Introduction to Prometheus

Maziyar Nazari
University of Colorado at Boulder
DevOps in the Cloud Computing Course

Abstract

Prometheus is one of the opensource monitoring solutions which started at SoundCloud around 2012-2013 and was made public in 2015. It is inspired by Google Borgmon, which uses time series data as a data source, to then send alerts based on this data. It fits very well in cloud native infrastructure. Prometheus is also a member of Cloud Native Computing Foundation (also called CNCF) after Kubernetes project. Prometheus, in general, has a built-in time-series database in which it stores time series data and it has some additional components by which it can make decisions based on processing on that data, like a query language by which we can extract our intended data. One of the main differences between Prometheus and other monitoring solutions is that it works based on pulling metrics from monitored systems (although it could have pushing metrics feature). It has some valuable features like Alerting, Visualization, Service Discovery, Pushing Gateway, Remote Storage and Exporters which can be added and benefit our cloud environment.

Architecture & Components

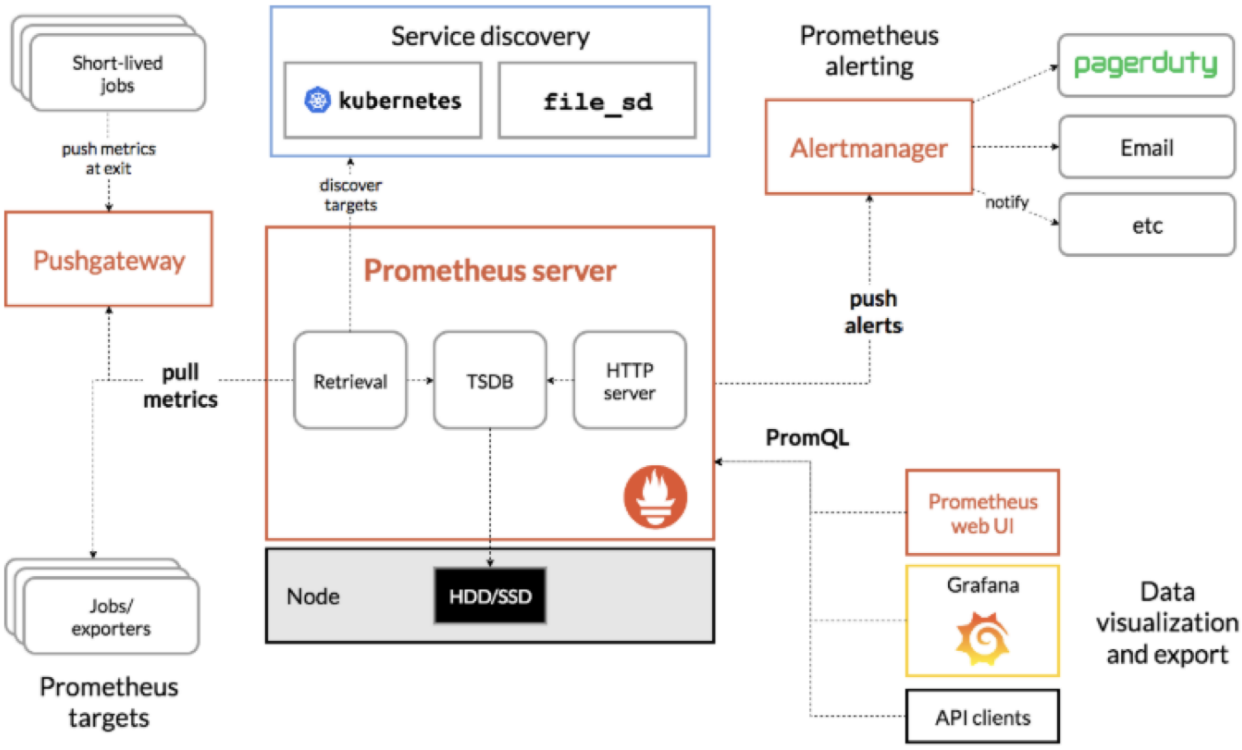
As we discussed earlier, , Prometheus stores time series data. It stores metrics in memory and local disk in its own custom, efficient format. It is written in Go. However, there are a lot of client libraries by which you can make use of Prometheus. Prometheus collects metrics from monitored targets by scraping metrics HTTP endpoint. This is fundamentally different than other monitoring and alerting systems. Rather than a custom script that checks on particular services and systems, the monitoring data itself is used. Scraping endpoints is much more efficient than a 3d party agent. A single Prometheus server is able to ingest up to one million samples per second and several million time series.

One of the basic concepts is Metrics in Prometheus. It stores time series data with a “metric name” and a set of key/value pairs called labels. The time series data also consists of actual data called “samples”. It can be a float64 value or millisecond precision timestamp.
Ex:

Graph | Console

Element	Value
go_memstats_alloc_bytes{instance="localhost:9090",job="prometheus"}	30950392
go_memstats_alloc_bytes{instance="localhost:9100",job="node_exporter"}	2251192

As it is indicated in the architecture, it has several components about which I will briefly talk and then I will summarize what I did in the Demo.



Prometheus Server:

It consists of a Time Series Database which can store and retrieve time series data and an HTTP Server which we can communicate with Prometheus server and its time series database using REST API. The retrieval part is for saving the pulled metrics into TSDB. Regarding the storage I should say that it could be a remote storage rather than HDD/SSD but we should have our own adapter (custom implementation or using already implemented) between Prometheus server and the remote storage (such as PostgreSQL).

Data Visualization:

When we clone or install the Prometheus project it will come with a built-in in-browser web app in which we can query the built-in database and extract our intended data and have a graph out of it. Furthermore, we can integrate some visualization tools like “Graphana” with Prometheus to access some additional more interesting features regarding visualization of data. However, we should clone or install other visualization tools to make use of them and they do not come with Prometheus itself.

Alert Manager:

We can integrate the Alert Manager component with Prometheus if we want. It enables Prometheus to send alerts if something strange happens. In summary, if we want that Prometheus alerts us, at the first step we should define our rules (for example: our CPU load in a specific machine must not be above 90%) in files with .rules suffix. Next step is that we should define receivers in alert manager in .yml format. If we want to use our email as an alert receiver we should specify a mail server which we can use to send us alerts. The other common alert receiver is Slack. We can get alerts in Slack notifications. By having these we should be able to receive alerts under specified condition which we stated in rules files.

Client Libraries:

We can define our own metrics in our application or system (such as a web app) and let Prometheus to scrape the HTTP endpoint which we defined in our application or system and pull metrics. Although Prometheus is written in Golang, there are a lot of libraries implemented in various programming languages which we can use to define metrics in our applications and systems and let Prometheus be aware of them. There are some official libraries which is supported by Prometheus community including: Python, Go, Ruby and Java/Scala, but there are a lot of unofficial libraries implemented in C++, Lua and etc. We can also have our own client library if they were not already implemented or other personal reasons. The only thing is that we should care about the standard which Prometheus documentation provides. For example, there are 4 types of Metrics and Prometheus allows us to provide data in those types:

- Counter: is being used to increase a numeric value
- Gauge: is being used to increase and decrease a numeric value
- Histogram: is being used to deal with something like request latency or temperature
- Summary: pretty much like Histogram but it has some additional features

And if we want to have our own custom library we should define Counter, Gauge and one of the Histogram or Summary in that. If someone wants to have his/her own library he/she must apply all of such rules in the library.

Pushing Gateway:

Sometimes metrics cannot be scraped. Examples are: batch jobs or servers which are behind the NAT. Thus, Push Gateway plays role as an intermediary between the Prometheus server and the service and instead of pulling metrics by Prometheus we will have pushing metrics into TSDB. However, there are some pitfalls regarding using this component. For instance, usually it is a single instance which results in single point of failure or after job is done we should not forget to delete the metric (by using curl -X DELETE … request to push gateway).

Querying:

Prometheus has a functional expression language called PromQL. PromQL is read-only and some functionalities like insertion cannot be done by that. It uses Instant Vector and Range Vector. Instant Vector is set of time series containing a single sample for each time series. Range Vector is set of time series containing a range of data points over time for each time series. It has functions and operators. It has arithmetic operators like: multiply and division, comparison binary operators like: < , =, Logical/set binary operators like Union, and aggregate operators like: sum, topk, etc.

Service Discovery:

We can integrate Service Discovery with Prometheus. It provides support for major cloud providers like AWS and Google Cloud. It also provide support for cluster managers like kubernetes and Mesos. It also provides support for generic mechanisms like DNS and Zookeeper.

Exporters:

There are some exporters which have been built that exporting Prometheus metrics from existing 3d party metrics. For example, if you want to pull some metrics about a Linux system like “cpu idle time” which is available in proc file system in Linux machine, you should use an already implemented exporter called “Node Exporter” which can gather those metrics from a Linux machine and provides those metrics in Prometheus language, so that Prometheus can scrape those data. There are lots of exporters implemented like MySQL exporter, Memcached Exporter, Consul Exporter, MongoDB Exporter and etc. These exporters can scrape metrics from those platforms.

Demo Summary:

I implemented a Flask application and tried to monitor that using Prometheus. The Flask application is a simple web app which could provide a REST API to which we can send requests. And also it connects to a MySQL DB and does a simple query. The application was built on docker containers and has been run using docker compose. I used /metrics path to provide data to Prometheus. I defined my own custom metrics. The metrics are:

- A Counter for the number of requests
- A Histogram for request latencies
- A Counter for the number of requests to DB
- A Histogram for DB request latencies

When a request to the application comes in, I count up the value of the Counter for the number of requests to the web app and also keep the time difference between the time that request comes in and the time that the response returns in the Histogram. I also do the same for each request to the DB for with respect to the Counter and Histogram. Then I indicated that we can simply collects these useful data from a web app in Prometheus and have a nice graph out of extracted data and monitor the web app.