Monitoring infrastructure with Prometheus

Łukasz Szczęsny
luk@wyb.cz
Ustroń, 2019-11-29

$ whoami

- SRE at Chronosphere
- FOSS and Open Hardware lover
- Automation enthusiast
- Co-organizer of the WarLUG (http://warszawa.linux.org.pl)

Monitoring

Monitoring - what?

- Datacenter environment
- Network devices
- Servers
- Operating systems
- Applications (databases, frontends, backends, etc.)
- Security!

Monitoring - why?

- Alerting
- Debugging
- Trending
- Plumbing

Monitoring - how?

- Metrics
- Tracing
- Logging
- Profiling

Monitoring - how?

- **Metrics**
- Tracing
- Logging
- Profiling

Prometheus

What is Prometheus?

- An OSS monitoring and alerting toolkit written in Go
- Inspired by Google's Borgmon monitoring system
- Started at SoundCloud in 2012
- Joined Cloud Native Computing Foundation in 2016
- Graphs and dashboards included

When to use Prometheus?

- It works well for recording any numeric time series
- It’s designed for reliability - each Prometheus server is standalone system, there are almost no external dependencies, extremely easy setup
- It’s suited well for monitoring infrastructure and distributed systems
- But what about Nagios/check_mk/Sensu/Graphite/younameit!?

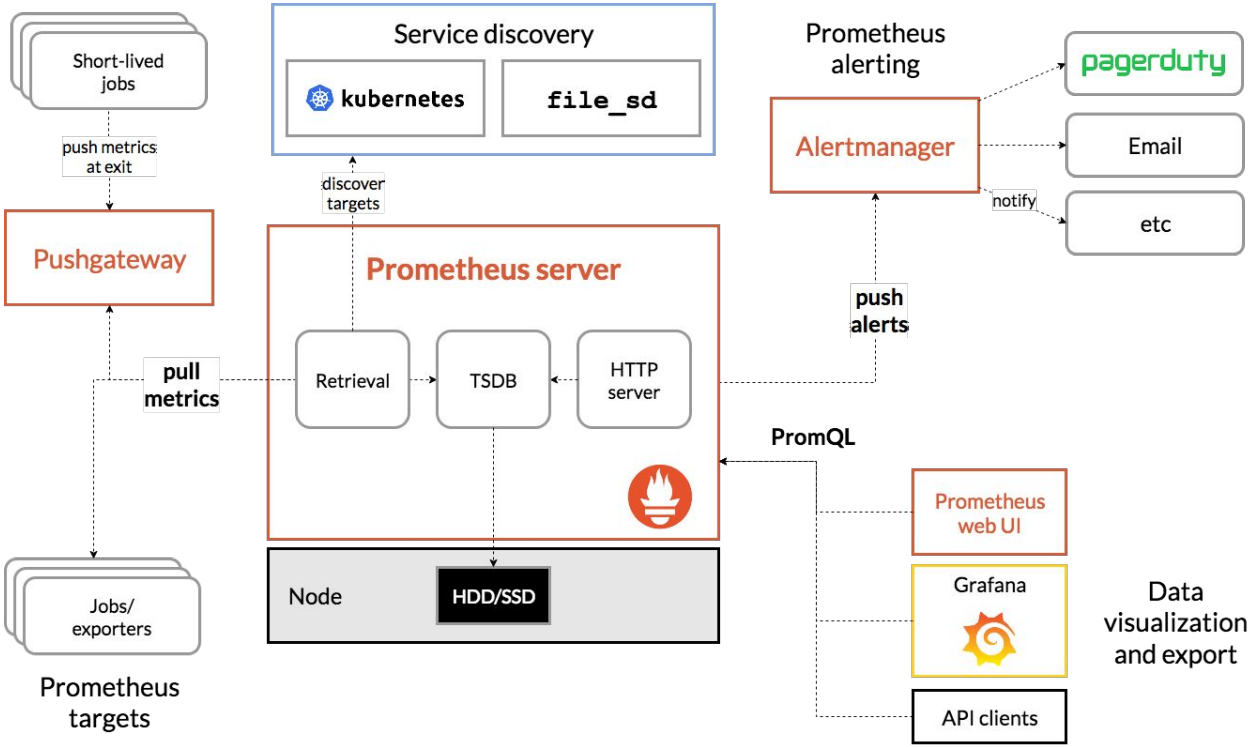
When not to use Prometheus?

- When 100% accuracy is required, such as for per-request billing
- When you need a long-term archival system
- When you need downsampling

Key concepts

- Multidimensional data model - time series data identified by metric name and key/value pairs
- Flexible query language - PromQL
- Pulling time series via HTTP
- Pushing time series is supported via an intermediary gateway

Architecture overview



Source: https://prometheus.io/docs/introduction/overview/#architecture

Exporters

- Software exposing existing metrics from 3rd party systems
- Official exporters
  - node_exporter
  - jmx_exporter
  - blackbox_exporter
  - mysqld_exporter
  - snmp_exporter
  - ...
- Multiple 3rd party exporters

Exporters

Datacenter environment / hardware	snmp_exporter, apcupsd_exporter
Network devices	snmp_exporter
Servers	ipmi_exporter
Operating systems	node_exporter, ebpf_exporter
Applications	mysqld_exporter, jmx_exporter, direct instrumentation ftw!
Security	osquery_exporter

Targets

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'alertmanager'
    static_configs:
      - targets: ['localhost:9093']

  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']
```

Prometheus

Targets

alertmanager (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9093/metrics	UP	instance="localhost:9093"	424ms ago	

node (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9100/metrics	UP	instance="localhost:9100"	2.562s ago	

prometheus (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	3.72s ago	

Jobs and instance

- ***instance*** - a target which is being scraped
- ***job*** - collection of instance with the same purpose

```
job: node
  -   instance 1: 1.2.3.4:5670
  -   instance 2: 1.2.3.4:5671
  -   instance 3: 5.6.7.8:5670
  -   instance 4: 5.6.7.8:5671
```

- Service discovery for targets

Service discovery

- Automagically retrieve scrape targets
- A must in dynamic environments where instances are spawned and killed all the time
- Generic `file_sd_config` can be used e.g. to add targets from Ansible inventory

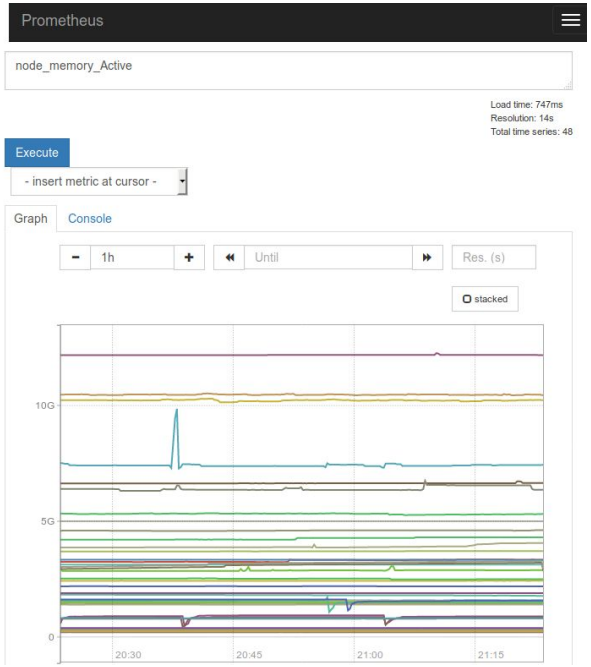
Direct instrumentation

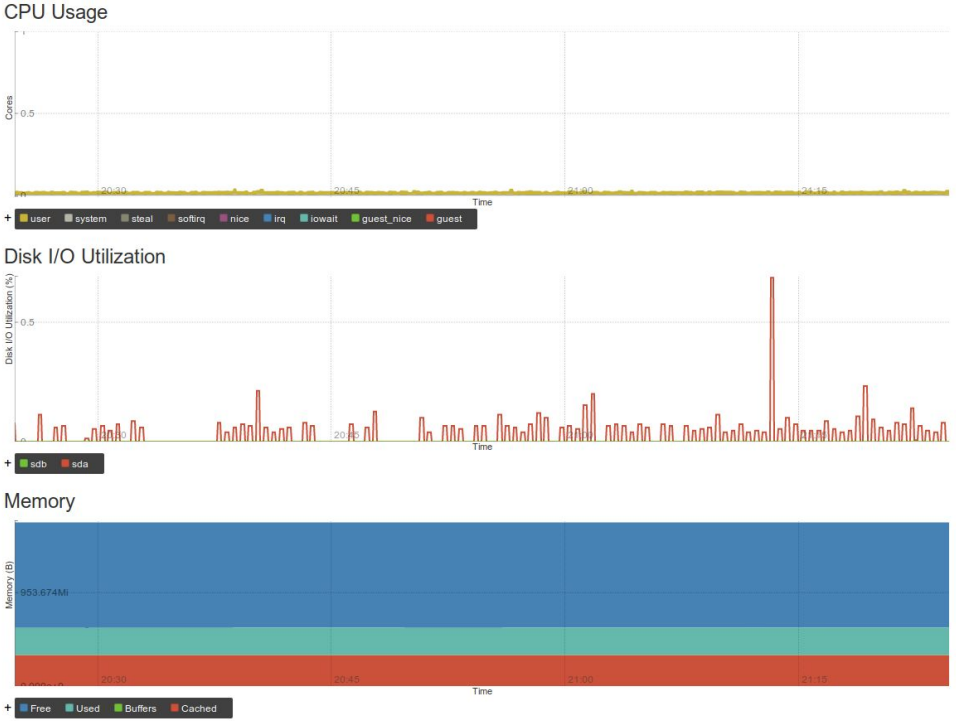
- Official libraries
  - Go
  - Java or Scala
  - Python
  - Ruby
- Multiple 3rd party libraries

Graphs and dashboards

- Built-in expression browser
- Console templates
- Grafana

Graphs and dashboards





Data model

- Counter
- Gauge
- Histogram
- Summary

Data model

- float64 values
- millisecond-precision timestamp

Data model

```
<metric name>{<label name>=<label value>, ...}
```

Data model

```
<metric name>{<label name>=<label value>, ...}
```

Data model

```
<metric name>{<label name>=<label value>, ...}
```

Data model

```
node_disk_reads_merged{device="sda"}
```

Data model

```
node_disk_reads_merged{device="sda"}
```

Data model

```
node_disk_reads_merged{device="sda"}
```

Data model

```
node_disk_reads_merged{device="sda", instance="server-01"}
```

Data model

```
node_disk_reads_merged{device="sda", instance!="server-01"}
```

Data model

```
node_disk_reads_merged{device="sda", instance=~"server-.*"}
```

Data model

```
node_disk_reads_merged{device="sda", instance!~"server-.*"}
```

Data model

```
node_disk_reads_merged{device="sda"}[5m]
```

Metrics and labels naming

- Metrics should have a suffix describing the unit in plural form, e.g. node_memory_usage***_bytes***
- Single unit - do not mix bytes with seconds!
- Preferably use base units - seconds, bytes, grams etc.
- Accumulating count has ***total*** as a suffix, e.g. https_requests***_total***
- Labels names should be chosen to differentiate the characteristics of the thing which is being measured, e.g. http_requests_total differentiate the request method, e.g. ***method=”GET|POST|PUT|...”***

Super short introduction to PromQL

Data types

- **Instant vector** - a set of time series containing a single sample for each time series, all sharing the same timestamp
- **Range vector** - a set of time series containing a range of data points over time for each time series
- **Scalar** - a simple numeric floating point value
- **String** - a simple string value; currently unused

Source: https://prometheus.io/docs/prometheus/latest/querying/basics/#expression-language-data-types

PromQL - operators

+	==	and
-	!=	or
*	>	unless
/	<	
%	>=	
^	<=	

PromQL - grouping

without () / by ()

PromQL - aggregation operators

sum
min
max
avg
stddev
without / by

stdvar
count
count_values
bottomk
topk
quantile

PromQL - matching

ignoring () / on ()

group_left () / group_right ()

PromQL - functions

```
abs()
absent()
ceil()
changes()
clamp_max()
clamp_min()
day_of_month()
day_of_week()
days_in_month()
delta()
deriv()
exp()
floor()
histogram_quantile()
holt_winters()
hour()
idelta()
increase()
irate()
label_join()
label_replace()
ln()
log2()
log10()
minute()
month()
predict_linear()
rate()
resets()
round()
scalar()
sort()
sort_desc()
sqrt()
time()
timestamp()
vector()
year()
<aggr>_over_time()
```

PromQL samples

```
node_network_receive_bytes_total{instance="nagios-01"}
```

PromQL samples

```
node_network_receive_bytes_total{instance="nagios-01"}[5m]
```

PromQL samples

```
rate(node_network_receive_bytes_total{instance="nagios-01"}[5m])
```

PromQL samples

```
sum(rate(node_network_receive_bytes_total[5m])) by (instance)
```

PromQL samples

```
topk(3, sum(rate(node_network_receive_bytes_total[5m])) by (instance))
```

PromQL samples

```
count(node_uname_info) by (release)
```

PromQL samples

```
sum by(kubernetes_pod_name)
(container_memory_usage_bytes{kubernetes_namespace="kube-system"})
```

Source: https://www.digitalocean.com/community/tutorials/how-to-query-prometheus-on-ubuntu-14-04-part-2

PromQL samples

```
sum(rate(http_requests_total{service="foo", code=~"5.."}[2m])) /
  sum(rate(http_requests_total{service="foo"}[2m])) * 100 > 0
```

PromQL samples

```
predict_linear(node_filesystem_free{job="node"}[1h], 4 * 3600) < 0
```

Rules

- Recording rules
  - precomputing frequently needed or computationally expensive expressions
  - save their result as a new set of time series
- Alerting rules
  - defining alert conditions based on PromQL expressions

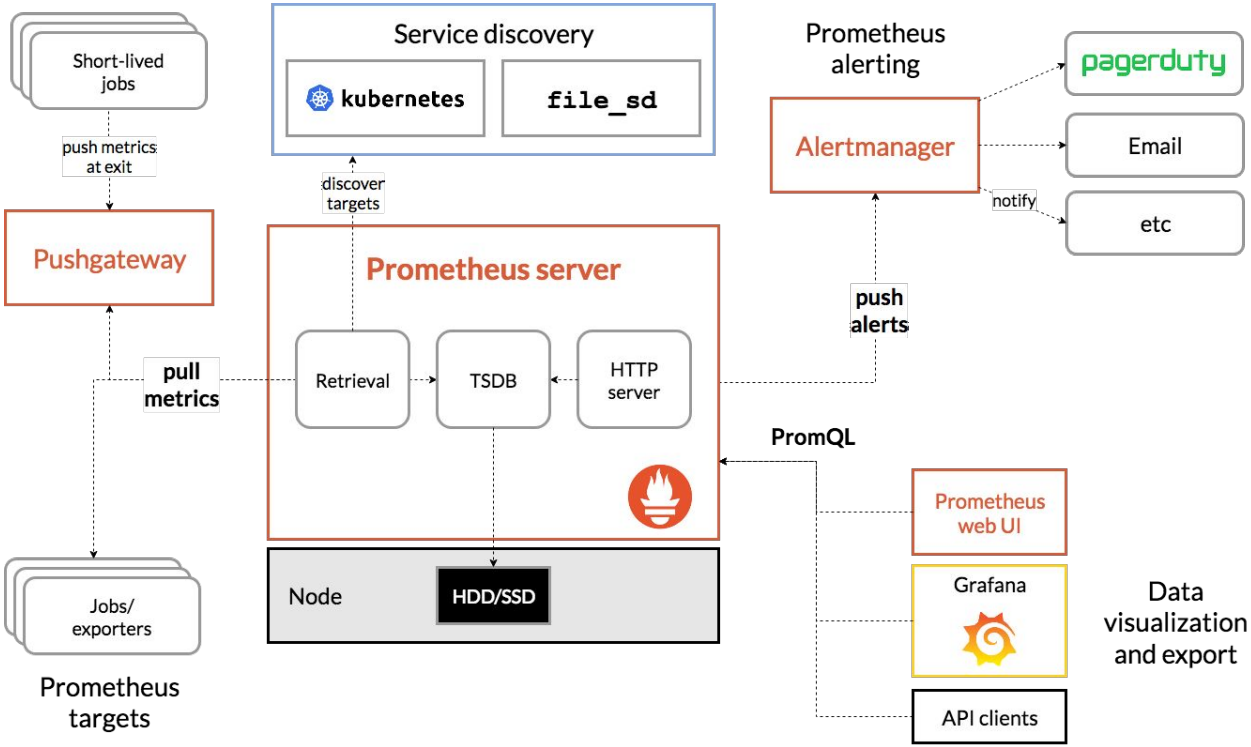
Rules

```
groups:
  - name: example
    rules:
    - record: job:http_inprogress_requests:sum
      expr: sum(http_inprogress_requests) by (job)

    - alert: HighErrorRate
      expr: job:request_latency_seconds:mean5m{job="myjob"} > 0.5
      for: 10m
      labels:
        severity: page
      annotations:
        summary: High request latency
```

Alerting

Architecture overview



Source: https://prometheus.io/docs/introduction/overview/#architecture

Alertmanager

- Receives alerts from Prometheus
- Takes care of deduplicating, grouping, and routing them
- Alerts inhibition
- Silencing alerts

Alerting rules

```
groups:
 - name: example
    rules:
    - alert: InstanceDown
      expr: up == 0
      for: 5m
      labels:
        severity: page
      annotations:
        summary: "Instance {{ $labels.instance }} down"
        description: "{{ $labels.instance }} of job {{ $labels.job }} has been down for more than 5 minutes."
        dashboard: "https://grafana.example.com/dashboard/db/job?var-job={{ $labels.instance }}"
        runbook: "https://wiki.example.com/runbook/IntanceDown"
```

Alerting rules

```
groups:
  - name: example
      rules:
      - alert: InstanceDown
        expr: up == 0
        for: 5m
        labels:
          severity: page
        annotations:
          summary: "Instance {{ $labels.instance }} down"
          description: "{{ $labels.instance }} of job {{ $labels.job }} has been down for more than 5 minutes."
          dashboard: "https://grafana.example.com/dashboard/db/job?var-job={{ $labels.instance }}"
          runbook: "https://wiki.example.com/runbook/IntanceDown"
```

Alertmanager - grouping

```
route:
  # The labels by which incoming alerts are grouped together. For example,
  # multiple alerts coming in for cluster=A and alertname=LatencyHigh would
  # be batched into a single group.
  group_by: ['alertname', 'cluster', 'service']
```

Alertmanager - routing #1

```
route:
  # A default receiver
  receiver: team-X-mails
  # The child route trees.
  routes:
  # This routes performs a regular expression match on alert labels to
  # catch alerts that are related to a list of services.
  - match_re:
      service: ^(foo1|foo2|baz)$
    receiver: team-X-mails
    # The service has a sub-route for critical alerts, any alerts
    # that do not match, i.e. severity != critical, fall-back to the
    # parent node and are sent to 'team-X-mails'
    routes:
      - match:
          severity: critical
        receiver: team-X-pager
```

Alertmanager - routing #2

```
receivers:
- name: 'team-X-mails'
  email_configs:
  - to: 'team-X+alerts@example.org'

- name: 'team-X-pager'
  email_configs:
  - to: 'team-X+alerts-critical@example.org'
  pagerduty_configs:
  - service_key: <team-X-key>
```

Alertmanager - inhibit rules

```
inhibit_rules:
- source_match:
    severity: 'critical'
  target_match:
    severity: 'warning'
  # Apply inhibition if the alertname is the same.
  equal: ['alertname', 'cluster', 'service']
```

Operating Prometheus

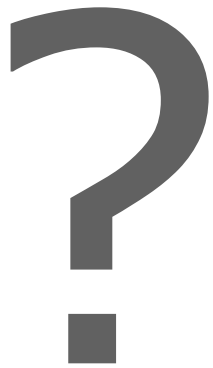
- High Availability
  - Alertmanager mesh configuration HA
  - For Prometheus itself just run two instances with the same configuration! Alertmanager will de-duplicate alerts
- Scalability
  - Divide workload by jobs
  - Instance per AZ, DC, etc.
  - Federation
- Capacity planning

Meta-monitoring

- Create an always firing alert (*expr: vector(1)*) in Prometheus
- Get an account on cron monitoring service
- Create a webhook in e.g. PD or Alertmanger to call cron monitoring service API - this alert should not page people!
- Voilà!

DEMO

Questions



Thank you!

- @wybczu
- luk@wyb.cz
- IRC: wybczu@freenode
- https://keybase.io/wybczu
- GPG key ID: 0x2EB8C2A248AD1541
  Key fingerprint = 5BEA 0D72 A27F ABB6 2F23 FF2A 2EB8 C2A2 48AD 1541

Resources

1. https://prometheus.io/docs/introduction/overview/
2. https://github.com/prometheus
3. https://promcon.io/
4. https://www.robustperception.io/blog/
5. https://groups.google.com/forum/#!forum/prometheus-users
6. Long-term storage:
    a. https://www.m3db.io/
    b. https://github.com/improbable-eng/thanos