MiSeRTrace: Kernel-level Request Tracing for Microservice Visibility

Thrivikraman V*
PES University
Bengaluru, Karnataka, India
thrivikraman.vs@gmail.com

Vishnu R. Dixit*
PES University
Bengaluru, Karnataka, India
rvdixit23@gmail.com

Nikhil Ram S*
PES University
Bengaluru, Karnataka, India
nikhilsram.off@gmail.com

Vikas K. Gowda*
PES University
Bengaluru, Karnataka, India
vikasgowdaoffl@gmail.com

Santhosh Kumar Vasudevan
PES University
Bengaluru, Karnataka, India
santvasu@gmail.com

Subramaniam Kalambur
PES University
Bengaluru, Karnataka, India
subramaniamkv@pes.edu

ABSTRACT

With the evolution of microservice applications, the underlying architectures have become increasingly complex compared to their monolith counterparts. This mainly brings in the challenge of observability. By providing a deeper understanding into the functioning of distributed applications, observability enables improving the performance of the system by obtaining a view of the bottlenecks in the implementation. The observability provided by currently existing tools that perform dynamic tracing on distributed applications is limited to the user-space and requires the application to be instrumented to track request flows. In this paper, we present a new open-source framework MiSeRTrace that can trace the end-to-end path of requests entering a microservice application at the kernel space without requiring instrumentation or modification of the application. Observability at the comprehensiveness of the kernel space allows breaking down of various steps in activities such as network transfers and IO tasks, thus enabling root cause based performance analysis and accurate identification of hotspots. MiSeRTrace supports tracing user-enabled kernel events provided by frameworks such as bpftrace or ftrace and isolates kernel activity associated with each application request with minimal overheads. We then demonstrate the working of the solution with results on a benchmark microservice application.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

request tracing, kernel tracing, microservice, MiSeRTrace, Thread State Model

*These authors contributed equally to this research.

ICPE '22 Companion, April 9–13, 2022, Bejing, China

ACM ISBN 978-1-4503-9159-7/22/04…$15.00
https://doi.org/10.1145/3491204.3527462

ACM Reference Format:
Thrivikraman V, Vishnu R. Dixit, Nikhil Ram S, Vikas K. Gowda, Santhosh Kumar Vasudevan, and Subramaniam Kalambur. 2022. MiSeRTrace: Kernel-level Request Tracing for Microservice Visibility. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22 Companion), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3491204.3527462

1 INTRODUCTION

Most of the recent client-server web applications are adopting microservice architectures due to their benefits like modularity and scalability [11] over their monolith counterparts [17]. The highly networked microservice architecture brings in challenges of visibility into the underlying functioning of the application, introduces overheads [16], and influences server design [12]. Our solution enables the required observability of the application through dynamic tracing at the kernel space. Dynamic tracing is the process of tracking the end-to-end path of user requests from the time the request hits the load balancer of the application until a response to that request is sent out. MiSeRTrace[1] (**Mi**cro**Se**rvice **R**equest **Trace**) provides insights into how every client request is serviced in the kernel which aids in identifying and understanding performance differentials.

Containers in a microservice application communicate with other containers through lightweight API calls which are internally TCP network transfers. Every container in a microservice application is a process, and this process forks many threads to handle the incoming requests. MiSeRTrace primarily tracks these events in order to isolate the path of every client request through the application. This level of monitoring avoids the requirement of tracing libraries such as the OpenTracing API [7] for instrumentation or proxy sidecars like Istio [5]. The tool utilizes certain static and dynamic kernel tracepoints to track the above events. Monitoring these tracepoints is enabled by kernel tracing utilities such as bpftrace [1] and ftrace [4]. Bpftrace is a high level tracing language based on extended Berkeley Packet Filter (eBPF) [3], which allows programs to execute sandboxed code in the kernel. Ftrace is a tracing framework that is built into the Linux kernel and provides visibility primarily into kernel functions and static events. Henceforth, we will refer to the data captured by these frameworks as trace logs.

[1] https://github.com/MiSeRTrace/MiSeRTrace

Our implementation supports both bpftrace and ftrace as tracing backends. Apart from tracepoints utilized by our framework, users can enable any of the tracepoints and features provided by bpftrace/ftrace that they wish to monitor. MiSeRTrace initially identifies the request spans, i.e. the duration spent by a particular thread on servicing a client request or a subsequent internal request. It then associates all these request spans with unique traces. MiSeRTrace subsequently buckets all the time-stamped user-enabled trace logs into the request spans that triggered them to assist in latency estimate studies.

2 RELATED WORK

When it comes to dynamic tracing, there are various tools that are widely used to monitor hotspots in the application. The OpenTracing API is one such framework that utilizes Trace IDs and Span IDs to achieve application instrumentation. A trace is used to uniquely identify all operations performed for an incoming client request, and a span is used to refer to a single operation within a trace. Jaeger [6] and Zipkin [10] are a couple of dynamic request tracers that utilize the Opentracing API. They are open source tracing tools that are used for monitoring and troubleshooting large scale distributed microservice applications. Kieker [14] is another tracing software used to monitor and analyze the performance of monolithic and microservice applications and performs request tracing with the help of measurement probes. Dapper [15], which is Google's distributed system tracing infrastructure, relies on the instrumentation of RPC libraries, control flow libraries, and threading. Dapper is capable of identifying distributed control paths with almost no intervention from the developers.

These tools make use of low latency datastores where the data is frequently pushed to and the developers can gather this information generally through a web based interface. However, they provide insights that are limited to the user space i.e they provide information about the path of the request through the various containers in the microservice and corresponding latency estimates. All kernel activity as a consequence of the client request is mostly unobserved. These frameworks also require the application being monitored or underlying libraries to be instrumented. Both the above issues are overcome by MiSeRTrace by providing the capacity to observe all kernel activity of an un-instrumented application.

3 ARCHITECTURE

The end-to-end usage and working of MiSeRTrace involves 4 main steps which are broadly illustrated in Figure 1.

(1) Trace Specification: MiSeRTrace currently supports bpftrace and ftrace as tracing backends. Events built into the kernel, tracers built into ftrace, custom user probes and events are all supported. The events required by the tool to trace requests through the application are enabled by default. The user can specify a trace configuration to observe kernel activity beyond the default setting. Trace environment specifications like PIDs of the application to be tracked, tracking of forked processes, clocks, and buffer sizes can be configured.

(2) Workload Generation and Tracing of the containerized microservice application: The specified kernel activities are efficiently recorded into the bpftrace/ftrace in-memory ring buffer during the workload execution. On tracing the select events required by MiSeRTrace, there is a reduction of request throughput by around 5-6%. Further, kernel events can be enabled based on the user's interest. MiSeRTrace post-processes the trace logs and hence does not add to the overhead. Upon completion of the workload, the trace logs generated are passed onto the tool.

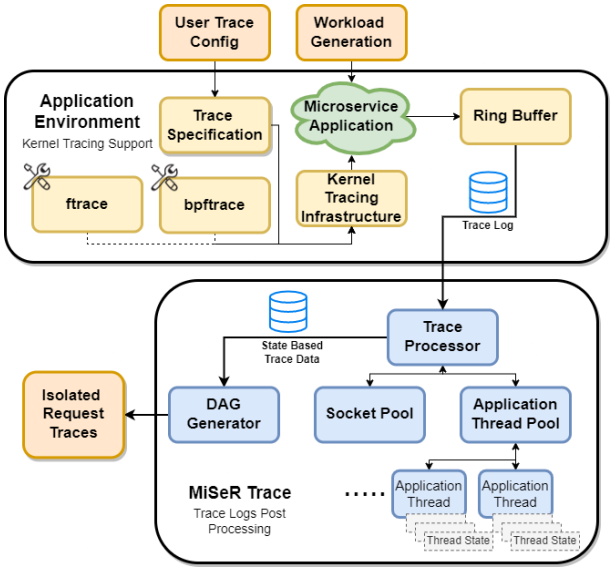

Figure 1: MiSeRTrace workflow

(3) Processing of the Trace Logs: MiSeRTrace performs sequential post processing on the kernel trace logs. Upon processing, the generated data consists of all the request spans, where spans are represented as states of the application threads as we will explain in the Thread State Model (TSM). All kernel events that were enabled by the trace specification are associated with the respective states.

(4) Generation of Isolated Request Traces: Once the trace logs have been completely processed, the Directed Acyclic Graph (DAG) Generator consolidates all the states associated with each unique client request and isolates them into time ordered traces. Each request trace is a DAG that is generated by a depth-first recursive algorithm.

MiSeRTrace is designed and implemented as the following 3 hierarchical cohesive entities.

(1) Trace Processor: The trace processor is responsible for the creation of thread pools and socket pools maintained in MiSeRTrace. The kernel trace logs are passed into the trace processor ordered by time of occurrence. Generation of trace IDs that uniquely identify the client requests is also handled here. The trace logs are validated, preprocessed, and sent to the thread processor. Once all the trace logs are processed by the lower levels, the trace processor transforms the state-based trace data into a representation suited for the DAG Generator.

(2) Thread Processor: The thread processor maintains the thread pool created by the trace processor. It tracks and creates representations of threads of the application brought about by forking of threads during workload runs. The threads are maintained in two pools, a pool of active threads and a pool of terminated threads.

(3) Thread State Processor and Thread State Model (TSM): Each thread object has its own State Processor which processes all trace records associated with that thread, updates states as per the TSM, and also propagates the trace ID through the request lifecycle. TSM is motivated by a need to associate steps within the request lifecycle to application threads. As per the model, there are two types of states depicted in Figure 2, namely Network Thread States and Fork Thread States. Each state is associated with only a single application thread.

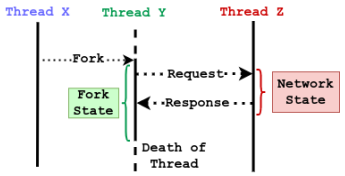

Figure 2: Types of Thread States

Network Thread States: They are created to track incoming TCP requests to a thread. A thread has one state for every incoming request. Each network state in a thread can be uniquely identified by a source thread, TCP 4 tuple, and the trace ID of the incoming request. The source thread refers to the thread which is sending data. Any communication between two threads involves the usage of a socket which is identified by the TCP 4 tuple which consists of the IP addresses and ports of both the sender and receiver. Each socket is transmitting either an application REQUEST or RESPONSE. The span of this state is from the time of request arrival to the time a response is sent for that request.

Fork Thread States: They are created when a thread forks to handle multiple incoming requests or send multiple asynchronous requests to other threads. Each state is uniquely identified by the parent thread and the trace ID of the request which triggered the parent thread to spawn the child thread. The span of this state is from the time the child thread is created to the time it terminates.

4 IMPLEMENTATION

MiSeRTrace has been implemented to allow flexibility in terms of intrusiveness by giving the user control over the amount of tracing. In order to minimize the overhead caused by tracing, only the PIDs that belong to the application are traced. This is done by examination of the PIDs present on the containerization engine's network and passing on the same to the tracing backends (bpftrace/ftrace). The size of the in-memory ring buffer is to be set depending on how much kernel activity is being captured to prevent loss of data in the trace logs.

The basis of tracing request flows through the application is by monitoring a set of system calls activated when sending and receiving messages (SendSyscallsSet and RecieveSyscallsSet in Algorithm 1), and kernel events that occur within these system calls. These are available as static tracepoints in the tracing backends. In addition to these, a custom kprobe required by MiSeRTrace is inserted into the kernel. This probe is associated with the sock_sendmsg and __sys_sendmsg functions and is used to detect data being sent over a TCP connection after necessary permissions are acquired. Custom probes such as these can also be added by the user to gain a view of kernel activity at maximum configurability. The tracing backends have respective provisions to add and activate such probes. To capture the reception of data over a TCP connection, a static tracepoint called tcp_rcv_space_adjust is used, which is triggered every time data is copied to the user space at the receiver's end. A total of 21 events are utilized by MiSeRTrace. The information provided by these events is used to update the thread states of the TSM in order to trace the request flow. Algorithm 1 is a simplified version of the process of creating and updating the Network Thread States for any given thread by monitoring the networking events. A similar algorithm also manages the Fork Thread States by utilizing the scheduling events - sched_process_fork and sched_process_exit.

Algorithm 1: TSM algorithm to handle network transfers

```
ThreadPool ← Map[Pid→Thread]
SocketPool ← Map[(IpPair, PortPair)→socket]
SendSyscallsSet ← Set{sendto, sendmsg, write, writev}
RecieveSyscallsSet ← Set{recvfrom, recvmsg, read, readv}
NetworkStateStore ← Map[(SrcThread, tcp4tuple, TraceId)
  → NetworkState] // one store per thread
if currentSyscall ∈ SendSyscallsSet then
    if currentTracepoint.event = tcpSendKprobe then
        senderSock ←
          SocketPool.get(currentTracepoint.tcp4tuple)
        senderSock.senderThread ← currentThread
        senderSock.type ← REQUEST
        /*To check if a Response is being sent*/
        forall state ∈ NetworkStateStore.values() do
            /*Source and destination here refer to the pairs
              (sourceIp, sourcePort) and (destinationIp,
              destinationPort) respectively*/
            if state.source = senderSock.destination then
                senderSock.type ← RESPONSE
                state.endSpan()
                break
            end
        end
    end
else if currentSyscall ∈ RecieveSyscallsSet then
    if currentTracepoint.event = tcpRcvSpaceAdjust then
        recieverSock ←
          SocketPool.get(currentTracepoint.tcp4tuple)
        if receiverSock.type = REQUEST then
            senderThread ← receiverSock.senderThread
            /*All active threadStates of the senderThread are
              propogated to the currentThread*/
            forall traceID ∈ senderThread.allTraces do
                stateKey ← (senderThread,
                  receiverSock.tcp4tuple, traceId)
                state ← createNetworkState(stateKey)
                state.source ← receiverSock.source
                state.startSpan()
                NetworkStateStore[stateKey] ← state
            end
        end
    end
end
```

While using ftrace, the logistics of managing the trace data is handled by trace-cmd [8], an interface for ftrace. In the case where bpftrace is used as the backend, there are a few extra steps to ensure while collecting trace logs. Bpftrace does not notify when space for the ring buffer is allocated or when the trace logs have been written to disk and hence these need to be managed by the user. The logs obtained from bpftrace are per-CPU time ordered logs but are not time-ordered globally across CPUs. MiSeRTrace also encompasses a sorting system for the ingestion of such trace logs from bpftrace.

5 RESULTS

MiSeRTrace was put to use on an open-source microservice benchmark suite known as DeathStarBench [13], which consists of many end-to-end services with representative workloads. The social network application is one of the benchmarks in DeathStarBench, implemented with loosely-coupled containerized [2] microservices communicating with each other via Thrift RPCs. It primarily consists of 3 levels - the front end (load balancer, Nginx), the logic, and the backend stores (Memcached, MongoDB, Redis).

The benchmark was run on a system with two CPU sockets, where each socket is an AMD EPYC 7401 24-Core Processor. Each core has 2 logical CPUs, and hence the machine has a total of 96 CPUs. Each socket consists of 4 NUMA nodes. This non-uniform memory access configuration where local and remote memory accesses take different times was used to simulate a distributed cluster-like environment. This server runs Linux kernel 5.4.0 and has a working memory of 128 GB.

A workload was generated on the benchmark using wrk [9], a HTTP benchmarking utility. Any form of statistical/state-based analysis using kernel events can be performed by feeding stubs of code into MiSeRTrace. As an example, the trace-points sched: sched_migrate_task and exceptions:page_fault_user were enabled for monitoring with the objective of counting the number of occurrences of these events. These trace logs were then processed by MiSeRTrace which produced the request flow DAGs for all client requests, one of which is represented in Figure 3. Each segment here represents a state as defined by the Thread State Model, whose span is represented by the length of the segment. Each state in the figure shows the PID of the associated thread and the name of the microservice running on it. In Figure 3, PID 2066822 forks PID 2066823 which sends a TCP request to PID 1966384. The span of a state is concluded upon sending a response/death of the thread. One of the insights that can be derived from this is that the number of page faults that occurred in the Home-Timeline Redis span is comparatively higher. These statistics can also be compared across multiple request traces to understand the performance differentials.

6 CONCLUSION AND FUTURE WORK

We have presented MiSeRTrace, a framework for tracing requests to microservice applications at the kernel space. It is capable of monitoring an un-instrumented application with minimal overheads. The tool includes provisions for enabling all the features and events of tracing backends such as bpftrace/ftrace. Subsequently, we demonstrated the usage of MiSeRTrace to trace the end-to-end path of client requests and user enabled events on a benchmark microservice application. With the exhaustive observability brought about by this approach, kernel-based application optimizations become much more accessible. In this paper, our experiments were conducted on a large core-count server. As future work, MiSeRTrace can also be extended to support tracing of kernel activity on a cluster of machines to derive kernel insights irrespective of the scale of the application.

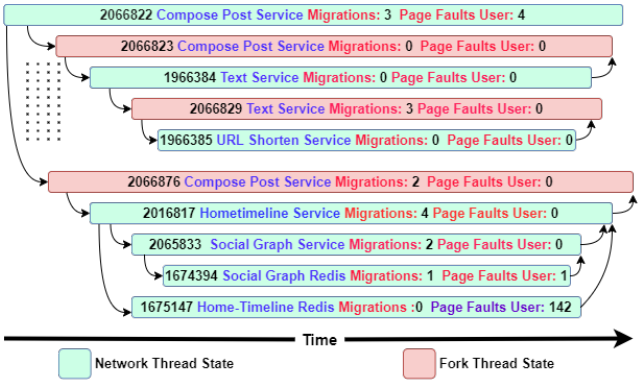

Figure 3: A representative section of states of a single client request trace along with the tally of the monitored events

REFERENCES

[1] [n. d.]. *bpftrace.* https://github.com/iovisor/bpftrace
[2] [n. d.]. *Docker.* https://www.docker.com
[3] [n. d.]. *eBPF.* https://ebpf.io
[4] [n. d.]. *ftrace.* https://www.kernel.org/doc/Documentation/trace/ftrace.txt
[5] [n. d.]. *Istio.* https://istio.io/latest/about/service-mesh/
[6] [n. d.]. *Jaeger.* https://www.jaegertracing.io
[7] [n. d.]. *OpenTracing.* https://opentracing.io
[8] [n. d.]. *trace-cmd.* https://trace-cmd.org/
[9] [n. d.]. *wrk.* https://github.com/wg/wrk
[10] [n. d.]. *Zipkin.* https://zipkin.io
[11] Sriyash Caculo, Kanishka Lahiri, and Subramaniam Kalambur. 2020. Characterizing the Scale-Up Performance of Microservices using TeaStore. In *2020 IEEE International Symposium on Workload Characterization (IISWC).* 48–59. https://doi.org/10.1109/IISWC50251.2020.00014
[12] Yu Gan and Christina Delimitrou. 2018. The Architectural Implications of Cloud Microservices. *IEEE Computer Architecture Letters* 17, 2 (2018), 155–158. https://doi.org/10.1109/LCA.2018.2839189
[13] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19).* Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013
[14] Wilhelm Hasselbring and André van Hoorn. 2020. Kieker: A monitoring framework for software engineering research. *Software Impacts* 5 (2020), 100019. https://doi.org/10.1016/j.simpa.2020.100019
[15] Benjamin H. Sigelman, Luiz André 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. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf
[16] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2016. Workload characterization for microservices. In *2016 IEEE International Symposium on Workload Characterization (IISWC).* 1–10. https://doi.org/10.1109/IISWC.2016.7581269
[17] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC).* 583–590. https://doi.org/10.1109/ColumbianCC.2015.7333476