Beyla performance calculation

This document is shared publicly by Grafana Labs

This document gathers all the information regarding performance implications of running Beyla in production where a diverse set of applications are instrumented.

In the first iteration of this we are going to focus on Beyla as single process (either in Kubernetes or not), therefore we are going to get out of the scope the following:

- Overhead caused by running Beyla in a Pod.
- Run Beyla as DaemonSet and have k8s cache functionality activated.
- Be exhaustive with all possible configurations of options and combinations of them.

In the present document we are running different experiments with different options activated. If one other option doesn't impact performance, it won't be documented publicly. We are using the following approach to measure the performance.

- Deploy Beyla in a local kind cluster using Helm (version 1.9)
- Deploy open telemetry demo with load generator activated
 - This script generates traffic, between 20 and 60 requests/s, being the /api/products the endpoint with more traffic. Since there are also requests to redis, kafka and internal RPC, the total amount of requests can be calculated like this.

73 reg/s 72.5 reg/s 72 reg/s 71.5 reg/s 71 reg/s 70.5 reg/s 70 reg/s 60.5 reg/s		
69 req/s 17:00 18: — sum(rate(http_client_r	00 19:00 20:00 21:00 22:00 03:00 01:00 02:00 03:00 04:00 05:00 06:00 07:00 08:00 09:00 10:00 11:00 12:00 13:00 equest_duration_seconds_count[24h])) + sum(rate(htp_server_request_duration_seconds_count[24h])) + sum(rate(rpc_server_duration_seconds_count[24h])) + sum(rate(rpc_server_duration_seconds_count[24h]))	14:00 15:00 16:00 s_count[24h])) + sum(rate(me
🛢 Queries 🚹 🖁		
Data source 🤑 grafa	nacloud-noteldemo-proi v Overy options MD = auto = 1496 Interval = 1m	Query inspector
✓ A (grafanaclo)	ud-nateldemo-prom)	0 ₺ ⊚ 前 #
Kick start your query	Explain • Run qur	
Metrics browser >	<pre>sum(rate(http_client_request_duration_seconds_count[24h])) + sum(rate(http_server_request_duration_seconds_count[24h])) + sum(rate(rpc_client_duration_seconds_count[24h])) + sum(rate(rpc_server_duration_seconds_count[24h])) + sum(rate(messaging_process_duration_seconds_count[24h])) + sum(rate(messaging_publish_duration_seconds_count[24h])) + sum(rate(db_client_operation_duration_seconds_count[24h]))</pre>	

- Measure performance with *application_process* and visualize data in Grafana.
 - Potentially measure the performance in a different way to measure overhead of *application_process* option

These are the different scenarios considered

Beyla self instrumented



features:

- application
- application_process



Summary:

Beyla uses around 75mb and 0.02% of CPU when only self-instrumented. Enabling application and application_process has a minimal overhead on the memory.

Instrument Go applications (1, many, with and without load)

1 process

config:
data:
attributes:
kubernetes:
cluster_name: kind
discovery:
allow_self_instrumentation: true
services:
- k8s_namespace: default
k8s_daemonset_name: beyla
- k8s_namespace: default
k8s_deployment_name: notel-demo-checkoutservice
prometheus_export:
port: 9090
path: /metrics
features:
- application
- application_process



In this case the memory didn't change, so we have memory around 78MB and CPU around 0.05%.

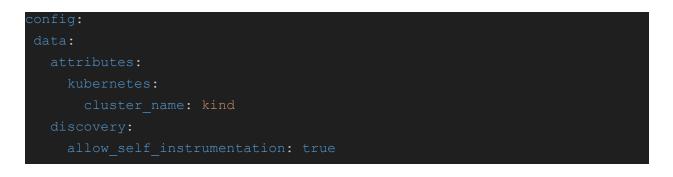
with traffic



Summary:

The memory stays constant but now Beyla has to process the requests, so the CPU it's around 0.1 %, which is more or less 10x than before.

many processes



- application_process



Summary:

When we instrument many processes at the same time, the memory increases dramatically (up to 130MB in this case), but then it returns to its normal state. The CPU also stays constant.

with traffic



The memory fluctuates, probably due some caching or metrics expiration. In this case the Go applications are receiving traffic, therefore the CPU is more busy again, reaching levels around 0.2%

Instrument non-Go applications (1, many, with and without load)

1	process
---	---------

config:
data:
attributes:
kubernetes:
cluster_name: kind
discovery:
allow_self_instrumentation: true
services:
- k8s_namespace: default
k8s_daemonset_name: beyla
- k8s_namespace: default
k8s_deployment_name: notel-demo-adservice
prometheus_export:
port: 9090
path: /metrics
features:
- application
- application_process



In this case, the values are pretty much the same as the case of Go.

with traffic



Summary:

In this case, the values are pretty much the same as the case of Go.

many process

config:
data:
attributes:
kubernetes:
cluster_name: kind
discovery:
allow_self_instrumentation: true
services:
- k8s_namespace: default
k8s_daemonset_name: beyla
- k8s_namespace: default
k8s_deployment_name:
notel-demo-adservice notel-demo-cartservice notel-demo-recommendationservi
ce
prometheus_export:
port: 9090
path: /metrics
features:
- application
- application_process



In this case, the values are pretty much the same as the case of Go.

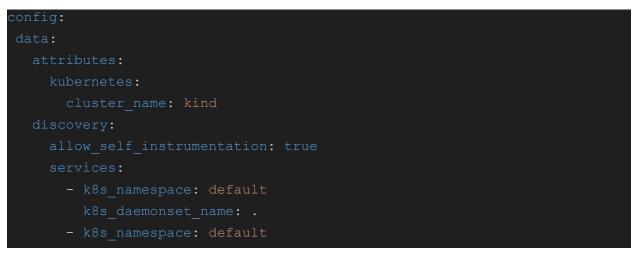




Summary:

In this case, the values are pretty much the same as the case of Go. The memory doesn't fluctuate like in the Go example.

Full OTEL demo with traffic





500 MII 400 MII		
300 Mil		
100 Mil 104200 154300 1544500 154500 154500 1547500		
- {name_*process_memory_usage_bytes*, host_name**beyla-ztkzb*, instance**10.244.0.53:9090*, job=*kube	ernetes 15:42:00 15:42:30 15:43:00 15:43:30 15:44:00 15:44:30 15:45:00 15:45:30 15:46:00 15:46:30 15:47:00 15:47:3	30
Raw Table		ble Raw
Expand results Result s process_memory_usage_bytes{host_name=*beyla-ztkzb", instance=*10.244.0.53:9090 75600 75600	Expand results 🕘 Resu	sult series: 1
b process_memory_usage_bytes{nost_name="beyia-ztkzb", instance="10.244.0.53.9090 /5000		.5498187

Instrumenting the whole OTEL demo causes a peak of 600mb initially, but then goes down to its normal levels of 75mb. The CPU increased to 0.5% utilization.

Different export modes (Prometheus, OTEL metrics, OTEL traces)

Prometheus



features:

- application
- application_service_graph
- application_span
- application process

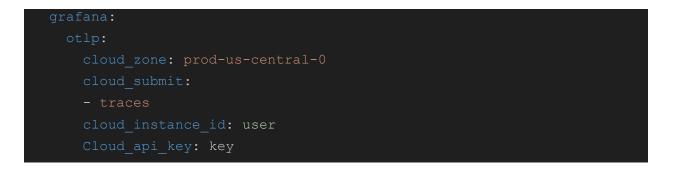


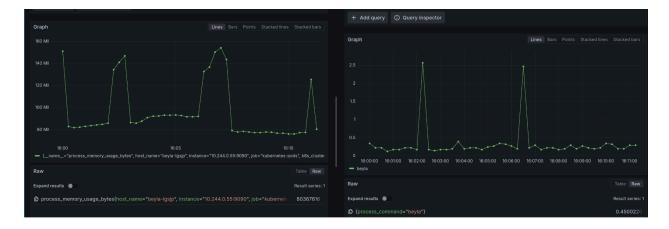
Summary:

Using application_span and application_service_graph causes an increase of memory of 10MB (87MB in total). The utilization increased from 0.5% to 1.2 %

OTEL traces

config:
data:
attributes:
kubernetes:
cluster_name: kind
discovery:
allow_self_instrumentation: true
services:
- k8s_namespace: default
k8s_daemonset_name: .
- k8s_namespace: default
k8s_deployment_name: .
prometheus_export:
port: 9090
path: /metrics
features:
- application
- application_process





Using traces is more costly for the memory, as it needs to create batches (that could explain the peaks of memory). However, it is more efficient in terms of memory as we don't need to create so many metrics.

OTEL metrics

```
- application
- application_process
- application_service_graph
- application_span
grafana:
otlp:
cloud_zone: prod-us-central-0
cloud_submit:
- metrics
cloud_instance_id: user
cloud_api_key: key
```



Using application_span and application_service_graph, with a combination of OTEL metrics causes an increase of memory of 50MB (107MB in total). The utilization increased from 0.5% to 1.6 %

Debug mode



```
discovery:
  allow_self_instrumentation: true
  services:
    - k8s_namespace: default
    k8s_daemonset_name: .
    - k8s_namespace: default
    k8s_deployment_name: .
    prometheus_export:
    port: 9090
    path: /metrics
    features:
        - application
        - application_process
```



Enabling debug mode causes an increase of 20/30 MB and CPU utilization around 2%.

Network observability



- k8s_namespace: d	default	
k8s_deployment_n	name: .	
prometheus_export:		
port: 9090		
path: /metrics		
features:		
- application		
- application proc	cess	
—		
- network		
- network		
	THERE Are from monotone monotone	
	Lines Bars Points Stacked lines Stacked bars Graph	Little Bars Points Started Inter Started Lates
Graph		Elected lines Stacked lines Stacked lines
Graph 700 Mi		Lines Bars Powers Stacked lines Stacked lines
Craph 700 Mil 600 Mil 500 Mil		Lines Dars Ponts Stacked bars
Graph 200 Mil 600 Mil 500 Mil		Lines Bar Ports Stacked lines Stacked laws

Sum	ma	r\/	
Sum	IIIa	ту	

Instrumenting the whole OTEL demo causes a peak of 600mb initially, but then goes down to to 120mb, which is 70mb more than without the option enabled. The CPU increased to 1.2% (before was 0.5%).

BPF probes latency overhead

The observed latency of all combined probes of Beyla running for 24h in OpenTelemetry demo is around 40ms.



This represents 500ns of latency per request:

