a similar toolkit implemented by the New York Times IT team, while Go-kit [] focuses on interoperability and interaction with disparate non-Go-kit services. Finally, there are many JVM-based frameworks such as Dropwizard [13]. Due to familiarity with the Python language, we decided to examine a framework based on Python: Nameko [9].

A. Nameko

Nameko is a popular microservices framework developed in Python. It uses RabbitMQ, a message queuing system, to propagate requests among microservices. It supports:

- Remote Procedure Calls over AMQP
- Asynchronous events over AMQP
- Simple HTTP GET and POST requests
- Websocket RPC and subscriptions

Nameko allows users to build a service that can respond to RPC messages, dispatch RPC messages and events on certain actions, and listen for events from other services. Nameko also allows users to define their own transport mechanisms and service dependencies. Each microservice is implemented as a Python class with the special rpc decorator denoting methods that can be called through RPC. A key feature of Nameko is dependencies which allow services to abstract away access to other services and service/system components. Dependencies are added to a service class declaratively and are responsible for providing objects that are injected into service class instances. These instances, called workers, are stateless and are created when an entry point (RPC call, HTTP request or publication in some queue) fires. The life cycle of a worker is as follows:

- Entry point fires
- Worker is instantiated from service class
- Dependencies are injected into the worker
- Method executes
- Worker is destroyed

A significant shortcoming of Nameko and many Python frameworks in general is that they provide concurrency but not parallelism. Each worker executes in its own ”green thread,” i.e. userspace thread, and these threads are multiplexed on the same kernel thread. The scheduling of green threads is cooperative (i.e. they yield either when they finish execution or when they block on I/O). The advantages of this approach is that there is no need for locking (the workers are stateless anyway) and userspace threads spin up much quicker than kernel-space threads. In listing 1 we see an example of a Nameko service that has an RPC entry point and depends on another service.

```python
from nameko.rpc import rpc, RpcProxy


```

```
class IdService:
  name = 'id_service'

    @rpc
    def get_id(self):
      return 0

class GreetingService:
    name = 'greeting_service'

  id_service = RpcProxy('id_service')

    @rpc
    def hello(self):
        return "Hello, {} with id {}!".
   format(name, self.id_service.get_id
   ())
```

Listing 1: Nameko example

Nameko's inter-service communication relies on RabbitMQ [10]. RabbitMQ is a message queuing framework that works with a variety of programming languages and features. It uses the Advanced Message Queuing Protocol to send messages efficiently among applications. AMQP is a wire-level protocol for high performance networking. It is an open standard and therefore it can be used by different services and allows Nameko to work with them. RabbitMQ also supports different types of distributed brokers for scaling, replication, and availability.

III. Profiling Nameko

As mentioned before, a system implemented as a collection by many microservices can be large and difficult to debug, particularly as the microservices interact with each other. A tracing framework/profiler would allow users and administrators to answer the following very important questions:

- If a request fails, which microservice caused the failure?
- If a request is slow, where is the bottleneck of the system? How do bottlenecks appear and disappear with different patterns of network traffic?
- How can one understand the layout and implementation of a large system when there is little or no documentation?

We provide answers to these questions by implementing a tracing/profiling framework on top of Nameko. Metadata are inserted at different points during the processing of an RPC call. These metadata include timestamps, a hierarchy of RPC calls, and the host names of the machines in which the calls are executed. Both synchronous and asynchronous RPC calls are supported.

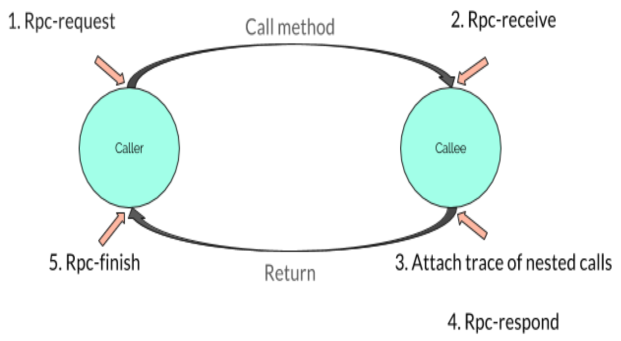

Fig. 1: Metadata injection points during an RPC execution.

We also created script that can render the metadata to produce a convenient and illustrative visualization of how each request flowed through a distributed system along with how much time was spent processing in each service.

In Figure 1 we see exactly when metadata are injected during the execution of an RPC. All entries include the aforementioned metadata. When an RPC call is made, an rpc-request entry (1) is added to the message headers. When the RPC is received by the callee, the metadata headers together with an rpc-receive entry (2) are saved in the service worker's context. The worker proceeds with its execution, making nested RPC calls if necessary. Once all processing is finished, the traces of the nested RPC calls are added to the worker's context (3) together with an rpc-respond entry (4) and are included in the RPC reply's headers. Finally, when the result of the RPC is received by the caller, an rpc-finish entry is recorded (5). Initially, we propagated each worker's context traces as metadata during nested calls but we realized that (a) it incurred a lot of overhead and (b) it did not offer anything in terms of functionality.

When the original client that made the initial RPC request receives a response, it parses the metadata from the response headers and writes the metadata to a local file. Each metadata entry contains 5 key-value pairs, where each key is 11 bytes or fewer and each value is 36 bytes or fewer (though may be longer if the server hostname, the service name, or the RPC method name is longer than 36 bytes). The five keys are the server hostname, the service name, the RPC method name, the current time, and a unique identifier associated with the RPC.

To a system administrator, reading through a potentially long and complicated metadata file would be tedious and unhelpful. Thus, we wrote a Python script that can parse the metadata file and render a simple and

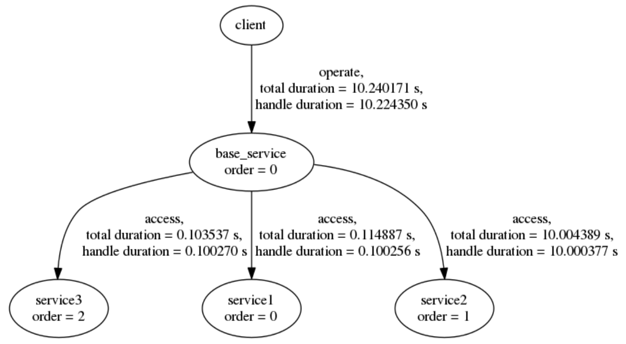
client
operate,
total duration = 10.240171 s,
handle duration = 10.224350 s
base_service
order = 0
access,
total duration = 0.103537 s,
handle duration = 0.100270 s
access,
total duration = 0.114887 s,
handle duration = 0.100256 s
access,
total duration = 10.004389 s,
handle duration = 10.000377 s
service3
order = 2
service1
order = 0
service2
order = 1

Fig. 2: Call-tree PNG image generated from Straggler application. The original application contained 10 leaf services. We shortened it to 3 leaf services to make the tree easier to fit. The total duration is the length of time between the timestamps in the rpc-request and rpc-finish entries for a given RPC call. The handle duration is the length of time between the timestamps in the rpc-receive and rpc-respond entries (i.e. the amount of time spent processing on the server).

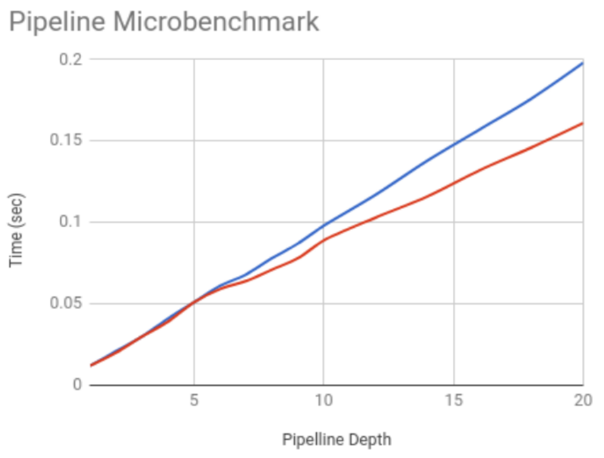

Fig. 3: Metadata overhead for a deep call stack. The orange line is the time without the metadata. The blue line is the time with the metadata.

easy-to-understand PNG image of the call tree that was spawned by the original client's initial request along with the time each server spent processing a request. The PNG image generated for the Straggler application is given in Figure 2.

In the metadata file, each RPC has an rpc-request entry and an rpc-finish entry, as shown in Figure 1. In step (3) of Figure 1, metadata from any nested RPC calls is inserted into the log. Thus, any entries between the rpc-request entry and the rpc-finish entry for a given RPC call *a* must correspond to nested RPC calls triggered by *a*. The script uses this structure to determine parent-child relationships for the generated tree.

It would also be possible to simply include the unique ID of an RPC call's parent in the metadata. However, this is not necessary in order to recover parent-child relationships, even for asynchronous RPC calls (since the metadata structure is still preserved even in asynchronous RPC calls). Thus, in order to avoid propagating redundant information through the network and unnecessarily complicating our framework design, we did not include this information.

IV. Evaluation

We evaluate our tracing system in a commodity laptop. We determine tracing's overhead by running some micro-benchmarks, we use it to detect straggler services and finally we profile a business application.

A. Microbenchmarks

We will now consider two different microservice layouts and discuss the metadata overhead of each.

First consider a service that consists of a deep call stack. The client makes an RPC request to service 1, which in turns makes a nested RPC request to service 2, which in turns makes a nested RPC request to service 3, and so on to some nonzero natural number n. The metadata overhead is shown in Figure 3. There is negligible overhead for small to medium depths, but the overhead becomes significant when the depth exceeds 15 levels.

The overhead becomes significant for a large number of levels due to the growing amount of metadata that needs to be propagated from the bottom level back to the client. Perhaps this overhead could be decreased by instead sending the metadata from each service directly back to the client, and including pointers to this metadata in the metadata that is sent from service to service back up the chain.

Next consider a service that has wide fanout. The client makes an RPC request to a base service, which in turn makes nested RPC requests to a large number of leaf nodes. The longest chain in this service is of length 2 (client $\rightarrow$ base service $\rightarrow$ leaf service). As shown in Figure 4, there is nearly constant overhead regardless of the fanout factor.

B. Straggler detection

Our tracing framework is helpful for detecting and identifying straggling microservices, i.e. services that

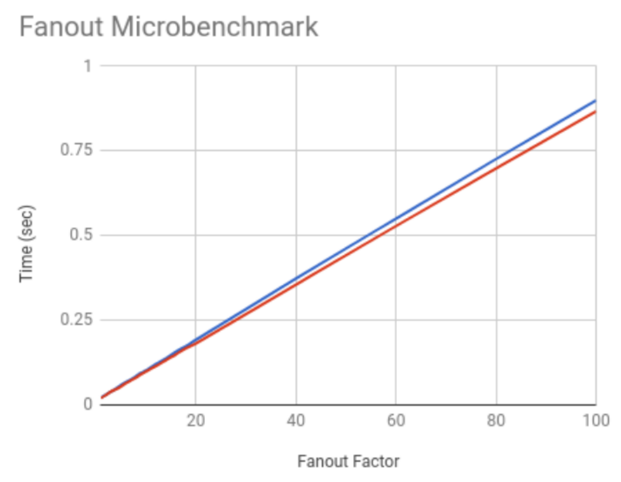

Fig. 4: Metadata overhead for a large fanout. The orange line is the time without the metadata. The blue line is the time with the metadata.

perform significantly worse than their peers, in a large system. Consider the setup shown in Figure 2. A client makes an RPC request to a base service, which in turn makes nested RPC requests to several leaf services. If one of the leaf services is a straggler, the entire response to the client will be delayed. The visual generated from the tracing framework metadata can be used to determine where the request spends the longest time in the system. In Figure 2, the request is held up by service2.

C. Real world application

We use our tracing framework to profile a business application provided by the Nameko framework. In this scenario we have 3 Nameko services that implement the business logic of an online store. The Gateway is a service exposing an HTTP API to be used by external clients e.g., web and bobile Apps. It coordinates all incoming requests and composes responses based on data from underlying domain services. The Orders service is responsible for storing and managing orders information and exposes a Nameko RPC API. This service is using PostgreSQL database as its backend. Finally, the Products service is responsible for storing and managing product information and similar to Orders exposes an RPC API used by other services. This service has Redis, a popular key-value store as its data store. In Figure 6 we see how much time is spent in each service for different request types. We can use this information to optimize such systems and eliminate bottlenecks.

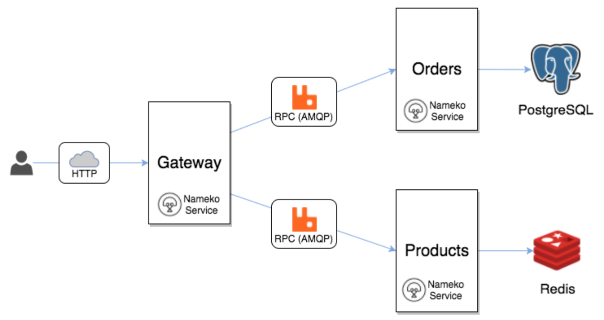

Fig. 5: Microservices composing a business application.

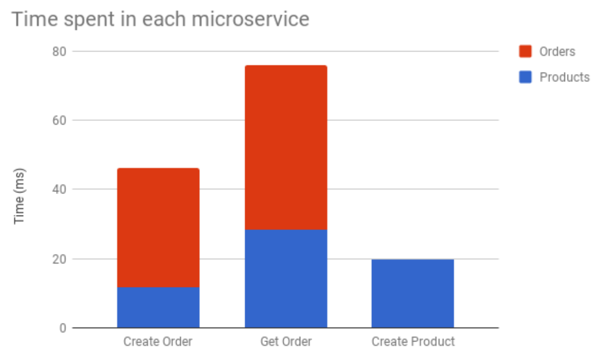

Fig. 6: Execution time break down for different request types in a business application.

V. Related Work

There are many tools that monitor the performance of individual nodes in a distributed system and provide metrics to users and administrators. Some of them focus on networking, either at the machine or the datacenter level [5]. Magpie [3] is a toolchain that gathers and processes events from operating system, middleware and application-level instrumentation. It provides a total ordering of events that can be used to produce a graph of the paths followed through a system. Other solutions such as Splunk [8] aggregate and index the logs of a large-scale system. In this case it is possible to recreate the path of tasks by following metadata such as IP addresses and message IDs. However, such approaches usually require a post-mortem or offline analysis of the logs without being able to preemptively diagnose problems. Next, we focus on the two systems that bear the most similarity to our work.

1) X-Trace: Similar to Magpie, X-Trace [2] is an integrated tracing framework that spans across different layers of the system stack. Metadata are added to each application-level request and are propagated to lower levels while keeping a single request identifier. This requires modifications to various network protocols so that metadata are propagated and added during each network action. X-Trace allows the separation of tracing and monitoring domains which enables some data to be delivered to the end-user and some to the ISP. The required modifications may have inhibited the widespread adoption of X-Trace as they need to be approved by various standards' organizations and committees. Also, with today's high-level, stateless, and containerized applications, low-level tracing might not be necessary. Cloud-hosted services that face performance issues can migrate very quickly and easily to different physical machines.

2) Pivot Tracing: The problem with all aforementioned systems is that the instrumentation is static. The decision of which events are recorded and reported to the users and administrators is made offline. Pivot Tracing [1] gives users - at runtime - the ability to define arbitrary metrics at various points of the system, and select, filter, and group by events meaningful at other parts of the system, even when crossing component or machine boundaries. It has been implemented in Java and evaluated in Hadoop pipelines consisting of various applications such as HDFS, MapReduce, etc. Pivot Tracing also proposes a "happened before join" operator that queries can use to group and filter events based on events that causally precede them during execution. This operator is implemented by using a metadata propagation mechanism similar to the one we propose in this paper.

VI. Conclusion

As applications become increasingly more complex, microservices adoption is spreading. This adoption has raised challenges regarding monitoring, tracing and performance analysis. In this paper, we try to address these issues by implementing a tracing framework for Nameko, a microservice framework built in Python. We show that the overhead added to the request execution time by tracing is minimal in most cases, and we present several scenarios that highlight our profiler's usefulness. Our system is currently tracing only the break-down of the execution time of each request it receives. Our next steps would be to add other metadata to keep track of the system's state, e.g. number of in-flight requests. We can also get ideas from the X-Trace and Pivot Tracing papers. It should be simple to add dynamic instrumentation since we are using Python, an interpreted language. We can also implement a metadata-collecting server used by system administrators instead of just sending all data back to the client. Lastly, we should add an authentication mechanism to ensure that clients are authorized to receive metadata from the system, to ensure that untrusted clients aren't given an unfettered view (through the metadata) of the private network that the system runs on.

VII. Acknowledgement

We would like to thank Prof. Mazieres for teaching a great class and his supervision and feedback on this project.

References

[1] 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 (SOSP '15). ACM, New York, NY, USA

[2] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. *X-Trace: A Pervasive Network Tracing Framework.* In NSDI 2007.

[3] Barham, P., Donnelly, A., Isaccs, R., and Mortier, R. *Using Magpie for Request Extraction and Workload Modeling.* In Proc. USENIX OSDI (2004)

[4] Hasselbring, W. and Steinacker, G. (2017) *Microservice Architectures for Scalability, Agility and Reliability in E-Commerce.* In: IEEE International Conference on Software Architecture 2017, April 03-07, 2017, Gothenburg, Sweden.

[5] Hussain, A., Bartlett, G., Pryadkin, Y., Heidemann, J., Papadopoulos, C., and Bannister, J. *Experiences with a continuous network tracing infrastructure.* In Proc. MineNet '05 (New York, NY, USA, 2005), ACM Press.

[6] Slee, M., Agarwal, A., and Kwiatowski, M. *Thrift: Scalable Cross-Language Services Implementation.* Tech. rep., Facebook, Palo Alto, CA, USA, April 2007.

[7] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht, *Occupy the cloud: Distributed Computing for the 99%*, in ACM Symposium on Cloud Computing, 2017

[8] Splunk, http://www.splunk.com

[9] Nameko, https://nameko.readthedocs.io

[10] Rabbit, https://www.rabbitmq.com

[11] Gizmo, https://github.com/NYTimes/gizmo

[12] Micro, https://micro.mu

[13] DropWizard, http://www.dropwizard.io

[14] Nginx Blog, https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices