

Zero-instrumentation observability based on eBPF

Nikolay Sivko, Co-Founder and CEO at Coroot

Observability is ...

... being able answer questions about your system:

- How is the system performing right now?
- How does its performance compare to an hour ago?
- Why are some requests failing?
- Why are certain requests taking longer than expected?



Systems a while ago



- A monolith application
- DB runs on dedicated nodes
- If something goes wrong:
 - Check the app's logs/metrics
 - Check the DB's logs/metrics
 - Check the hardware



Modern systems



- Hundreds or even thousands of services dynamically allocated to nodes
- Nodes are dynamic and can appear and disappear
- If something goes wrong:
 - Troubleshooting follows the system's topology
 - Analysis of extensive telemetry, from application latency to EBS performance

Making a system observable

- Collecting telemetry data: metrics, logs, traced, profiles
 - Time- and resource-consuming process since it requires adding instrumentation into every application
 - Hard to achieve 100% coverage without blind spots (3rd party and legacy services)
- Storing telemetry data in some databases
- Learning how to troubleshoot your system using all that data
 - The most challenging part



Collecting telemetry data

Before answer HOW to gather, let's discuss WHAT to gather or what we want to know about our apps.

- SLI (Service Level Indicators): requests, errors, latency
- Communication with other services or databases: requests, errors, latency
- Resource-related metrics: CPU, Memory, Disk
- Network-related metrics: latency, connectivity, packet loss
- Node-level metrics and logs
- Runtime-related metrics: GC, Thread Pools, Connection pools, Locks
- Orchestrator-related metrics
- Logs to identify application-specific issues
- Profiles to explain spikes in CPU or Memory usage



Collecting telemetry data

- It's possible to collect all these data without using eBPF, but eBPF allows to achieve that in MINUTES
- There are always legacy and 3rd party services that you can't instrument. eBPF doesn't require code changes and redeployments.
- Usually, developers instrument only most critical services, so you can't be sure that you have no blind spots.
- Instrumentation is a continuous process, so you need to ensure that every new service integrates OpenTelemetry SDKs.



A quick intro into eBPF

- A feature of the Linux kernel
- Allows to run small programs in the kernel-space and call them on any kernel or app function call
- Such programs have access to function arguments and returning values
- Then, they can send some data to a program in the user-space

eBPF is just a way how we can obtain data, we just need to implement kernel-space and user-space programs

How to use eBPF

- **Kprobe** allows to capture any kernel function call
 - Kernel functions can be renamed, deleted, their arguments can change
 - Some functions, in fact, almost never change
- Tracepoints statically instrumented places in the kernel which are relatively stable comparing to Kprobe
- **Uprobe** allows to capture user-space programs calls
- **MAPS** allow to store some state in the kernel space
- PERF_MAPS allows to share data between the kernel-space and userspace

It's good to know, but you don't have to write your own eBPF programs. There are a lot of ready-made tools, such as **Coroot**



Coroot-node-agent (Apache 2.0 license)

- An open-source Prometheus/OpenTelemetry compatible agent that gathers metrics, logs, traces and profiles
- Discovers containers/processes running on the node
- Discovers their logs (k8s, docker, containerd, journald) and sends them over OTLP
- Extracts repeated patterns from logs and generates log-based metrics
- Monitors TCP connections of every container
- Measures network latency between each container and its peers
- Tracks communications between services (requests, errors, latency), supports HTTP, GRPC, Postgres, MySQL, MongoDB, Redis, Memcached, Cassandra, Kafka, Rabbitmq, NATS



How the agent leverages eBPF

tracepoint/task/task_newtask: tracking new process creation. It reports only a PID, then the agent discovers container metadata using /proc

tracepoint/oom/mark_victim: marking a process as a victim of the OOM killer

tracepoint/sched/sched_process_exit: tracking process terminatio. If a process was terminated by the OOM killer, the event is enriched with the reason of the termination

tracepoint/syscalls/sys_enter_open(at): tracking file openings to identify the logs and partitions used by a specific container



How the agent leverages eBPF

tracepoint/syscalls/sys_enter_connect: tracking FD of a TCP connection

tracepoint/sock/inet_sock_set_state: tracking peers and states of TCP connections

tracepoint/tcp/tcp_retransmit_skb: tracking TCP retransmissions



How the agent leverages eBPF

tracepoint/syscalls/sys_enter_write/writev/sendmsg/sendto: track writes to an FD (socket)

tracepoint/syscalls/sys_enter_read/readv/recvmsg/recvfrom: track reads from an FD (socket)

2-phase L7-protocol parsing:

- Kernel space: high-performance protocol detection
- User-space: protocol parsing for generating metrics and traces



Capturing the data before encryption and after decryption.

- For apps using OpenSSL:
 - uprobe/SSL_read
 - uprobe/SSL_write
- For GO apps:
 - uprobe/go_crypto_tls_write
 - uprobe/go_crypto_tls_read



eBPF: performance impact

The Linux kernel ensures minimal interruption to kernel code execution by validating each eBPF program before execution:

- Program **must** have a finite complexity.
- The verifier evaluates all possible execution paths within configured upper complexity limits

Communication between kernel-space and user-space programs occurs through a ring buffer:

 If the user-space program delays data reading, it may miss data due to overwriting

For observability, it's a great deal: although we might lose some telemetry data, we can be sure that there is no impact on performance



eBPF-based metrics



eBPF-based metrics



- L7: requests per second, Errors, Latency
- Network Round-trip-time (RTT)
- TCP: connections, failed connection attempts, retransmissions (can signify packet loss)



eBPF-based traces (spans)



	Client	Status	Duration	Name	Details
E	арр	🔗 ОК	27.04ms	query	PREPARE AS select title, body from articles where id = any (\$1)
E	арр	S OK	53.30ms	query	insert into articles (created, title, body) values (\$1, \$2, \$3)
E	арр	🤣 ОК	38.06ms	query	select id from articles where created < \$1 order by created desc limit \$2
Ξ	арр	📀 ок	26.22ms	query	PREPARE AS select id from articles where created < \$1 order by created desc limit \$2

- Traces are extremely useful for identifying the particular requests within an anomaly
- They also provide a more granular distribution of requests by latency and status



eBPF-based tracing limitations

- The OpenTelemetry SDKs generate a TRACE_ID for each request and propagate it to other services
- When using eBPF, TRACE_IDs are not available, limiting us to capturing individual spans (requests) rather than complete traces
- There's a tool that claims to generate TRACE_IDs by intercepting and modifying requests, but I think it's not a good idea

Coroot supports both traditional OpenTelemetry integration and eBPF-based tracing methods



eBPF-based continuous CPU profiling

- Allows to explain any anomaly in CPU usage precise to the particular line of code
- Doesn't require any code changes
- Gathers per-process call stacks and aggregates them by containers
- Resolution by default is 60 seconds, so you can compare profiles within and anomaly with previous periods



eBPF-based CPU profiling



Q Search

													_
total (4 s, 100%)													
coredns (4 s, 100%)													
[unknown] (370 ms, 1 n	untime.goexit (3 s, 88%)												ru
[unknown] (300 r rur	github.com/coredns/cade c c	github.con	n/miekg/di	ns.(*Sen	rver)).serveUDP.func3 (2 s, 53%)			ç net/http.(ne	runtime.mcall (32	runtii	ru
runtime.mstart (run	github.com/coredns/core r ç	github.cor	m/miekg/d	ins.(*Ser	erver	r).serveUDPPacket (2 s, 53%)		ç net/http.(n	runtime.goexit0 (3	runtii	ru	
runtime.mstart0 run	github.com/miekg/dns.(*	github.co	m/miekg/	dns.(*Se	erve	er).serveDNS (2 s, 53%)		ç r net/h r	п	runtime.schedule	runtii	ru	
runtime.mstart1 run	github.com/miekg/dns.(*	github. g	ithub.com	/miekg/	/dns	s.HandlerFunc.ServeDNS (2 s, 48%)			ç r net/h r	n	runtime.findrunna	runti	n
runtime.sysmon run	github.com/miekg/c run	runtin g	ithub.com	/coredna	ns/c	oredns/core/dnsserver.(*Server).ServePacket.func1 (2 s, 48	3%)		net/h r	n	runtime.e runt		n
rur r runtime	github.com/miekg/c run	runtin	jithub.com	/coredn	Ins/c	coredns/core/dnsserver.(*Server).ServeDNS (2 s, 48%)			githu	n	runt		0
rut	github.com/miekg/ r ru	runtin	github.con	n/coredr	dns/	coredns/plugin/metrics.(*Metrics).ServeDNS (2 s, 48%)		1	ç gi	n	runt		1
rut	net.(*UDPConn). r r.	runtin	github.co	m/cored	edns	/coredns/plugin.NextOrFailure (2 s, 43%)		github.c	gi	n			
	net.(*UDPCor ne ri	rur ru	github.co	m/cored	edns	coredns/plugin/errors.(*errorHandler).ServeDNS (2 s, 43%))	gi g ç	gr	n			
	net.(*UDPCor r	ru ru	github.co	m/cored	edns	coredns/plugin.NextOrFailure (2 s, 43%)		9 5		n			
	net.(*netFD).	r	github.co	m/cored	edns	/coredns/plugin/loadbalance.(*RoundRobin).ServeDNS (2 s	s, 43%)	ç i	S	n			
	internal/poll.	r	github.co	m/cored	edns	/coredns/plugin/loadbalance.RoundRobin.ServeDNS (2 s, 4	(3%)		Ł	n			
	syscall.recv	1	github.co	om/core	edna	s/coredns/plugin.NextOrFailure (1 s, 36%)	runtime.moi		c	1			
	syscall.rec		github.c	om/core	redn	s/coredns/plugin/cache.(*Cache).ServeDNS (1 s, 36%)	runtime.new		c	r			
	syscall.Sy		gith	github.	. gi	ithub.com/coredns/coredns/plugin/loadbalance.(*RoundRo	runtime.cc		c				
	runt		gith	gith	n	github.com/coredns/coredns/plugin/metrics.(*Recorder).V	runtime.g						
	runt		gith	gith		github.com/coredns/coredns/plugin/pkg/dnstes runtime.(runtir		55				
	runt		gith	9 9		github.com/coredns/coredns/request.(*ScrubWr run rt rt	runt						
	run		9 9	ç	ç	github.com/miekg/dns.(*response).WriteMsg (8 run n	run						
	run		9 5	ç	ç	ç github.com/miekg/dns.(*response).Write (800 ru n	run						
			çç	1	1	ç github.com/miekg/dns.WriteToSessionUDP (8							



eBPF-based CPU profiling (comparison mode)



Q Search													
total													
coredns													
[unknown] (+0.5: r	runtime.goexit												rur
[unknown] (+0	github.com/coredns/	github.cor	m/miekg/dns.(*Serv	miekg/dns.(*Server).serveUDP.func3 (+6.2%)							ne runtime.mcall (-2.4	runtim	rur
runtime.msta	github.com/coredns/	github.co	om/miekg/dns.(*Ser	erver).s	mul	net/http.(*c	r ı	n runtime.goexit0 (-2	runtim	rur			
runtime.msta	github.com/miekg/dr	github.co	om/miekg/dns.(*Ser	erveDNS (+6.2%)	annt	r net/htt ru		n runtime.schedule (runtim	rur			
runtime.msta	github.com/miekg/dr	githul g	jithub.com/miekg/d	andlerFunc.ServeDNS (+4.3%)	1.172	r net/htl ne		n runtime.findrunnat	runt				
runtime.sysm	github.com/mi runti	runtii gi	ithub.com/coredns.		net/ht ne		n runtime.epi runt		r				
runt runtir	github.com/mierunti	runtii	github.com/coredn	ns/core	edns/core/dnsserver.(*Server).ServeDNS (+4.7%)				githut n		n i runt		
runt	github.com/mi i runt	runtii	github.com/coredns/coredns/plugin/metrics.(*Metrics).ServeDNS (+5.0%)						g gitt n		n runt		
runt	net.(*UDPCon run	github.com/coredns/coredns/plugin.NextOrFailure (+5.1%) github						gitt in		n			
	net.(*UDPCr n runt	ruri	github.com/cored	dns/co	oredns/plugin/errors.(*errorHandler).ServeDNS (+5.1%)		gr		gitt in	Ш	n		
	net.(*UDPC	rur	github.com/cored	github.com/coredns/coredns/plugin.NextOrFailure (+5.1%) g +							n		
	net.(*netFD)		github.com/cored	hub.com/coredns/coredns/plugin/loadbalance.(*RoundRobin).ServeDNS (+5.1%)						H	n		
	internal/pol		github.com/cored	dns/co	oredns/plugin/loadbalance.RoundRobin.ServeDNS (+5.1%		b		n				
	i syscall.rec		github.com/cored	edns/c	oredns/plugin.NextOrFailure (+1.2%)	runtime.m			c		r		
	i syscall.re		github.com/core	edns/c	coredns/plugin/cache.(*Cache).ServeDNS (+0.81%)	runtime.ne			с	5	r		
	r syscall.		github githu	ut git	hub.com/coredns/coredns/plugin/loadbalance.(*RoundR	o runtime.c			c				
	r runti		github gith	hu g	github.com/coredns/coredns/plugin/metrics.(*Recorder).	N runtime.g			c				
	r runt		github gith	h g	jithub.com/coredns/coredns/plugin/pkg/dnstest.(runtim	e runti							
	r runti		github g g	9 9	github.com/coredns/coredns/request.(*ScrubWrite ru ru	n rur							
	runt		git git g	<u>c</u>	github.com/miekg/dns.(*response).WriteMsg (+1. rt + ru	n ru							
	runt		git git g	ç	ç github.com/miekg/dns.(*response).Write (+0.87 r ru	n ru							

How Coroot works



- coroot-node-agent: gathers metrics, logs, traces, and profiles. Installed on every node in the cluster (k8s, docker, VM, bare-metal)
- Prometheus for storing metrics
- ClickHouse for storing logs, traces, and profiles
- Coroot: UI, alerts, predefined inspections
- You can use Coroot as an OpenTelemetry backend for logs and traces



Conclusion

- eBPF is awesome!
- It enables gathering almost any telemetry data you need without requiring code changes.
- The performance impact on your apps is negligible.
- Want to gain system visibility in minutes? Install Coroot (Open Source, Apache 2.0).

https://github.com/coroot/coroot



Thank you, Let's connect!

https://www.linkedin.com/in/nikolay-sivko https://twitter.com/NikolaySivko https://www.coroot.com https://github.com/coroot/coroot

