Intelligent and Secure Networks - Project Presentation

Del corso del

Prof. Mauro Femminella

Venturi Marco, 346951

Perugia, Anno Accademico 2023/2024
Università degli Studi di Perugia
Corso di laurea magistrale in Ingegneria Informatica e Robotica
Curriculum Data Science
Dipartimento di Ingegneria

Contents

1 **Prometheus project and goals** 2

2 **Prometheus installation** 3

3 **Kube-state-metrics** 8

4 **Simulation service** 9

5 **JSON pseudo-storage service** 12

6 **Conclusion** 14

1. Prometheus project and goals

Prometheus is an open-source monitoring and alerting toolkit, integral to the Cloud Native Computing Foundation. Its multi-dimensional data model captures time series data, identified by metric names and key/value pairs, collected via a pull model. PromQL, its powerful query language, enables users to analyze and aggregate metrics for system insights. Supporting various service discovery mechanisms, Prometheus excels in dynamic environments, making it popular in container orchestration systems like Kubernetes. The toolkit includes an alerting system, permitting users to define rules and receive notifications through diverse channels. Prometheus stores data locally in an efficient time-series database, with configurable retention policies. Often paired with Grafana for visualization, Prometheus integrates seamlessly with third-party systems through exporters. With an active community and a scalable design, Prometheus is widely adopted for monitoring large, distributed environments. Federation enables collaboration between multiple Prometheus instances, facilitating monitoring across diverse clusters or geographic locations. Overall, Prometheus stands as a robust and versatile solution for achieving observability and reliability in modern, dynamic systems.

The aim of this tutorial is to implement a Prometheus server on a single-node Kubernetes cluster. In this scenario, Prometheus gathers data not only from metrics endpoints but also from the kube-state-metrics service. Another component of this project is a Node.js application designed to periodically save data from the Prometheus scrape system. Inside the cluster, a Node.js server is also deployed for simulation purposes, featuring custom metrics. All code can be find here

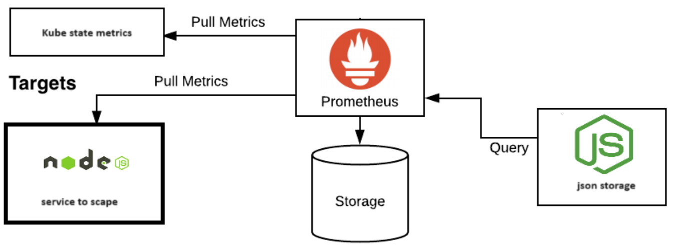

Figure 1.1: Project schema.

2. Prometheus installation

Step 1: Create Namespace

First, i create a Kubernetes namespace for all monitoring components. All the Prometheus kubernetes deployment objects get deployed on the monitoring namespace, in this way i organize and isolate Prometheus resources.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: monitoring
```

Step 2: Define ClusterRole

Prometheus uses Kubernetes APIs to read all the available metrics from Nodes, Pods, Deployments, etc. For this reason, i need to create an RBAC policy with read access to required API groups and bind the policy to the monitoring namespace.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - nodes
  - nodes/proxy
  - services
  - endpoints
  - pods
  verbs: ["get", "list", "watch"]
# ... (additional rules)
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: default
  namespace: monitoring
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
  namespace: monitoring
```

Step 3: Create ConfigMap

All configurations for Prometheus are part of prometheus.yaml file and all the alert rules for Alertmanager are configured in prometheus.rules.

- prometheus.yaml: This is the main Prometheus configuration which holds all the scrape configs, service discovery details, storage locations, data retention configs, etc).
- prometheus.rules: This file contains all the Prometheus alerting rules.

By externalizing Prometheus configs to a Kubernetes ConfigMap, the Prometheus image doesn't need to be rebuilt whenever I need to add or remove a configuration. It is enough to update the ConfigMap and restart the Prometheus pods to apply the new configuration.

```
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: prometheus-config
  labels:
    name: prometheus-config
  namespace: monitoring
data:
  prometheus.rules: |-
    # ... (Prometheus alerting rules)
  prometheus.yml: |-
    # ... (Prometheus configuration)
```

The prometheus.yaml contains all the configurations to discover pods and services running in the Kubernetes cluster dynamically. i have implemented the following scrape jobs in our Prometheus scrape configuration:

- kubernetes-apiservers: It gets all the metrics from the API servers.
- kubernetes-nodes: It collects all the kubernetes node metrics.
- kubernetes-pods: All the pod metrics get discovered if the pod metadata is annotated with prometheus.io/scrape and prometheus.io/port annotations.
- kubernetes-cadvisor: Collects all cAdvisor metrics.
- kubernetes-service-endpoints: All the Service endpoints are scrapped if the service metadata is annotated with prometheus.io/scrape and prometheus.io/port annotations. It can be used for black-box monitoring.

Step 4: Deploy Prometheus

Create a file named prometheus_deployment.yaml and copy the following contents onto the file. In this configuration, we are mounting the Prometheus config map as a file inside /etc/prometheus as explained in the previous section.

```
# ... (Deployment template)
    spec:
      containers:
      - name: prometheus
        image: quay.io/prometheus/prometheus:v2.0.0
        imagePullPolicy: IfNotPresent
        args:
```

```
     - '--storage.tsdb.retention=6h'
     - '--storage.tsdb.path=/prometheus'
     - '--config.file=/etc/prometheus/prometheus.yml'
    command:
    - /bin/prometheus
    ports:
    - name: web
      containerPort: 9090
    volumeMounts:
    - name: config-volume
      mountPath: /etc/prometheus
    - name: data
      mountPath: /prometheus
   restartPolicy: Always
   securityContext: {}
   terminationGracePeriodSeconds: 30
   volumes:
   - name: config-volume
     configMap:
       name: prometheus-config
   - name: data
     emptyDir: {}
```

I'm not using any persistent storage volumes for Prometheus storage as it is a basic setup. When setting up Prometheus for production uses cases, make sure you add persistent storage to the deployment.

Step 5: Define Prometheus Service

To access the Prometheus dashboard over a IP or a DNS name, you need to expose it as a Kubernetes service. Create a file named prometheus_nodeport.yaml and copy the following contents. We will expose Prometheus on all kubernetes node IP's on port 30900.

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus
  namespace: monitoring
spec:
```

```yaml
  selector:
    app: prometheus
  type: NodePort
  ports:
  - name: prometheus
    protocol: TCP
    port: 9090
    nodePort: 30900
```

Step 8: Access Prometheus

After deploying, access Prometheus via the NodePort configured in the Service. For example, if the nodePort is set to 30900, you can access Prometheus at `http://<node-ip>:30900`. Another way for access prometheus dashboard is using kubectl port forwarding, you can access a pod from your local workstation using a selected port on your localhost. This method is primarily used for debugging purposes. using this command `kubectl port-forward -n monitoring <prometheus-deployment> 8080:9090`

Note: Ensure RBAC roles are correctly assigned, and the specified images are available. This guide assumes a basic understanding of Kubernetes concepts and a pre-existing Kubernetes cluster. Adjust parameters based on your environment and requirements.

3. Kube-state-metrics

Kube-State-Metrics is an add-on service that listens to the Kubernetes API server and generates metrics about the state of the objects. The implementation of Kube-State-Metrics is necessary for Prometheus configuration, as all the metrics gathered by Kube-State-Metrics are directly ingested by Prometheus, eliminating the need for manual configuration in Prometheus to scrape these metrics. Kube-State-Metrics is available as a public docker image, for use it is necessary to deploy the following Kubernetes objects for Kube-State-Metrics to work:

- A Service Account
- Cluster Role – For Kube-State-Metrics to access all the Kubernetes API objects.
- Cluster Role Binding – Binds the service account with the cluster role.
- Kube-State-Metrics Deployment
- Service – To expose the metrics

The purpose of this guide is not to implement Kube-State-Metrics, but only to show how to implement a functional monitoring system, so the above yml and configuration are not showed in this document, but can be find here

4. Simulation service

The provided JavaScript code is an example of a simple Express.js web server that exposes custom metrics for Prometheus monitoring. Here is a breakdown of key components related to Prometheus custom metrics:

1. **Default Metrics:**

```
prometheus.collectDefaultMetrics({ register:
 prometheusRegistry });

```

   This line initializes and collects default metrics provided by the `prom-client` library. These metrics include information about the Node.js process (CPU usage, memory usage, etc.) and are registered with the `prometheusRegistry`.

2. **Default Labels:**

```
prometheusRegistry.setDefaultLabels({
  app: 'simulation'
});

```

   Sets default labels for application metrics. In this case, it sets the label app to the value `'simulation'`.

3. **Custom Counter Metric:**

```
const numberOfRequests = new prometheus.Counter({
  name: 'simulation_app_requests_total',
  help: 'Total number of requests to the
 simulation app',
  labelNames: ['method', 'route', 'statusCode'],
  registers: [prometheusRegistry],
});

```

Declares a custom counter metric named `simulation_app_requests_total`. It counts the total number of requests to the simulation app and includes labels for `method`, `route`, and `statusCode`.

4. **Custom Histogram Metric:**

```
const requestDurationHistogram = new
 prometheus.Histogram({
  name:
 'simulation_app_request_duration_milliseconds',
  help: 'Histogram of request durations for the
 simulation app in milliseconds',
  labelNames: ['method', 'route', 'code'],
  registers: [prometheusRegistry],
  buckets: [1,2,3,4,5,10,25,50,100,250,500,1000],
});

```

Declares a custom histogram metric named `simulation_app_request_duration_milliseconds`. It measures the duration of requests and includes labels for `method`, `route`, and `code`. The specified `buckets` parameter defines the boundaries for the histogram.

5. **HTTP Routes and Metric Observations:**

```
app.get('/', (req, res) => {
  // ... (route logic)
  requestDurationHistogram.labels(req.method,
 req.route.path,
 res.statusCode).observe(responseTimeInMilliseconds)
});

app.get('/simulate-requests', (req, res) => {
  // ... (route logic)
  requestDurationHistogram.labels(req.method,
 req.route.path,
 res.statusCode).observe(responseTimeInMilliseconds)
```

```
  });

```

Defines HTTP routes, including a simulated request route. Observations are made on the `requestDurationHistogram` for the duration of each request.

6. **Metrics Endpoint:**

```
  app.get('/metrics', (req, res) => {
    res.set('Content-Type',
   prometheus.register.contentType);
    prometheusRegistry.metrics().then(data =>
   res.status(200).send(data));
  });

```

Exposes a `/metrics` endpoint that returns the registered metrics in the Prometheus exposition format. The response includes the default metrics and the custom metrics defined earlier.

The repository contains a simulation_script.py Python script used to simulate 'n' requests for each simulation server endpoint, testing the functionality of custom metrics.

5. JSON pseudo-storage service

```
  // Function to retrieve data from the specified HTTP endpoint and save it as a JSON file
async function fetchDataAndSave() {
  try {
    // Make an HTTP GET request to the Prometheus endpoint
    const response = await axios.get(’http://prometheus.monitoring.svc.cluster
    .local:9090/api/v1/query?query={__name__!=""}’);

    // Extract data from the response
    const jsonData = response.data;

    // Generate a timestamp and construct a filename
    const timestamp = new Date().toISOString();
    const filename = ‘data_${timestamp}.json‘;

    // Write the JSON data to a file
    fs.writeFileSync(filename, JSON.stringify(jsonData, null, 2));

    // Log a message indicating the successful data save
    console.log(‘Data saved to ${filename}‘);
  } catch (error) {
    // Handle errors, log an error message
    console.error(’Error fetching data:’, error.message);
  }
}

```

The fetchDataAndSave function is designed to be called periodically, and each time it executes, it performs the following steps. Firstly, it makes an HTTP GET request to a specified Prometheus endpoint using the axios library, targeting the \api\v1\query endpoint with a query parameter ({__name__!=""}) to fetch all time series with non-empty metric names. The received data, typically containing metric values and related details, is then extracted from the response. Subsequently, a timestamp is generated in ISO format, and a unique filename is constructed incorporating this timestamp. The function proceeds to write the extracted JSON data to a file using the fs module, with the filename reflecting the timestamp for identification.

6. Conclusion

In this Prometheus tutorial project, you can explore the basic functionality of Prometheus with a closer look at all the tools available for monitoring the cluster and services. Using a Node.js server to store long-term time series data is not the recommended approach; typically, tools like Thanos or Grafana are employed for such purposes. In this case, I opted not to implement additional services to avoid complicating the situation and exceeding the scope of the guide.

For further studies, it would be interesting to delve into advanced topics such as the Prometheus Operator, Prometheus Federation, Alarm Manager, and the logic behind remote_write or remote_read. These concepts can enhance your understanding of Prometheus and its capabilities for more complex monitoring scenarios.