# Acala: Aggregate Monitoring for Geo-Distributed Cluster Federations

Chih-Kai Huang, Guillaume Pierre

# Acala: Aggregate Monitoring for Geo-Distributed Cluster Federations

Chih-Kai Huang
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
chih-kai.huang@irisa.fr

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
guillaume.pierre@irisa.fr

## ABSTRACT

Distributed monitoring is an essential functionality to allow large cluster federations to efficiently schedule applications on a set of available geo-distributed resources. However, periodically reporting the precise status of each available server is both unnecessary to allow accurate scheduling and unscalable when the number of servers grows. This paper proposes Acala, a monitoring framework for geo-distributed cluster federations which aims to provide the management cluster with aggregate information about the entire cluster instead of individual servers. Our evaluations, based on actual deployment under controlled environment in the geo-distributed Grid'5000 testbed, show that Acala reduces the cross-cluster network traffic by up to 99% and the scrape duration by up to 55%.

## CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Computer systems organization** → **Cloud computing**;

## KEYWORDS

Monitoring, Geo-distributed cluster federations, Prometheus, Metrics aggregation and deduplication, Fog computing.

## 1 INTRODUCTION

The rapid development of edge and fog computing technologies creates new opportunities to deploy very large geo-distributed platforms covering a region or even an entire country [10]. Managing such platforms requires an efficient resource orchestrator, which enables administrators to treat the set of machines located in numerous strategic locations with the same flexibility as if they were a single homogeneous cluster. Many research projects are aiming at reusing and/or extending the popular Kubernetes (K8s) orchestrator in this new geo-distributed context [8, 37]. When the system size grows, and in the possible presence of unreliable network connections between the available resources, it quickly becomes desirable to organize the platform as a *federation* of multiple independent geo-distributed clusters, each of which is in charge of the resources located in a particular region [23].

In a cluster federation, a "management cluster" is in charge of deciding which of the "member clusters" will be in charge of handling each newly deployed application. Although the original KubeFed project allowed little control of the choice of member cluster [23], newer designs support a range of fine-grained placement policies based on metrics such as cluster load, location, and network usage [32]. These policies base themselves on detailed monitoring information about the status of available resources, provided by a robust monitoring framework such as Prometheus and its extension Prometheus Federation [26].

We demonstrate in this paper that monitoring a large cluster federation is a very challenging task because the number of metrics and the volume of monitoring data to be reported to the management cluster grows linearly with the system size. Even for medium-sized clusters, the necessary monitoring network traffic grows to such large values that it may represent the majority of the system management traffic, and may eventually saturate the existing inter-cluster network links. We however note that the fine-grained monitoring data that are being reported to the management cluster are in fact not necessary to support the cluster federation. We therefore aim to reduce the volume of management data to provide the cluster federation with accurate and up-to-date information while significantly reducing the networking overhead of the federated monitoring framework itself.

This paper proposes Acala, an extension of Prometheus which reports information about entire member clusters rather than the individual servers within them. It uses two techniques to reduce the number of metrics to be reported to the management cluster: *metrics aggregation* merges together the metric values of multiple servers to report the aggregate status of the entire cluster rather than its individual servers; and *metrics deduplication* avoids one to repeatedly report the same metrics in case their value does not change significantly.

Our evaluations based on actual deployment in the Grid'5000 testbed [5] show that Acala can reduce the volume of cross-cluster network traffic by up to 99%, while reducing the necessary time to scrape metrics by up to 55%. Moreover, the resource usage of Acala components also remains acceptable.

The rest of this paper is organized as follows. Section 2 discusses the motivation behind this work. Background and Related Work are expressed in Section 3. Section 4 introduces how Acala works and two data reduction strategies. Section 5 shows the evaluations of the proposed framework and strategies, the conclusions of the paper are in Section 6.

## 2 MOTIVATION

Managing multiple geo-distributed federated clusters like a fog computing platform or a telco cloud use case is a difficult challenge. Some works focus on the management problem of multiple Kubernetes clusters, such as Kubernetes Cluster Federation (KubeFed) and multi-cluster Kubernetes (mck8s). KubeFed [23] is proposed to empower users to manage multiple Kubernetes clusters from one main cluster. However, in its current design, KubeFed mainly places workloads manually with limited support for automated policy-based scheduling among the available clusters. This design

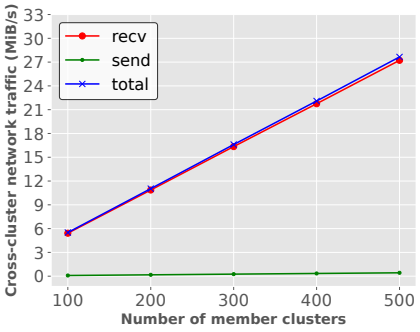



**Figure 1: Cross-cluster network traffic in the management cluster when using mck8s.**

therefore makes it hard to manage the workloads in a large-scale environment automatically. mck8s [32] extends KubeFed and provides automatic placement, scaling, and bursting of container-based applications in geo-distributed cluster federations. However, both works use a centralized control method to manage the resources, which necessarily implies possible scalability issues.

To illustrate this problem, we leverage a real deployment in the Grid'5000 testbed. In the setup, we use "Kubernetes in Docker" (kind) to launch large numbers of Kubernetes clusters [20]. The first cluster acts as our management cluster. Then, we launch up to 500 member clusters. Each cluster contains two servers (one control plane and one worker node), resulting in up to 1000 nodes in total.

Figure 1 depicts the aggregate volume of cross-cluster network traffic after deploying a large mck8s federation with no application workload. *Recv* and *send* show the network traffic received/sent by the management cluster. We sum *Recv* and *send* as the *total* network traffic. The scrape interval of Prometheus in mck8s is set to 5 seconds, which means that the management cluster fetches metrics from every cluster once every 5 seconds. We observe a linear growth up to 27.7 MiB/s for monitoring 500 member clusters (1,000 nodes), which may be enough to saturate many fog computing networks. The same linear growth appears when increasing the number of servers per cluster (not shown in the figure for clarity reasons). This very large management traffic is due to the resource monitoring used by mck8s to implement sophisticated scheduling functionalities. It does not appear when using KubeFed, which schedules workloads without considering the cluster status.

This simple experiment motivates our work: we aim to support advanced scheduling policies but without paying the price of detailed resource monitoring. As a result, the precious platform's network resources may be used for actual user workloads rather than cluster management operations.

## 3 BACKGROUND AND RELATED WORK

Fog computing extends the cloud computing concept with additional resources located closer to the end-users. It has received much attention from academia in the last few years. Many prior studies present different facets of fog/edge computing, including placement of jobs and services [12], service caching [15], seamless application migration [31], and supporting data stream processing [1]. These works are based on a single distributed cluster, which will necessarily face the scalability problem. To handle this issue, we are now witnessing an increasing adoption of geo-distributed multi-cluster deployments. Some works focus on job scheduling [16], whereas others address resources management [17, 32] and fault prediction [30]. These studies rely on a monitoring system to collect the metrics. However, they do not aim to solve the problem of monitoring itself in a geo-distributed cluster federation.

The main purposes of geo-distributed resource monitoring are to track resource usage of the computing nodes, especially in potentially resource-restricted and unstable environments. A number of open source and commercial monitoring tools for cloud platforms such as DARGOS [25], JCatascopia [36], and Nagios [24] are not considered suitable for fog computing environments [11]. On the other hand, some authors present monitoring solutions and architectures that are designed with the specific constraints of fog computing in mind, such as PyMon [13], FMonE [4] and Prometheus [28]. The most popular of these tools is Prometheus. Since 2016, Prometheus has been accepted by the Cloud Native Computing Foundation (CNCF) as a "graduated" project, which shows its great potential in conjunction with Kubernetes. At the same time, many works use Prometheus as their monitoring solution [2, 6, 7, 14, 22, 34].

Prometheus provides a function called "Federation" which allows a Prometheus server to collect the metrics from other Prometheus servers. A common use case is building a global-view Prometheus server which scrapes and stores the monitoring data from other Prometheus servers. Two levels of the federation are instance-level drill-down and job-level drill-down. In Prometheus terminology, an *instance* is an endpoint that user can scrape from and a *job* is a collection of *instances* with the same purpose. Prometheus Federation is often used to monitor systems in geo-distributed environments [3, 9, 32, 35].

However, Prometheus Federation features three limitations that generate the high network traffic highlighted in Section 2 and make it unsuitable for our purposes. First, the highest scrape level of Prometheus Federation is job-level, and it uses the *match* mechanism to select the series of metrics. For example, the operator can write job="node-exporter" in a federation server's configuration file to scrape the metrics that match this label from the target Prometheus servers. It results in scraping the matching metrics from all the nodes[1] in the target cluster when job="node-exporter" is set. This design is suitable for backing the metrics for high availability purposes but not fitting for the management cluster to manage the federated clusters. It wastes the network bandwidth to transmit and disk resources to save the same node metrics in the management cluster. Second, Prometheus Federation will append all original labels in each metric when a Prometheus server scrapes from the target Prometheus server to identify where the metric comes from. However, all original labels are unnecessary for recognition, and the scheduler may not need this detailed information to make the decision. Furthermore, the labels are attached before the metrics transmission, which increases the cross-cluster network traffic. The third point is that Prometheus Federation collects the monitoring data at a fixed periodicity. The system will therefore scrape all the

[1] We assume all nodes in all clusters have installed node-exporter and labeled job="node-exporter".

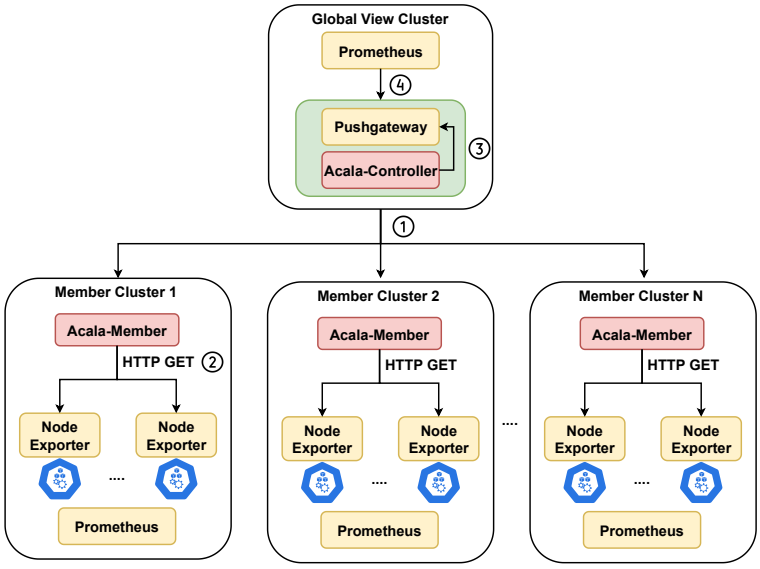



**Figure 2: Overview of Acala architecture and scrape flow.**

metrics even if some of the metric values did not change, which once again will waste network bandwidth.

Prometheus also supports a feature called "recording rules" which is similar to Acala's metrics aggregation. Using it, one can pre-aggregate selected metrics, store the results in the member clusters, and scrape them from other Prometheus servers with appropriate labels. However, recording rules in Prometheus need to be defined manually for each metric in each member cluster, which is error-prone and may increase the deployment and configuration cost in large-scale environments. Moreover, Prometheus does not provide metric deduplication so it reports data to the global view cluster periodically, regardless of whether the value has changed since the previous scraping period.

To overcome these monitoring challenges, we base our work on Prometheus and introduce Acala. Acala automatically aggregates the metrics whose metric name and labels are identical in different servers, which reduces the cross-cluster network traffic as well as the deployment and configuration cost. It also deduplicates metric values and thereby avoids transferring unchanged values over and over again.

## 4 SYSTEM DESIGN

The objective of this work is to monitor resources with lower cross-cluster network traffic in geo-distributed cluster federations. In this section, we discuss the operation of Acala and introduce two data reduction strategies specially designed for Acala to reach our goal.

### 4.1 System Model

A geo-distributed cluster federation is a set of multiple clusters that can choose clusters to be global view clusters, which are responsible for collecting metrics from other clusters. Clusters that are not selected are member clusters. Each cluster consists of several computing nodes, and we assume that each node in the cluster has enough resources that can run the applications to provide monitoring. All nodes in a cluster are located in the same area. Each node and cluster is connected by the network and can communicate. Although the current design can support multiple layers, for the sake of simplicity, we leverage a two-tier architecture in this paper.

Acala is built on several components from the Prometheus ecosystem, including the Prometheus server, node-exporter [27], and Pushgateway [29]. The system overview is shown in Figure 2.

*Prometheus servers in member clusters:* The duty of these servers is to scrape [2] time-series data about local metrics in each member cluster, and to store them in their local database. They constitute the source of data before aggregation. They can also be used for querying detailed per-node metrics, for example for anomaly detection, diagnosis or system management purposes. Moreover, these Prometheus servers can also be configured to trigger alerts about nodes with abnormal metric values in their member cluster, such as fully saturated nodes.

*Prometheus server in the global view cluster:* The Prometheus server in the global view cluster is used to save the aggregated data from the member clusters and their local metrics. The federation's scheduler can leverage this Prometheus server to query member cluster information and make the scheduling decisions.

*Node-exporter:* This component is our monitoring agent for the per-node metrics. We install a node-exporter for each node in each cluster to expose hardware and operating system metrics.

*Pushgateway:* The Pushgateway is installed in the global view cluster. It is a middleware that can expose these metrics for the Prometheus server to scrape. Moreover, Pushgateway also acts as a cache for metric values.

Acala introduces two new components: Acala-Controller, and Acala-Member. Acala-Controller is responsible for scraping the metrics from the target member cluster, adding the labels to identify

[2] Scrape is the action from Prometheus or Acala to fetch metrics from the target.

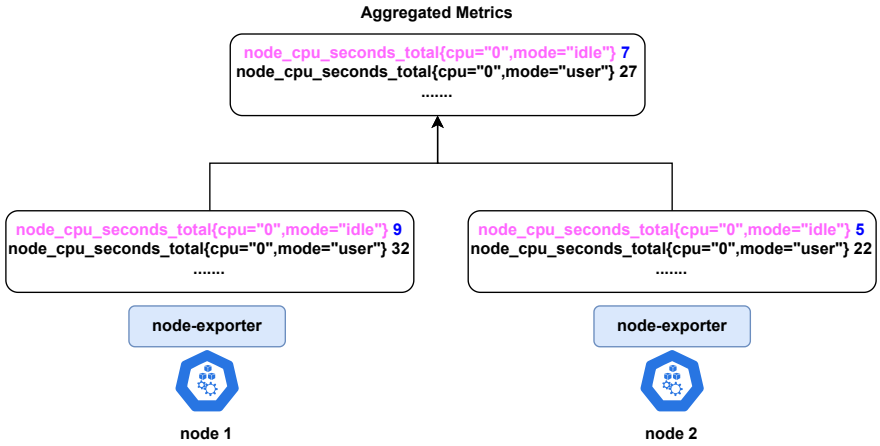



**Figure 3: An example of metrics aggregation.**

the member cluster, and pushing the metrics to the Pushgateway. The task of Acala-Member is to pull the metrics from node-exporter in each cluster and execute proposed data reduction strategies. The data transmission between Acala-Controller and Acala-Member is compressed using gzip. The detailed scrape steps are as follows:

Step 1: When it is time for Acala-Controller to scrape the metrics, the controller sends a request to the target Acala-Member.

Step 2: After Acala-Member receives the request, Acala-Member uses the HTTP GET method to pull the metrics from the local computing nodes through node-exporter. Meanwhile, Acala-Member executes Algorithm 1 to modify the metrics. Finally, Acala-Member compresses the metrics and sends them back to Acala-Controller.

Step 3: Acala-Controller decompresses the metrics and leverages the HTTP POST method to push metrics to the Pushgateway. In this step, the Acala-Controller adds the labels (IP address of control plane and cluster name) to identify the member cluster.

Step 4: The global-view Prometheus server periodically scrapes the metrics from the Pushgateway (at a user-defined periodicity independent from the periodicity of inter-cluster metrics transfer) and stores them locally. The administrator or federation scheduler can then query the monitoring data of the member cluster via this Prometheus server.

## 4.2 Timing to Scrape Metrics

Similar to the original design of Prometheus, the timing to scrape the metrics from the target member cluster is determined by a fixed scrape interval. We leverage a timer in the Acala-Controller to perform regular scrape actions. When the timer counts down to 0, the system scrapes the metrics once, then sets the timer back to the default values configured by the administrator. A shorter scrape interval value means that data in the global view clusters will be more precise in representing the actual status of the member cluster, yet at the cost of additional inter-cluster network traffic. The default scrape interval is defined as 5 seconds.

## 4.3 Data Reduction Strategies

To address the problems mentioned in Section 3, we propose two data reduction strategies: metrics aggregation and metrics deduplication highlighted in Algorithm 1.

The data model of metrics in Prometheus is composed of a *metric_name*, any number of pairs *label_name*, *label_value*, and finally a *metric_value*. The notation of a metric is:

$$metric\_name\ \{label\_name = label\_value,\ \ldots\}\ metric\_value$$

*4.3.1 Metrics Aggregation.* Each node in the member clusters deploys the node-exporter to expose its node-related metrics. In standard Prometheus Federation design, the highest scrape level is *job*, which will scrape the metrics from all nodes in the target member cluster and append all original labels for these metrics. In contrast, we choose metrics aggregation between the nodes in the target member cluster as our solution, elevating the point of monitoring view from node to cluster. For easy understanding, we use *metric name with labels* to represent *metric name*, *label name*, and *label value*.

Figure 3 presents an example of metrics aggregation. Node-exporter of node 1 exposes the metric *node_cpu_second* $\{cpu =$ "0", $mode =$ "*idle*"$\}$ 9 and node 2 has the same metric name with labels (fuchsia color). Metrics aggregation will thus aggregate both metric names with their labels and metrics values (blue color). For the value of the metric, we simply average the individual values together, considering that the federation schedulers only need to know about the general status of the cluster rather than detailed per-server metrics. The node-exporter exposes node-related metrics such as utilized CPU, memory, and network bandwidth. These metrics can be aggregated with other metrics with the same name and labels. When metrics do not have identical name and labels within the cluster, Acala reports them non-aggregated to the global view cluster. In case more detailed per-node information is needed, the administrator can request the Prometheus server deployed in each cluster directly.

The main idea of this strategy is to average values whose metric name and labels are identical in different servers. This method

**Algorithm 1:** metrics aggregation and deduplication

**Output:** $AM$: A dictionary of Aggregated Metrics ($key$: metric name with labels, $value$: metric value)

```
1  Function Aggregation(M_node, AM, counter):
2      if AM == ∅ then
3          AM ← M_node
4          for key ∈ AM do
5              counter_key ← 1
6      else
7          for key, value ∈ M_node do
8              if key ∈ AM then
9                  AM_key ← AM_key + value
10                 counter_key ← counter_key + 1
11             else
12                 AM_key ← value
13                 counter_key ← 1
14     return AM, counter
15 Function Deduplication(LastAM, AM):
16     if LastAM != ∅ then
17         for key ∈ AM do
18             if LastAM_key != AM_key then
19                 DAM ← key and AM_key
20     else
21         DAM ← AM
22     return DAM
23 Function Main:
24     while true do
25         Wait for connection
26         if Received scraping request then
27             M_node ← Pull Metrics from each node in cluster
28             for Each node in cluster do
29                 AM, counter ← Aggregation(M_node, AM, counter)
30             Update value of AM to average values based on AM and
                 counter
31             if Deduplication is enabled then
32                 TempAM ← AM
33                 AM ← Deduplication(LastAM, AM)
34                 LastAM ← TempAM
35             Compress AM
36             send AM back to Acala-Controller
37             AM ← ∅
38             counter ← ∅
```

can solve two problems we mentioned before. The first problem is to collect monitoring data for each node in the target member cluster. Metrics aggregation merges the metrics from all nodes into a single aggregated metrics, which can decrease the volume of monitoring data to reduce cross-cluster network traffic. Moreover, metrics aggregation averages metrics values to represent the overall cluster status. This is in line with other related work [32] which also applies the aggregating strategy to represent the overall cluster resources situation. However, they perform aggregation *after* all individual metrics have been scraped, transferred and stored in the global view cluster.

As discussed in Section 3, Prometheus Federation adds all original labels in each metric to identify which server each metric belongs to. In contrast, metrics aggregation keeps the metric labels unchanged, the same as before aggregation. For the cluster information, we add the labels including the IP address of control plane and cluster name (set by administrators manually) to indicate the member cluster in Acala-Controller, which takes place after the transmission. Therefore, metrics aggregation can reduce cross-cluster network traffic.

*4.3.2 Metrics Deduplication.* Prometheus Federation blindly scrapes metrics from the member clusters at a periodic interval. As a result, in case some metrics values do not change frequently, they get transferred repeatedly and unnecessarily, which consumes network bandwidth to transfer these redundant data. To further reduce cross-cluster network traffic, we propose a second data reduction strategy – *metrics deduplication*.

Metrics deduplication compares the metric values with the most recently transferred one. If the values are identical, the deduplication strategy removes this metric from this metric value transfer. On the other hand, if the metric value changes, the system will include this metric again to report the fresh data.

However, note that Prometheus includes a metrics staleness mechanism. If no new value is reported after 5 minutes (default of Prometheus), this metric will be marked as stale and its value will be excluded from results returned to the federation scheduler. When using metrics deduplication, this staleness mechanism may exclude valuable deduplicated values from the results. Therefore, Acala leverages Pushgateway to cache these metrics locally so that the Prometheus server in the global view cluster can scrape from Pushgateway and keep fresh metric values in the Prometheus server without having to repeatedly transfer them from the member clusters.

To allow Acala to perform both metrics aggregation and deduplication, the algorithm will perform aggregation first and then deduplication based on the aggregated data. Although both data reduction strategies may run independently, we leave this topic for future work.

The metrics aggregation and deduplication process are illustrated in Algorithm 1. When a request for a new scrape action arrives at the Acala-Member in the target member cluster, the Acala-Member pulls the metrics from each node through node-exporter (lines 26-27). If the metric names with labels are identical, the value of matched metrics get summed and the counter for this metric is incremented (lines 8-10). However, if the metric does not have the same metric name with labels, the algorithm appends this metric as a new metric and sets its counter to one (lines 12-13). After all metrics finish aggregation, the value of metrics are averaged (line 30). When the deduplication function is enabled, the algorithm copies the full Aggregated Metrics ($AM$) to $TempAM$. Then, it executes the deduplication function to remove the metrics with the same value as the last scrape (lines 16-22). $LastAM$ gets the metrics from $TempAM$, which are full aggregated metrics that can be compared for the next scrape (line 34). Finally, Acala-Member compresses the metrics, sends them back to Acala-Controller, clears the data, and waits for the next scrape request (lines 35-38).

The proposed framework and two data reduction strategies elevate the traditional view of monitoring in Prometheus Federation from "node" to "cluster". The design of Acala is to hierarchically monitor different levels of metrics. The original Prometheus Federation scrapes per-server metrics from all member clusters to the global view cluster, where all the detailed metrics can be found. Instead, Acala keeps the detailed per-server metrics, which are neither aggregated nor deduplicated, in the member cluster. It then reports the modified metrics to the global view cluster. Using metrics aggregation, the monitoring data in the global view cluster

represent the overall member cluster status. The layer of monitoring will be "cluster status" in the global view cluster, and "node status" in each member cluster. Note that, although Acala performs metric aggregation and metric deduplication, from a macro perspective, our solution does not discard any data. The operator can still query detailed per-node metrics in the member clusters for anomaly detection and system management.

## 5 PERFORMANCE EVALUATION

To understand the performance of Acala with our proposed methods, we evaluate our framework in different aspects when we increase the number of computing nodes in one member cluster. The detailed experimental setups for the evaluation are described next.

### 5.1 Experimental Setup

For the sake of making our work as close as possible to the production environment, we implement a prototype of our framework and run it in the Grid'5000 geo-distributed testbed which is composed of eight sites located in different cities in France and more than 750 physical compute nodes pooled in homogeneous clusters [5]. We discuss the setup along the following four aspects: deployment of the experiment, performance indicators, comparison methods, and tools for collecting the data.

**Deployment of the experiment:** To support the design features described in Section 4, we utilize Python 3.10 to implement Acala-Controller and Acala-Member. We leverage Kubernetes (v1.23.5) for container orchestration to build the environment to analyze Acala in a geo-distributed cluster federation. At the same time, we use different open-source projects in Kubernetes clusters for different functions. Cilium v1.11.4 is our Container Network Interface (CNI) that provides, secures, and observes network connectivity between container workloads in Kubernetes. Kube-Prometheus-stack v34.10.0 is a collection of Kubernetes manifests, including Prometheus v2.34.0 and node-exporter v1.3.1.

We first launch one Kubernetes cluster to be our global view cluster. This cluster contains two nodes (one for the control plane and one worker node). All the nodes are running in VMs, and each VM has 4 CPU cores and 16 GiB of memory. For the member cluster, we create the VMs from 2 nodes to 31 nodes. Each set has one control plane and several worker nodes. The VMs in the member cluster consist of 2 CPU cores and 8 GiB of memory. Each cluster installs the Prometheus server, and node-exporter deploys on each node in each cluster.

We place our Acala-Controller in the global view cluster and Acala-Member in the member cluster to measure the performance. Moreover, Acala-Controller is installed on the same node as the Prometheus server, which can reduce the inter-node network traffic when the Prometheus server in the global view cluster scrapes from Pushgateway. Meanwhile, the Pushgateway is equipped in the same pod as Acala-Controller, which can make the metrics transmission in the pod. In Kubernetes, a Pod is the smallest deployable unit of computing [18]. We leverage a Kubernetes Deployment [19] for these two components so that our software will automatically restart if a component error occurs.

**Performance indicators:** The main goal of Acala is to reduce the cross-cluster network traffic in geo-distributed cluster federations. Hence we measure network traffic as our primary indicator in the experiment. A lower network traffic implies better performance. Moreover, efficiency is a pivotal point in evaluating a system. Therefore, the scrape duration and resource consumption are also the objectives we consider. If scrape duration and resource consumption are shorter and lower, the overall efficiency is better.

**Comparison methods:** In our proposed system, the data reduction strategy is a method to reduce the metrics when the global view clusters scrape from the member clusters. To evaluate the performance of metrics aggregation and metrics aggregation with deduplication, we compare them with unmodified Prometheus Federation. In addition, we examine these three methods with different scrape interval (5 s and 60 s).

**Tools for collecting the data:** After deploying the system, we start to collect the related data. All experiments are gathered for 6 minutes. There are three performance indicators for evaluating the Acala. Each indicator has its method of collection. For the cross-cluster network traffic, we use *tcpdump* to capture the network traffic. The scrape duration is based on the *time.perf_counter()* function in the Acala source code to measure the execution time of each step. We sum the execution time of Acala-Member, Acala-Controller, and the duration of Prometheus scrape from the Pushgateway to become our scrape duration. The resource usage of the Acala components, including CPU and memory, is monitored by the Kubernetes Metrics Server (v0.6.1) [21].

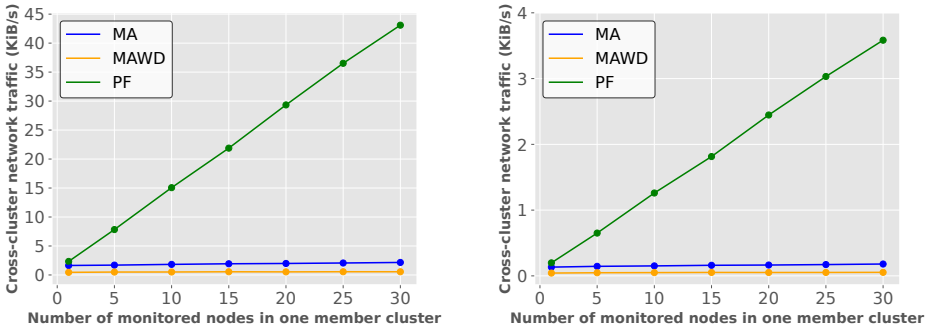
### 5.2 Experimental Results

This section shows the experimental results of three performance indicators.

*5.2.1 Cross-Cluster Network Traffic.* Figures 4 and 5 show the experimental results of cross-cluster network traffic on average and per scrape, respectively. Figures 4(a) and 5(a) present the results of the system scraping the metrics every 5 seconds whereas the outcomes of 60 second scrape interval are shown in Figures 4(b) and 5(b). To increase readability in the figures, we denote Metrics Aggregation as MA, Metrics Aggregation With Deduplication as MAWD, and Prometheus Federation as PF.

Figure 4(a) shows that metrics aggregation with deduplication significantly reduces cross-cluster network traffic, which is 0.45 KiB/s, whereas the network traffic in metrics aggregation and Prometheus Federation are 1.64 KiB/s and 2.33 KiB/s when monitoring a single worker node in the member cluster. Using metrics aggregation with deduplication in Acala and compared to Prometheus Federation, the reduction of network traffic in the single-node case is 1.88 KiB/s, which is 81% lower, and there are 30% lower when we apply the metrics aggregation as our data reduction strategy. Figure 4(b) shows the same trend that both of our proposed methods have lower network traffic than the Prometheus Federation. If the monitored nodes are set to 20, 25, and 30, the network traffic is 98%/93%, 98%/94%, and 99%/95% lower when we use the metrics aggregation with deduplication/metrics aggregation and compare to Prometheus Federation.

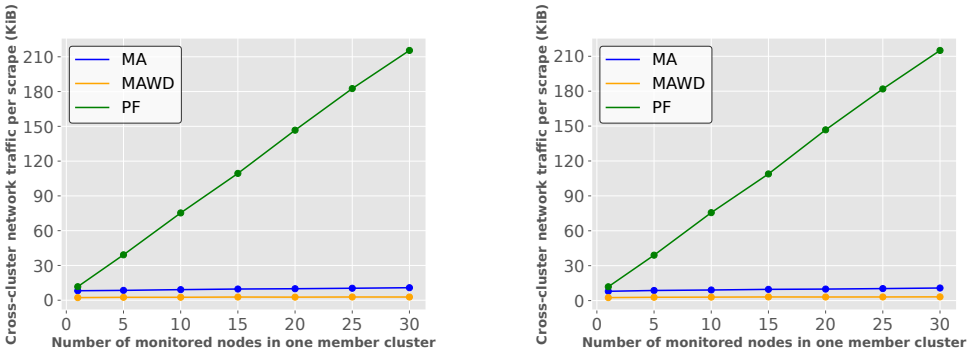
Overall, Figure 4 demonstrates that no matter how many monitored nodes are in the experiment, both of our proposed methods




(a) Scrape interval is set to 5 seconds.

(b) Scrape interval is set to 60 seconds.

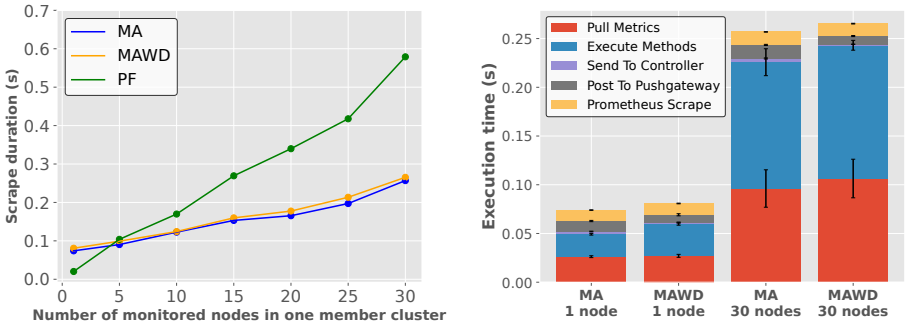
**Figure 4: Average cross-cluster network traffic with scrape interval is set to 5 seconds (a) and 60 seconds (b).**




(a) Scrape interval is set to 5 seconds.

(b) Scrape interval is set to 60 seconds.

**Figure 5: Cross-cluster network traffic per scrape with scrape interval is set to 5 seconds (a) and 60 seconds (b).**

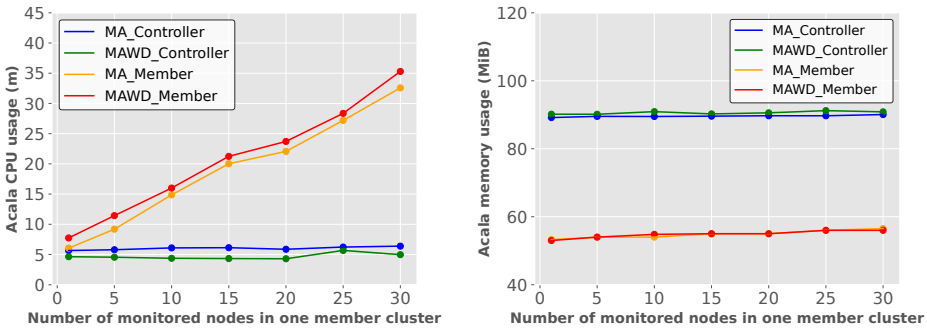



(a) Scrape duration.

(b) Execution time of each step.

**Figure 6: Scrape duration (a) and Execution time of each step (b) when scrape interval is set to 5 seconds.**

perform significantly better than Prometheus Federation. The design of Prometheus Federation will scrape the metrics from all nodes in the target member cluster to the global view cluster. Our strategy is also to scrape the metrics from all nodes, but we make this task in the member cluster, making the transmission happen in the same cluster, which can reduce the cross-cluster network traffic. Moreover, our methods aggregate the same metrics between the monitored nodes, which can decrease the volume of monitoring




(a) CPU consumption.

(b) Memory consumption.

**Figure 7: CPU (a) and Memory (b) consumption when scrape interval is set to 5 seconds.**

data to reduce cross-cluster network traffic and make the view of monitoring from node to cluster. In addition, the method of metrics aggregation with deduplication is even lower than metrics aggregation since unmodified data does not get sent multiple times. If the value of the metric is the same as the current time and the last time, metrics aggregation with deduplication will remove these metrics to save network bandwidth between clusters.

Acala collects metrics from monitored targets based on the fixed scrape interval. However, the current design does not smooth the data transmission over time as data get transferred at periodic intervals (same as Prometheus Federation). Therefore, we also want to know how much network bandwidth is used per scrape in this experiment. The results of cross-cluster network traffic per scrape are shown in Figure 5. In the case of 5 second scrape interval, we see in Figure 5(a) that the cross-cluster network traffic of Prometheus Federation experiences linear growth from 11.67 KiB (for 1 node), 39.17 KiB (for 5 nodes) to 215.48 KiB (for 30 nodes). The difference between 1 and 30 monitored nodes is 203.81 KiB, which is 1746% greater. This is because Prometheus Federation scrapes the metrics from all nodes in the member cluster. Moreover, it also appends all original labels in each metric to identify the scraped target. These strategies significantly increase cross-cluster network traffic. Figure 5(b) reflects that the results are almost the same as with 5 seconds scrape interval case in our methods of metrics aggregation and metrics aggregation with deduplication. The network traffic of both methods grows a little when the number of monitored nodes increases. When increasing the monitored nodes from 1 to 30, the network traffic of metrics aggregation with deduplication/metrics aggregation is 2.57/8.00 KiB and 3.21/10.81 KiB, respectively. The growth rates are 25% and 35%, which are lower than Prometheus Federation. Although our methods aggregate the metrics, some metrics are specific to nodes. These metrics will append to aggregated metrics, which will increase a little the cross-cluster network traffic.

These results demonstrate that our solution works well in one member cluster with various numbers of worker nodes. We expect that the cross-cluster network traffic of multi-clusters deployment will grow proportionally since our approach aggregates metrics between nodes in a cluster but not between different clusters.

*5.2.2 Scrape Duration.* We now study the time it takes to scrape metrics using Acala. For the sake of clarity, we only show the results of 5 seconds scrape interval in Figure 6. We see in Figure 6(a) that scrape duration grows with the number of worker nodes that need to be scraped. However, the growth rates of Prometheus Federation's scrape duration are greater than both of our methods. Acala starts to outperform Prometheus Federation with about 5 monitored nodes. In the case of a single node, the scrape duration of metrics aggregation and metrics aggregation with deduplication is greater than Prometheus Federation because Acala must execute additional operations compared to Prometheus Federation. In the case of 30 nodes in the member cluster, the scrape duration of Prometheus Federation is around 0.58 s, whereas the scrape duration of metrics aggregation with deduplication is 0.27 s (53% lower that Prometheus Federation). Furthermore, the metrics aggregation in the same case performs even better, up to 55% shorter than Prometheus Federation. In general, our methods perform better than Prometheus Federation when the cluster contains more nodes.

The detailed execution times of each step in the proposed approach are shown in Figure 6(b). We present two cases with 1 node and 30 nodes, and split the scrape time along the five main steps of Acala: *Pull Metrics*, *Execution Methods*, *Send To Controller*, *Post To Pushgateway*, and *Prometheus Scrape*. We can see that the total execution time of metrics aggregation with deduplication is greater than metrics aggregation in both cases. Based on the figure, we can find that the *Execution Methods* is slightly greater because metrics aggregation with deduplication will compare the last values of the metrics, which consumes more time for execution. The execution time of the 30 nodes situation is greater than 1 node. The major increases are from *Pull Metrics* and *Execution Methods*. More nodes need to be processed by the Acala-Member, which takes more time.

*5.2.3 Resources Consumption of Acala Components.* To better understand the efficiency of our system, we measure the resource usage to see how much CPU and memory are needed. Same as scrape duration experiments, we only show the results of 5 seconds scrape interval in Figure 7. The CPU usage[3] of Acala components

[3]In the Kubernetes metrics server, the unit of CPU usage is millicpu or millicores (m). For example, 100m is equivalent to 0.1 vCPU/core for cloud providers or 0.1 hyper thread on bare-metal Intel processors [33].

is depicted in Figure 7(a). We found that the CPU usage of Acala-Member grows as the number of monitored nodes increases, and metrics aggregation with deduplication is a little greater than metrics aggregation. There are two reasons for these results: one is that more nodes need to execute, and the other is because comparison consumes CPU resources. At the same time, the Acala-Controller's CPU usage of metrics aggregation with deduplication is lower than metrics aggregation because the transmission volume is smaller, which reduces the execution of functions such as decompression in Acala-Controller. Regardless of the Acala-Controller or Acala-Member, the memory consumption of both approaches is almost the same, which is around 90 MiB for Acala-Controller and 55 MiB for Acala-Member as shown in Figure 7(b).

## 6 CONCLUSION

This article presents Acala, a monitoring framework for geo-distributed cluster federations. Acala exploits two strategies called metrics aggregation and metrics deduplication for reducing the volume of monitoring data that needs to be reported to the management cluster. Acala performs more efficiently than regular Prometheus Federation because of lower cross-cluster network traffic and shorter scrape duration. Using actual deployment for experiments, we show that Acala can reduce the cross-cluster network traffic by up to 99% and the scrape duration by up to 55% compared to Prometheus Federation.

## ACKNOWLEDGMENTS

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

## REFERENCES

[1] HamidReza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. 2021. Model-based Stream Processing Auto-scaling in Geo-Distributed Environments. In *Proc. ICCCN*.
[2] Alkiviadis Aznavouridis, Konstantinos Tsakos, and Euripides G. M. Petrakis. 2022. Micro-Service Placement Policies for Cost Optimization in Kubernetes. In *Proc. ANIA*.
[3] Davaadorj Battulga, Mozhdeh Farhadi, Mulugeta Ayalew Tamiru, Li Wu, and Guillaume Pierre. 2022. LivingFog: Leveraging fog computing and LoRaWAN technologies for smart marina management (experience paper). In *Proc. ICIN*.
[4] Álvaro Brandón, María S Pérez, Jesus Montes, and Alberto Sanchez. 2018. Fmone: A flexible monitoring solution at the edge. *Wireless Communications and Mobile Computing* 2018 (2018).
[5] Franck Cappello et al. 2005. Grid'5000: A large scale and highly reconfigurable Grid experimental testbed. In *Proc. IEEE/ACM Intl Workshop on Grid Computing*.
[6] Gabriele Carcassi et al. 2020. SLATE: Monitoring Distributed Kubernetes Clusters. In *Proc. ACM PEARC*.
[7] Yu-Wei Chan, Halim Fathoni, Hao-Yi Yen, and Chao-Tung Yang. 2022. Implementation of a Cluster-Based Heterogeneous Edge Computing System for Resource Monitoring and Performance Evaluation. *IEEE Access* 10 (2022).
[8] Michael Chima Ogbuachi, Anna Reale, Péter Suskovics, and Benedek Kovács. 2020. Context-Aware Kubernetes Scheduler for Edge-native Applications on 5G. *Journal of Communications Software and Systems* 16, 1 (2020).
[9] Jaeeun Cho and Younghan Kim. 2021. A Design of Serverless Computing Service for Edge Clouds. In *Proc. ICTC*.
[10] Cloud Native Computing Foundation. 2021. Kubernetes at the Edge: Organizations are using edge technologies, but there is room to grow. https://bit.ly/3OAnuIT.
[11] Breno Costa, João Bachiega, Leonardo Rebouças Carvalho, Michel Rosa, and Aleteia Araujo. 2022. Monitoring fog computing: A review, taxonomy and open challenges. *Computer Networks* 215 (2022).
[12] Ali J. Fahs and Guillaume Pierre. 2020. Tail-Latency-Aware Fog Application Replica Placement. In *Proc. ICSOC*.
[13] Marcel Großmann and Clemens Klug. 2017. Monitoring Container Services at the Network Edge. In *Proc. ITC*.
[14] Tengfei Hu and Yannian Wang. 2021. A Kubernetes Autoscaler Based on Pod Replicas Prediction. In *Proc. ACCTCS*.
[15] Chih-Kai Huang and Shan-Hsiang Shen. 2021. Enabling Service Cache in Edge Clouds. *ACM Transactions on Internet of Things* 2, 3 (2021).
[16] Jiaming Huang, Chuming Xiao, and Weigang Wu. 2020. RLSK: A Job Scheduler for Federated Kubernetes Clusters based on Reinforcement Learning. In *Proc. IEEE IC2E*.
[17] Dongmin Kim, Hanif Muhammad, Eunsam Kim, Sumi Helal, and Choonhwa Lee. 2019. TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform. *Applied Sciences* 9, 1 (2019).
[18] Kubernetes. cited Dec 2022. Pods. https://kubernetes.io/docs/concepts/workloads/pods/.
[19] Kubernetes. cited Oct 2022. Deployments. https://reurl.cc/9pgGkx.
[20] Kubernetes SIGs. cited Oct 2022. Kubernetes in Docker. https://kind.sigs.k8s.io/.
[21] Kubernetes SIGs. cited Oct 2022. Kubernetes Metrics Server. https://reurl.cc/RrmAGG.
[22] Thomas Lin, Simona Marinova, and Alberto Leon-Garcia. 2020. Towards an End-to-End Network Slicing Framework in Multi-Region Infrastructures. In *Proc. IEEE NetSoft*.
[23] Multicluster Special Interest Group. cited Oct 2022. Kubernetes Cluster Federation. https://bit.ly/3QW2cHw.
[24] Nagios. cited Oct 2022. Nagios. http://www.nagios.org/.
[25] Javier Povedano-Molina, Jose M. Lopez-Vega, Juan M. Lopez-Soler, Antonio Corradi, and Luca Foschini. 2013. DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds. *Future Generation Computer Systems* 29, 8 (2013).
[26] Prometheus. cited Oct 2022. Federation. https://reurl.cc/9Ga2E8.
[27] Prometheus. cited Oct 2022. node-exporter. https://github.com/prometheus/node_exporter.
[28] Prometheus. cited Oct 2022. Overview. https://reurl.cc/e3A1aL.
[29] Prometheus. cited Oct 2022. pushgateway. https://github.com/prometheus/pushgateway.
[30] Behshid Shayesteh, Chunyan Fu, Amin Ebrahimzadeh, and Roch Glitho. 2022. Automated Concept Drift Handling for Fault Prediction in Edge Clouds using Reinforcement Learning. *IEEE Transactions on Network and Service Management* 19, 2 (2022).
[31] Paulo Souza Jr, Daniele Miorandi, and Guillaume Pierre. 2022. Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes. In *Proc. IEEE ICFEC*.
[32] Mulugeta Ayalew Tamiru, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. 2021. mck8s: An orchestration platform for geo-distributed multi-cluster environments. In *Proc. ICCCN*.
[33] The Kubernetes Authors. cited Oct 2022. Resource Management for Pods and Containers. https://tinyurl.com/2p9a976r.
[34] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. 2021. Machine Learning-Based Scaling Management for Kubernetes Edge Clusters. *IEEE Transactions on Network and Service Management* 18, 1 (2021).
[35] Panagiotis Trakadas, Panagiotis Karkazis, H-C Leligou, Theodore Zahariadis, Wouter Tavernier, Thomas Soenen, Steven Van Rossem, and L Miguel Contreras Murillo. 2018. Scalable monitoring for multiple virtualized infrastructures for 5G services. In *Proc. SoftNetworking*.
[36] Demetris Trihinas, George Pallis, and Marios D. Dikaiakos. 2014. JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud. In *Proc. CCGrid*.
[37] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. 2018. Fogernetes: Deployment and management of fog computing applications. In *Proc. IEEE/IFIP NOMS*.