REVISION HISTORY

Version	Date	Change description	Created by	Updated by	Reviewed by
V 1.0	November, 2022	Release 6.0	Hemanth/Yash	Raksha	Chiranjib

Table of Contents

Prometheus Role in CAN Monitoring

Prometheus is an open-source monitoring solution for collecting and aggregating metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels. Every time series is uniquely identified by its "*metric name*" and "*labels*".

Installation

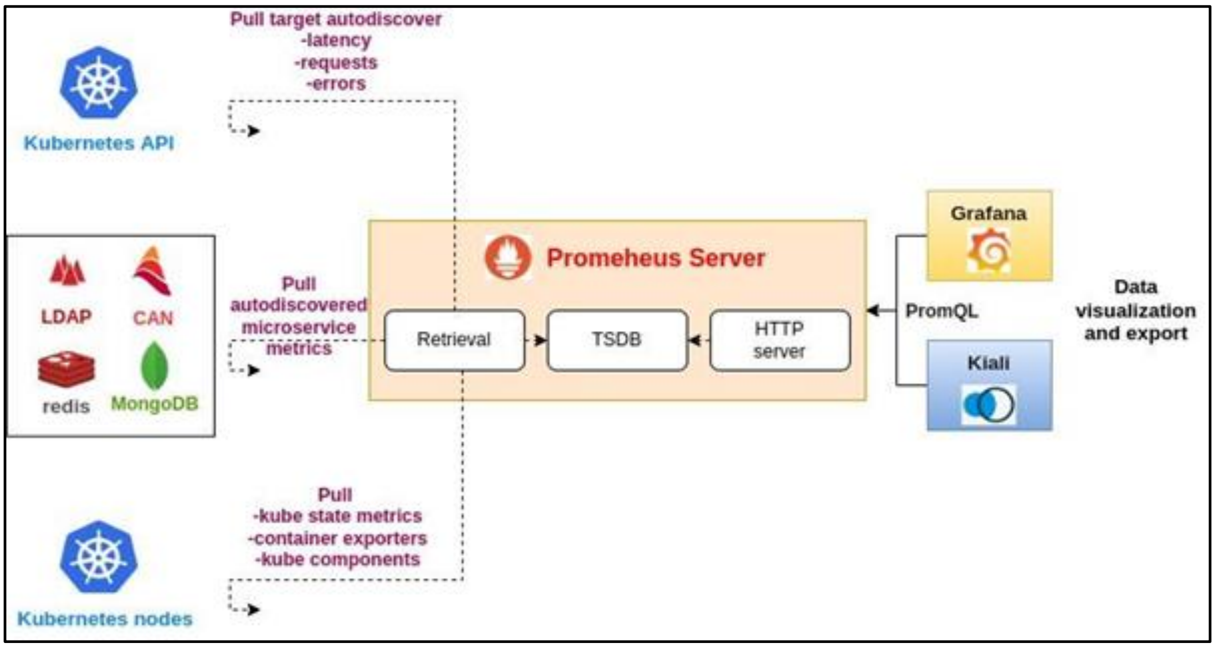
There are several ways to install Promotheus like using helm chart, using manifest file, using docker images and many more. Since CAN uses Istio service mesh, enable "Promethus" as an add-on that comes as a part of its core package. This will ease the installation and execution of Prometheus.

Istio's Role with Prometheus

In an Istio service mesh, each component exposes an endpoint that emits metrics. Prometheus works by scraping these endpoints and collecting the results. To simplify the configuration of metrics, Istio offers a mode of operation called ***"Metrics merging"*** (enabled by default) though which *"prometheus.io"* annotations are added to all data plane pods to set up scraping. If these annotations already exist, they are overwritten. With this option, the Envoy sidecar merges Istio's metrics with the application metrics.

Architecture

Basic architectural view of Prometheus is as shown:



Prometheus server pulls metrics from,

1. Kube API server that is instrumented and exposes Prometheus metrics by default, providing monitoring metrics like latency, requests, errors, etcd and cache status.
2. Auto discovered micro-service metrics from applications like CAN, MongoDB, Worker, Controller etc.
3. Also, Prometheus must collect metrics related to the Kubernetes services and nodes using:

- Container exporters i.e. cAdvisor provides the resource usage and performance characteristics of their running containers such as memory, CPU etc.
- Kube-state-metrics for integrated and cluster level metrics: deployments, pod metrics, resource reservation, etc.
- Kubernetes control plane metrics: kubelet, etcd, dns, scheduler, etc.
- Kubernetes Metric server is a cluster-wide aggregator of resource usage data. It collects resource metrics from the kubelet running on each worker node and exposes them in the Kubernetes API server through the Kubernetes Metrics API.

Prometheus stores all scraped samples locally and runs rules over this data to either aggregate or record new time series from existing data. In this case, **Grafana** and **Kiali** are used to visualize and to generate alerts from the collected data.

Metric Types

Promotheus supports three types of metrics. It is important to understand “what are they?” and “when to use what?” while creating a custom metric.

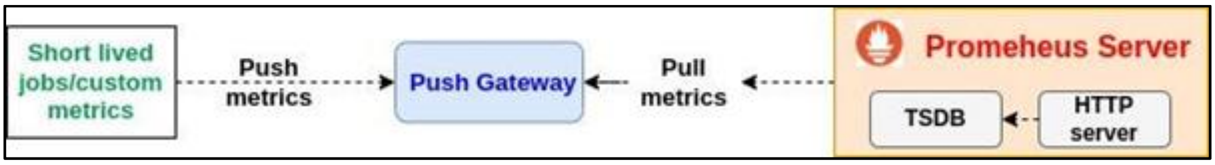
1. **Counter**: A counter is a cumulative metric that represents a single monotonically increasing counter whose value can only increase.
   For example, Number of requests served, tasks completed, or errors.
2. **Gauge**: A gauge is a metric that represents a single numerical value that can arbitrarily go up and down.
   For example, number of currently running processes, number of alarms received within the context of CAN application.
3. **Histogram:** A histogram samples observation and counts them in configurable buckets. It also provides a sum/count/average of all observed values.
   For example, request durations or response sizes.
4. **Summary:** Similar to a *histogram*, a summary samples observation (like request durations and response sizes).
   For example, Number of requests made by Controller to worker within a duration of 5 minutes.

Pushing Custom Metrics

Apart from the metrics scraped by Prometheus, there is need to monitor few components that cannot be scraped. Hence, pushing custom metrics to the Prometheus data source can be accomplished using **“Pushgateway”**.

Pushgateway is a feature of Prometheus that allows ephemeral and batch jobs to expose their metrics to the application. Often, these types of jobs do not have a long enough lifecycle for software programs to scrape them and pull the vast amount of metrics necessary for effective network monitoring.

With Pushgateway, metrics series is continually exposed to Prometheus. This Pushgateway application has client libraries for Go, Java, Scala, Python, and Ruby, making it useful for a wide audience. Java client library is used to push the custom metrics in the example provided later.



Installing PushGateway

1. **Add the required repository:**

```
$ helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
```

2. **Install Pushgateway:**

```
$ helm install pushgateway prometheus-community/prometheus-
pushgateway -n default
```

Configuring Pushgateway as a scrape target

Configure Pushgateway as a Scrape target for Prometheus Server. Add the below lines in scrape_configs section in the prometheus configmap.

```
- job_name: custom-prometheus-pushgateway
  honor_labels: true
  static_configs:
  - targets:
    - pushgateway-prometheus-pushgateway.default.svc.cluster.local:9091
```

If the Prometheus service was already running, stop it and re-start the service.
Use below commands:

```
kubectl delete -f prometheus.yaml -n istio-system
kubectl apply -f prometheus.yaml -n istio-system
kubectl get pods -n istio-system
```

Prometheus Pushgateway is successfully installed and configured. Next step is to push the custom metric to PushGateway from the client application (written in java/shell/Go/python), so that Promotheus can scrape the custom metric.

Prometheus metric format

The pushed metrics are managed in groups, identified by a grouping key of any number of labels, of which the first must be the job label. The groups are easy to inspect via the web interface or to delete them together in the later stages. In general, one has to understand how to read the metric. A metric is composed by several fields:

- Metric name
- Any number of labels (can be 0), represented as a key-value array
- Current metric value

General syntax of metric display is as follows:

```
<MetricName>{<labelName1>=<Value1>......<labelN>=<ValueN>} <metricValue>
```

For example:
istio_requests_total{destination_workload_namespace=~"avanseus-workspace", destination_workload=~"worker", reporter="source", source_workload=~"controller", source_workload_namespace=~"avanseus-workspace"} 345

Where,

- Istio_requests_total is <MetricName>

- destination_workload_namespace=~"avanseus-workspace" is label name and value pair. Many such labels are present in the above example.
- 345 is the value captured at a given point of time for Istio_requests_total metric.

Pushing metric from CLI

As mentioned earlier, it is possible to push the custom metric from CLI via CURL command or through shell scripts. Below are the two different ways to push it using CURL command:

- Push a single sample into the group identified by {job="some_job"}:

```
echo "some_metric 3.14" | curl --data-binary @-
http://pushgateway.example.org:9091/metrics/job/some_job
```

- Push complex sample into the group identified by: {job="some_job",instance="some_instance"}:

```
cat <<EOF | curl --data-binary @-
http://pushgateway.example.org:9091/metrics/job/some_job/instance/some_
instance
  # TYPE some_metric counter
  some_metric{label="val1"} 42
  # TYPE another_metric gauge
  # HELP another_metric Just an example.
  another_metric 2398.283
  EOF
```

Note: From the above example, it infers that pushing complex custom metric from CLI would be tedious. Hence, use client library to do the same.

Pushing metric from Java-client

To better understand this, consider a scenario wherein define a custom metric called “Total_Processed_Line_Count” to know the number of lines processed from the provided input file as received from Controller application from the CAN (consumer) module during a prediction run.

Ideally, after the Controller module completes its batch job, “Total_Processed_Line_Count” metric should give the same count value as that of the number of lines present in the input file. This metric also helps to understand the number of predictions happened for the given input file.

In order to do this we need to do the following steps in sequence:

1. Add push gateway dependency in your pom.xml

```
<!-- Pushgateway exposition-->
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient_pushgateway</artifactId>
  <version>0.16.0</version>
</dependency>
```

2. Create a collector registry and pushgateway object.

```
CollectorRegistry pushRegistry = new CollectorRegistry();
```

```
PushGateway pushGateway = new PushGateway("pushgateway-prometheus-
pushgateway.default.svc.cluster.local:9091"); //EndPoint of your pushGateway server
```

3. Create the object of appropriate metric type (counter/guage/histogram) and register with required CollectorRegistry by specifying 'metricName' and other optional attributes. Then calculate the metric value and push it to the gateway.

```
CollectorRegistry pushRegistry = new CollectorRegistry();
Gauge gauge = (Gauge) Gauge.build().name("total_processed_line_count").help("Total
number lines processed in the input file.").register(pushRegistry);
gauge.set(lineCount); //your logic to calculate lineCount
Map<String,String> queryAttributesMap=new HashMap<>();
queryAttributesMap.put("application","controller");
queryAttributesMap.put("namespace","avanseus-workspace");
queryAttributesMap.put("predictionInputFile",inputFilePath);
pushGateway.push(pushRegistry, "avanseus_prediction_count", queryAttributesMap);
```

The above example is already implemented in the Avanseus Controller application which is responsible for performing batch prediction by accepting prediction input file from the consumer module (CAN) as a multipart request. Because of which a custom metric named "Total_Processed_Line_Count" is readily available in Avanseus Grafana dashboard.

Data Retention in Prometheus

Sometimes adding many custom metrics may result in excessive disk usage. Given that disk space is a finite resource, limit must be set on how much of it Prometheus will use.

By default, Prometheus keeps 15 days of data. However, the default value can be changed using *"--storage.tsdb.retention.time"* flag when Prometheus starts. It defines the duration the data has to be kept in the time-series database (TSDB).

Pushing a custom metric at a lesser frequency (~15 secs or 1 min) and keeping 15 days of metric-data consumes huge disk space. In similar cases, Avanseus is not interested in persisting 15 days of historical metric data instead it is willing to store a lesser period of data (~4 hrs or 1 day). However, Prometheus does not support multiple retention policies. Only a single retention policy can be configured for all the stored data.

Therefore, the easiest solution would be to run **multiple Prometheus instances** with distinct scrape configs and distinct retention periods.

Configuring Additional Instance of Prometheus

This is an optional step. Install an additional instance of Prometheus if and only if a custom metric data is pushed at a lesser frequency (~15 seconds or 1 min or 5 minutes). The idea here is to configure this Prometheus instance with a custom data retention policy. By default, Prometheus keeps 15 days of data. Hence, collecting custom metric at lesser frequency there is a high probability of running out of space. The solution is to install an additional Prometheus instance with custom data retention policy (~2 hours or 4 hours or 2 days) as shown in the below steps:

1. **Create a namespace :**

```
$ kubectl create ns prometheus
```

2. **Install Prometheus:**
   Go to kubernetes_resources/Helm_Charts/NFS_STORAGE_HELM/ folder.
   Use the folder “**avanseus-prometheus-chart**” and execute the below command to install prometheus:

   ```
   $ helm install prometheus ./avanseus-prometheus-chart –n
   prometheus
   ```

   The above command will install another instance of Prometheus with the data retention period of 2-hour i.e. Prometheus can keep data only for 2 hours.

In this case, use **prometheus-pushgateway.prometheus.svc.cluster.local:9091** as a pushGateway server endpoint when you push metrics using CLI or Java Client.

Future Scope of Work

In the previous section, an example is highlighted on how to push a custom metric (named “Total_Processed_Line_Count”) to prometheus pushgateway from client application. This idea can be extended for tracking any business critical metrics like *“Total_Alarm_Count”, “Total_ApplicationException_Count”, “Duration_Of_Dataload”* and many more in future days. These metrics can be viewed from the Grafana dashboard by configuring the appropriate panels and it is possible to create alerts and notifications through which pro-active actions can be taken to avoid the possible damage.