Aggregate Monitoring for Geo-Distributed Kubernetes Cluster Federations

Chih-Kai Huang, Guillaume Pierre

Aggregate Monitoring for Geo-Distributed Kubernetes Cluster Federations

Chih-Kai Huang and Guillaume Pierre

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, an aggregate monitoring framework for geo-distributed Kubernetes cluster federations which aims to provide the management cluster with aggregated information about the entire cluster instead of individual servers. Based on actual deployment under a controlled environment in the geo-distributed Grid'5000 testbed, our evaluations show that Acala reduces the cross-cluster network traffic by up to 97% and the scrape duration by up to 55% in the single member cluster experiment. Our solution also decreases cross-cluster network traffic by 95% and memory resource consumption by 83% in multiple member cluster scenarios. A comparison of scheduling efficiency with and without data aggregation shows that aggregation has minimal effects on the system's scheduling function. These results indicate that our approach is superior to the existing solution and is suitable to handle large-scale geo-distributed Kubernetes cluster federation environments.

Index Terms—Monitoring, Geo-distributed cluster federations, Prometheus, Kubernetes, 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 [12]. 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 [10], [51]. 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 [33].

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 [33], newer designs support a range of fine-grained placement policies based on metrics such as cluster load, location, and network usage [39]. 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 [44], [46].

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 cross-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.

We propose 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 metrics 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.

This article is an extended version of an earlier conference paper that demonstrated in a federation with two clusters that aggregation and deduplication can significantly reduce the cross-cluster network traffic [23]. The extensions from the current article are as follows: (1) we extend the related work section to report in detail about the positioning of Acala compared to existing monitoring solutions designed to operate in fog computing environments; (2) we give additional details about the aggregation and deduplication algorithms, and discuss error handling in case certain components crash; (3) we expand the performance evaluation to highlight Acala's resource usage in the member and management clusters; (4) we quantify the per-server inaccuracies that are introduced by aggregation; (5) we evaluate Acala's performance when running in a large federation with up to 50 clusters with 20 servers each (1,000

- *Chih-Kai Huang and Guillaume Pierre are with Univ Rennes, Inria, CNRS, IRISA (e-mail: chih-kai.huang@irisa.fr; guillaume.pierre@irisa.fr).*

servers in total); and finally (6) we evaluate the impact of aggregation on scheduling efficiency based on a real-world dataset.

Our evaluations based on actual deployments in the geo-distributed Grid'5000 testbed [7] show that Acala reduces the volume of cross-cluster network traffic by up to 97% compared with vanilla Prometheus while reducing the necessary time to scrape metrics by up to 55% in a single member cluster experiment. At larger scales, Acala also performs well in reducing the cross-cluster network traffic by about 95%. Moreover, the resource usage of Acala components also remains acceptable in the single cluster case, and we prove that our solution can save memory resources in the larger case. Finally, we show that data aggregation has minimal impact on scheduling efficiency.

The remainder of this article is organized as follows. Section 2 discusses the motivation behind this work. We present the background and related work in Section 3. Section 4 introduces how Acala works and two data reduction strategies. Section 5 shows the evaluations of the proposed framework with strategies, and the conclusions of the paper are in Section 6.

2 Motivation

Managing geo-distributed federated clusters like a fog computing platform or a telco cloud use case is a difficult challenge [30]. Some works focus on the management problem of multiple Kubernetes clusters, such as Kubernetes Cluster Federation (KubeFed) and multi-cluster Kubernetes (mck8s). KubeFed [33] 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 therefore makes it hard to manage the workloads in a large-scale environment automatically. mck8s [39] 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 [40]. 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 1,000 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

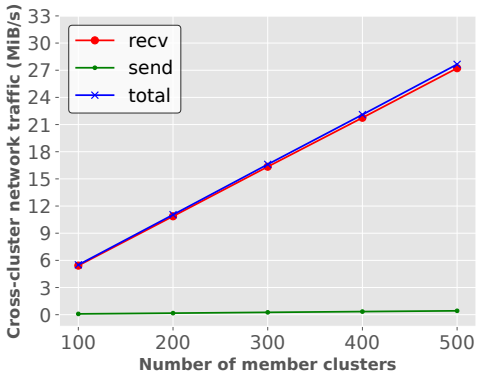


Fig. 1. Cross-cluster network traffic in the management cluster when using mck8s.

(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 [1] 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 [19], service caching [25], [26], seamless application migration [38], and supporting data stream processing [2]. 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 [27], whereas others address resources management [28], [39] and fault prediction [37]. These studies rely on a monitoring system to collect the metrics. However, they do not aim to solve the problem of monitoring itself in geo-distributed Kubernetes cluster federations.

The main purpose of geo-distributed resource monitoring is to track the resource usage of the computing nodes, especially in potentially resource-restricted and unstable environments. A number of open source and commercial monitoring tools such as DARGOS [35], Zabbix [52], PCMONS [17], JCatascopia [50], and Nagios [34] were developed to suit cloud computing requirements. However, they are not considered appropriate for geo-distributed fog computing environments [4], [16]. On the other hand, some authors present monitoring solutions and architectures designed with the specific constraints of fog computing in mind.

PyMon [21] is an extension to the lightweight open-source software Monit [32]. PyMon provides a monitoring

solution for fog environments specially designed to run on ARM-based single-board computers. However, its scalability was not evaluated, which may not suit large-scale scenarios [16].

FMonE [6] also aims to address monitoring challenges in fog environments with an independent, stand-alone solution. The authors evaluate their work using up to 78 Virtual Machines (VMs). However, such a number remains very far from the scale at which global fog platforms are expected to operate.

FogMon [20] proposes a Peer to Peer (P2P) monitoring architecture that deploys an agent in each member node called "Follower." Followers report the hardware-based metrics to the "Leader" node. Each Follower node is linked to a single Leader node and runs in a classic client-server model. To reduce the network traffic between Followers and Leaders, FogMon adopts a solution similar to Acala's deduplication, where Followers only send data with the average or variance value greater than a threshold compared to the last sent. AdaptiveMon [15] extends FogMon with two additional functions: Indicators Selection and Change Rate. Indicators Selection decreases the number of metrics, whereas Change Rate adjusts the frequency of metrics reported from Follower to Leader. However, these two solutions are not designed for cluster federation environments where nodes are not considered individually but cluster by cluster.

The most popular monitoring tool is Prometheus [46]. Since 2018, Prometheus has been accepted by the Cloud Native Computing Foundation (CNCF) as a "graduated" project, which shows its great potential in conjunction with Kubernetes [14]. At the same time, many research works use Prometheus as a basis for system monitoring [3], [8], [9], [22], [31], [49].

Prometheus provides a function called "Federation" which allows a Prometheus server to collect the metrics from other Prometheus servers [44]. 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 the user can scrape from and a *job* is a collection of *instances* with the same purpose. Prometheus Federation is used to monitor systems in numerous studies [5], [11], [18], [24], [39].

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 that are 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 metrics even if some of the metrics 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 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 metrics 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 metrics values and thereby avoids transferring unchanged values over and over again.

4 System Design

The objective of this work is to monitor computing resources in geo-distributed Kubernetes cluster federations while reducing the required cross-cluster network traffic as well as the deployment and configuration costs. In this section, we discuss the operation of Acala and introduce two data reduction strategies designed for Acala to reach our goal.

4.1 System Model

A geo-distributed Kubernetes cluster federation is a set of multiple "member" Kubernetes clusters in various locations that are considered as a single execution platform thanks to a "management" cluster which is in charge of collecting metrics data from the member clusters and deciding which application should be running in which member cluster. Each cluster consists of several computing nodes, and we assume that each node has enough resources to run the applications to provide monitoring. All computing nodes in a cluster are located in the same area. The network connects each node and cluster 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 [45], and Pushgateway [47]. The system overview is shown in Figure 2.

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

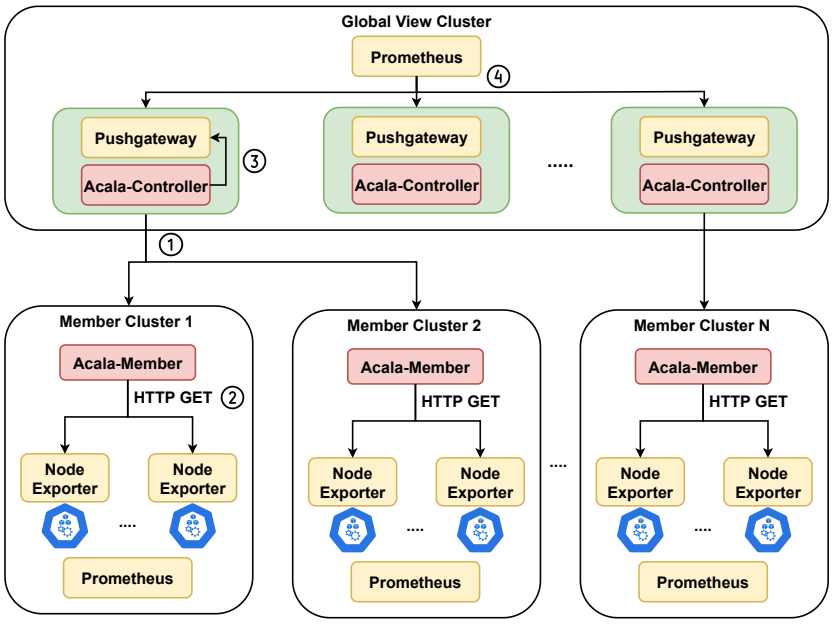


Fig. 2. Overview of Acala architecture and scrape flow.

Prometheus server in member clusters: The duty of these servers is to scrape[2] time-series data about local metrics in each member cluster and 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 monitors 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 metrics 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 the member cluster, and pushing the metrics to the Pushgateway. Acala-Controller is located in the global view cluster, and the administrators may launch additional Acala-Controller instances to accommodate the larger number of member clusters. In this case, each Acala-Controller can be configured to scrape metrics from a designated subset of member clusters. The task of Acala-Member is to pull the metrics from the Node-exporter in a single member cluster and execute proposed data reduction strategies. The data transmissions between Acala-Controller and Acala-Member are compressed using gzip. The detailed scrape steps are as follows:

1: When it is time for the Acala-Controller to scrape the metrics, the controller sends a request to the target Acala-Member.
2: After Acala-Member receives the request, Acala-Member uses the HTTP GET method to pull the metrics from the local computing nodes through the 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.
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.
4: The global-view Prometheus server periodically scrapes the metrics from the Pushgateway (at a user-defined periodicity independent from the periodicity of cross-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 periodic scrape actions. When the timer counts down to 0, the system scrapes the metrics once, and 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 cluster will be more precise in representing the actual status of the member clusters yet at the cost of additional cross-cluster network traffic. The default scrape interval is defined as 5 seconds.

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

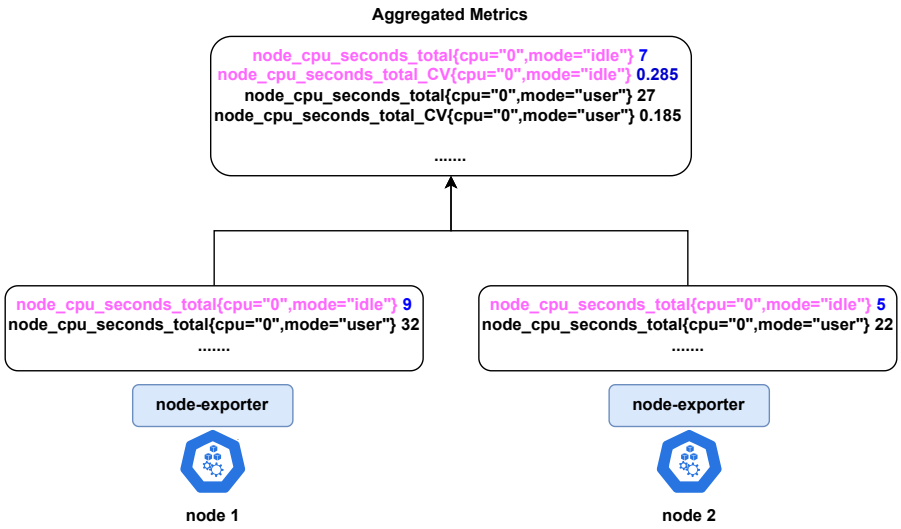


Fig. 3. An example of metrics aggregation.

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 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 metrics that are 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 $node\_cpu\_second\ \{cpu = \text{“0”}, mode = \text{“}idle\text{”}\}\ 9$, and node 2 has the same metric name with labels (fuchsia color). Metrics aggregation will thus aggregate both metric names with their labels and metric values (blue color). The resulting aggregated metric values are composed of the average of all individual values as well as the coefficient of variation between them[3]. Note that the federation schedulers only need to know about the general status of the member clusters rather than detailed per-server metrics.

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 aggregate values whose metric name and labels are identical in different servers. This method can collect and report monitoring data from each node in the target member cluster while significantly reducing cross-cluster network traffic. Moreover, metrics aggregation averages metrics values to represent the overall cluster status. This is similar to other related work [39], 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 metrics labels unchanged, the same as before aggregation. For the cluster information, we add the labels including the IP address of the 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 more cross-cluster network traffic.

4.3.2 Metrics Deduplication

Prometheus Federation blindly scrapes metrics from 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 each aggregated metric value with the most recently transferred one. If the value

3. In statistics, the coefficient of variation is a standardized measure of the dispersion of aggregated values. It is defined as the ratio between the standard deviation σ and the mean μ of the distribution: $CV = \frac{\sigma}{\mu}$.

Algorithm 1: metrics aggregation and deduplication

Output: $AMWCV$: A File of Aggregated Metrics With Coefficient of Variation

```
1  Function Aggregation():
2      M_node ← Pull Metrics from each node in the cluster
3      if AM == ∅ then
4          AM ← M_node
5      else
6          for key, value ∈ M_node do
7              if key ∈ AM then
8                  AM_key.append(value)
9              else
10                 AM_key ← value
11     return AM
12 Function Calculation(AM):
13     for key ∈ AM do
14         AverageDict_key ← MEAN(AM_key)
15         CVDict_key ← STD(AM_key)/AverageDict_key
16     return AverageDict, CVDict
17 Function
     Dedup(AM, AverageDict, CVDict, LastAM, DedupFunc):
18     for key ∈ AM do
19         if DedupFunc then
20             if LastAM then
21                 if LastAM_key != AM_key then
22                     Build AMWCV based on AverageDict
                         and CVDict (Deduplicated)
23             else
24                 Build AMWCV based on AverageDict and
                     CVDict (Full)
25         else
26             Build AMWCV based on AverageDict and
                 CVDict (Full)
27     return AMWCV
28 Function Main:
29     while true do
30         Wait for the connection
31         if Received scraping request then
32             if It is a full request then
33                 clear LastAM
34             AM ← Aggregation()
35             AverageDict, CVDict ← Calculation(AM)
36             if deduplicationfunction is enabled then
37                 AMWCV ←
                     Dedup(AM, AverageDict, CVDict, LastAM, 1)
38                 LastAM ← AM
39             else
40                 AMWCV ←
                     Dedup(AM, AverageDict, CVDict, LastAM, 0)
41             Compress AMWCV
42             send AMWCV back to Acala-Controller
43             AM, AverageDict, CVDict ← ∅
```

is identical, the deduplication strategy removes this metric from this metrics 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 metrics values in the Prometheus server without having to repeatedly transfer them from 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 checks the type of request. If it is a full request, the algorithm clears the $LastAM$ which contains the latest reported metrics values (lines 32-33). Then, Acala-Member pulls the metrics from each node through the Node-exporter. If the metric name with labels is already present in Aggregated Metrics AM, the value of matched metrics is appended to it. However, if AM does not have the same metric name with labels, the algorithm adds it as a new metric (lines 1-11). After all metrics finish aggregation, the algorithm computes the average and coefficient of variations of each metric (line 35). If deduplication is enabled, the function then checks $LastAM$. If $LastAM$ exists, it means that the computed metric values should be compared with the previous one stored in $LastAM$. If the values stored in $LastAM$ and AM are not identical, the algorithm appends the new value to the deduplicated $AMWCV$ file. If deduplication is disabled and/or $LastAM$ is empty, then the system creates a full $AMWCV$ file (lines 12-22). After returning this file, the procedure copies the aggregated metrics (AM) to $LastAM$, which are full aggregated metrics that can be compared for the next scrape (line 38). Finally, Acala-Member compresses the $AMWCV$ file using gzip, sends it back to Acala-Controller, clears the data, and waits for the subsequent scrape request (lines 41-43).

Metrics aggregation and metrics deduplication are well-established techniques. However, our work applies these methods and implementation within a geo-distributed Kubernetes cluster federation environment to build a fog computing platform where this environment has yet to be extensively explored. Moreover, 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 in the member cluster, which are neither aggregated nor deduplicated. It then reports the modified metrics to the global view cluster. Using metrics aggregation, the monitoring data in the global view cluster represents 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 metrics aggregation and metrics deduplication, from a macro perspective, our solution does not discard any data. The operator can still query detailed per-node metrics in member clusters for anomaly detection and system management.

4.4 Kubernetes Deployment and Error Handling

Acala is designed to use Kubernetes to manage its own deployment due to Kubernetes "graduated" maturity level certified by the Cloud Native Computing Foundation (CNCF) [13]. This maturity level is usually considered a stable and production-ready solution. Using this level of container orchestrator can make the design of Acala closer to the real environment and further improve the current environment. However, we argue that the concepts and algorithms proposed in this paper can be easily applied and integrated with other current or future monitoring solutions and container orchestrators.

Kubernetes offers various workload resource types such as Deployment and Job [43]. We leverage a Kubernetes Deployment for deploying the Pushgateway, Acala-Controller, and Acala-Member in their respective Kubernetes cluster. A critical advantage is that these applications will automatically restart if any component failure occurs.

The activation of deduplication requires a careful system design to ensure that metrics values are not lost. Acala-Controller uses HTTP POST to send metrics to the Pushgateway. If the Acala-Controller fails to send data to the Pushgateway, it assumes that the Pushgateway may have failed and lost previous metrics values. It therefore continues to scrape the metrics periodically from Acala-Member, but it sends full data requests. When the Pushgateway recovers, the Acala-controller can give it a fresh set of metrics values before returning to its normal behavior.

If an error occurs with the Acala-Controller and Kubernetes decides to restart it, the Acala-Controller will behave as if it was its first time launch. It then sends a first full data request to the Acala-Member before starting again to accept deduplicated metrics values.

Finally, in case Acala-Member fails, it will restart with an empty $LastAM$ and thus send the full data to the Acala-Controller before returning to its normal behavior.

5 Performance Evaluation

We evaluate Acala's performance using four separate experiments: (i) replaying Google cluster-usage traces in a member cluster to study the distribution of monitored metrics values across the cluster's servers; (ii) evaluating the cross-cluster network traffic and other performance indicators for a single member cluster with the various number of worker nodes; (iii) exploring Acala's scalability with a greater number of member clusters; and (iv) scheduling workloads across member clusters with Acala monitoring framework.

5.1 Experimental Setup

For the sake of making our work as close as possible to a production environment, we implement a prototype of our framework and run it in the Grid'5000 geo-distributed testbed [7]. We discuss the setup along the following four aspects: deployment of the experiment, performance indicators, comparison methods, and tools for collecting the data.

Deployment. 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 test environment and 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 launch one management (global view) cluster and one or more member clusters.

- The management cluster contains two nodes (one for the control plane and one worker node). Each node runs inside a VM with 4 CPU cores and 16 GiB of memory for all four experiments.
- In the first and second experiments, we create a single member cluster with a number of nodes between 2 and 31 nodes. The VMs in the member cluster have 2 CPU cores and 8 GiB of memory.
- In the third experiment, we launch up to 50 member clusters, each of which has 20 nodes (1,000 nodes in total). To mimic the limited resources of worker nodes in a fog computing environment, the VMs in the member clusters have 1 CPU core and 4 GiB of memory for worker nodes and 2 CPU cores and 8 GiB of memory for the control plane.
- In the fourth experiment, we run 5 member clusters, each of which contains 1 control plane and 6 worker nodes. Each VM in all member clusters is equipped with 2 CPU cores and 8 GiB of memory.

We deploy Acala-Controller in the global view cluster and Acala-Member in each member cluster. 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 launched in the same Pod as Acala-Controller, which enables local metrics transmission within this Pod. In Kubernetes, a Pod is the smallest deployable unit of computing [41].

Performance indicators. The first experiment aims to evaluate the dispersion of metrics values across an active member cluster. We therefore evaluate the Coefficient of Variation (CV) across values of the same metrics among the member cluster. The main goal of Acala is to reduce the cross-cluster network traffic in geo-distributed cluster federations. Hence in the second and third experiments, we measure network traffic as our primary indicator in the experiment. 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. The overall efficiency is better if scrape duration and resource consumption are shorter and lower. The objective of the fourth experiment is to understand the impact of monitoring data accuracy with the Acala framework. Therefore, we inject and schedule the workloads across the member clusters based on the resource status of workloads. Then, we check

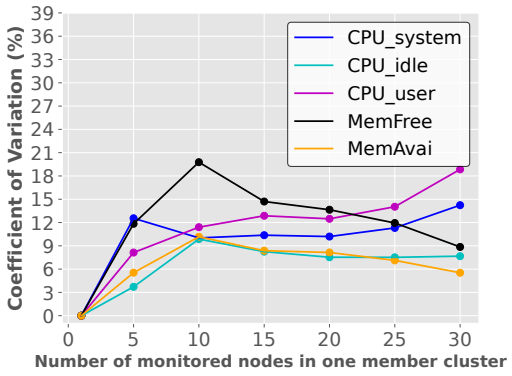


Fig. 4. Coefficient of variation when injecting workloads in a member cluster.

how many tasks can be completed as the indicator. A higher completion rate means better performance.

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. Comparing our framework to a production-ready monitoring solution as a baseline can better reflect the effectiveness of our approach in real-world scenarios. In addition, we examine these three methods with different scrape interval (5 s and 60 s).

Tools for collecting the data. The results of experiments in Section 5.3 and Section 5.4 are gathered for 6 minutes. Three performance indicators that need other tools or functions to collect related data for evaluating Acala. 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) [29].

5.2 Distribution of Resource Usage in One Member Cluster

Aggregating metrics values across a cluster obviously results in dropping detailed information about individual servers. We however argue that the resource usage of each worker node in a single Kubernetes cluster is sufficiently well balanced, so individual metrics values do not differ very much, and aggregated information is sufficient to perform accurate task scheduling. Note that, although Acala does not report per-server metrics to the management cluster, these measurements remain available in each member cluster (e.g., for troubleshooting).

To understand the distribution of resource usage in a member cluster, we replay a workload in the member cluster based on the *Google cluster-usage traces*, which is a real-world dataset from a Google cluster [36]. A Google cluster consists of multiple machines arranged in racks and interconnected through a high-bandwidth cluster network. In this cluster, there is a shared cluster management system responsible for scheduling workloads to these machines in a cluster. The Google cluster-usage traces include data about thousands of deployed applications in a cluster with several important parameters, including resource requirements (CPU, RAM), duration, and inter-arrival rates. We build a container application based on the stress-ng tool [48] to generate actual resource usage. In each experiment, we inject the workload in the member cluster for a duration of 60 minutes (1,096 tasks in total) and then wait for 30 more minutes for letting jobs complete and release the computing resources. We monitor five metrics in the cluster with a scrape interval of 5 s and compute the coefficient of variation across the cluster's servers for clusters configured with different numbers of servers. The metrics are chosen as follows:

- node_cpu_seconds_total{"mode=system"}: Time spent in kernel space of all the node's CPU cores. We use "CPU_system" in the figure.
- node_cpu_seconds_total{"mode=idle"}: Time during which each of the node's CPU cores remained idle. We use "CPU_idle" in the figure.
- node_cpu_seconds_total{"mode=user"}: Time spent in user space of all node's CPU cores. We use "CPU_user" in the figure.
- node_memory_MemFree_bytes: Free memory on the node. We use "MemFree" in the figure.
- node_memory_MemAvailable_bytes: Available[4] memory on the node. We use "MemAvai" in the figure.

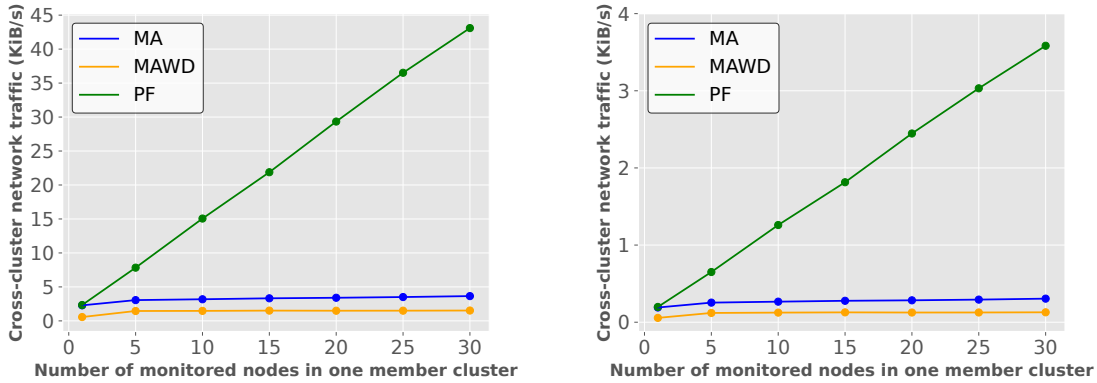
The results of this experiment are plotted in Figure 4. We can see that the CV of all five metrics in most of the cases remains between 3.7% and 19.7% except for the single-node case where no inter-node variations exist and CV is therefore equal to 0. An interesting result is that the CV of CPU resources tends to grow for cluster sizes greater than 25 servers. The reason is that the workload can be handled by fewer than 25 servers, so some servers remain idle while others are active. This experiment shows that the distribution of resource usage metrics remains relatively well-balanced in a wide variety of situations, which indicates that Kubernetes does an excellent job at balancing the load across available servers. It also demonstrates that reporting only an aggregate of these metrics values to the management cluster depicts a sufficiently accurate picture of the cluster's situation to allow efficient scheduling decisions.

5.3 Performance in a Single Member Cluster

In this experiment, we evaluate the performance improvements brought by Acala's aggregation strategies using a single member cluster with a variable number of servers.

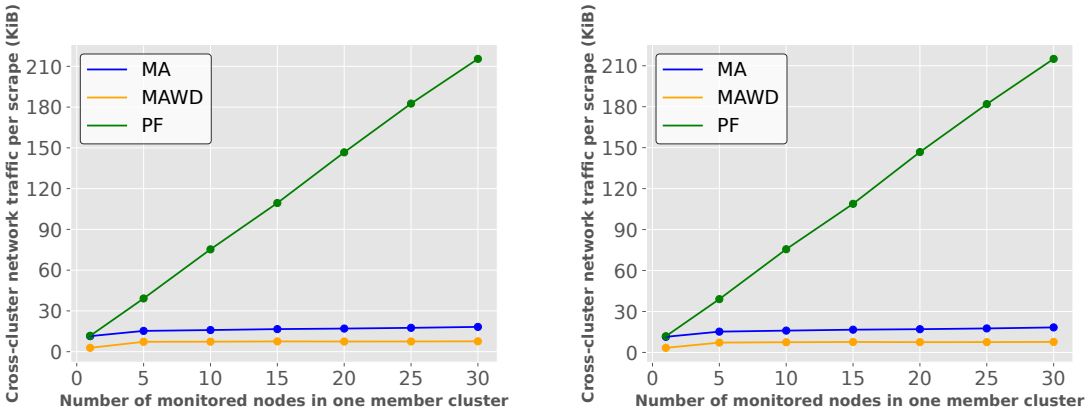
Cross-cluster network traffic. Figures 5 and 6 show the experimental results of cross-cluster network traffic on average and per scrape, respectively. Figures 5a and 6a present the results of the system scraping the metrics every 5 seconds, whereas the outcomes of 60 seconds scrape

4. Available memory includes unallocated (free) memory as well as the cached and buffered memory that are currently occupied by the system but potentially reclaimable.



(a) Scrape interval set to 5 seconds. (b) Scrape interval set to 60 seconds.

Fig. 5. Average cross-cluster network traffic with scrape interval set to 5 seconds (a) and 60 seconds (b).



(a) Scrape interval set to 5 seconds. (b) Scrape interval set to 60 seconds.

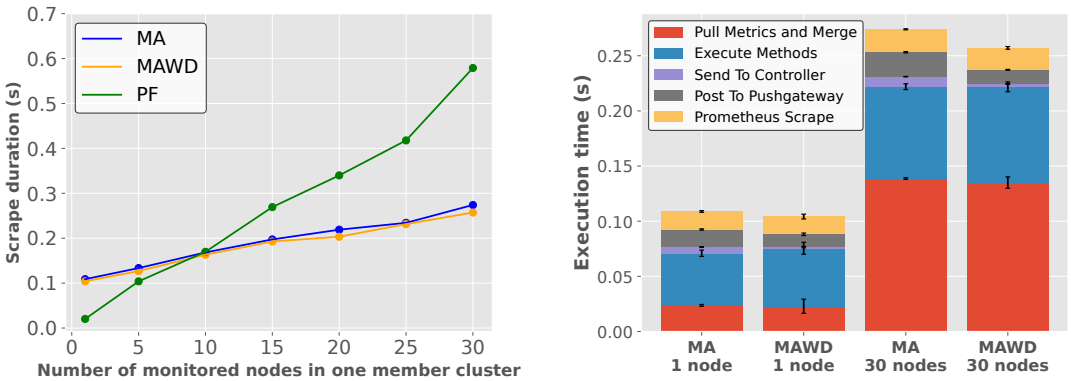
Fig. 6. Cross-cluster network traffic per scrape with scrape interval set to 5 seconds (a) and 60 seconds (b).

interval are shown in Figures 5b and 6b. To increase the readability of the figures, we denote Metrics Aggregation as MA, Metrics Aggregation With Deduplication as MAWD, and Prometheus Federation as PF.

Figure 5a shows that metrics aggregation with deduplication significantly reduces cross-cluster network traffic, which is 0.57 KiB/s, whereas the network traffic in metrics aggregation and Prometheus Federation are 2.29 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 1.76 KiB/s, which is 76% lower, and there are 2% lower when we apply the metrics aggregation as our data reduction strategy. Figure 5b 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 95%/88%, 96%/90%, and 97%/92% lower when we use the metrics aggregation with deduplication/metrics aggregation and compare to Prometheus Federation.

Overall, Figures 5 demonstrate that no matter how many monitored nodes are in the experiment, both of our proposed methods perform significantly better than Prometheus Federation. The design of Prometheus Federation will scrape the metrics of all nodes in the target member cluster to the global view cluster. Our strategy is also to scrape the metrics that are all nodes, but we make this task in the member cluster, making the transmission happen inside the 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 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 unchanged data is not 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 a fixed scrape interval. However, the current design does not smooth the data transmission over time as data get transferred at periodic interval (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 Figures 6. In the case of 5 seconds scrape interval,



(a) Scrape duration. (b) Execution time of each step.

Fig. 7. Scrape duration (a) and execution time of each step (b) when scrape interval is set to 5 seconds.

we see in Figure 6a 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 1,746% greater. This is because Prometheus Federation scrapes the metrics that are 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 6b 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 5 to 30, the network traffic of metrics aggregation with deduplication/metrics aggregation is 7.21/15.24 KiB and 7.71/18.33 KiB, respectively. The growth rates are 7% and 20%, which are lower than the Prometheus Federation. Although our methods aggregate the metrics, some metrics are specific to nodes. These metrics will be appended to aggregated metrics, which will increase cross-cluster network traffic a little.

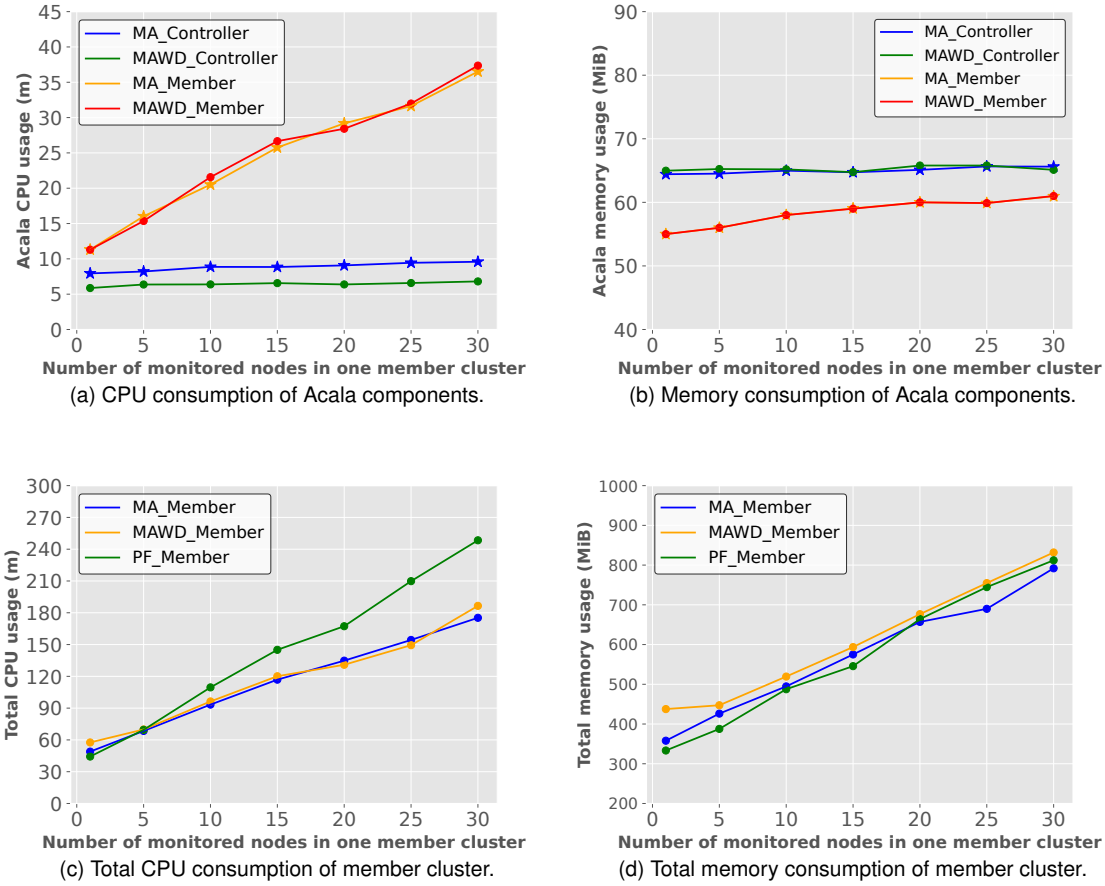
Scrape duration. We now study the time it takes to scrape metrics using Acala. For the sake of clarity, we only show the results of 5 seconds scrape interval in Figures 7. We see in Figure 7a that scrape duration grows with the number of worker nodes that need to be scraped. However, the growth rates of Prometheus Federation's scrape duration are greater than those of both of our methods. Acala starts to outperform Prometheus Federation with about 15 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 is 0.27 s (54% lower than Prometheus Federation). The metrics aggregation with deduplication 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 are shown in Figure 7b. We present two cases with 1 node and 30 nodes and split the scrape time along the five main steps of Acala: *Pull Metrics and Merge*, *Execution Methods*, *Send To Controller*, *Post To Pushgateway*, and *Prometheus Scrape*. We can see that the total execution time of metrics aggregation is greater than metrics aggregation with deduplication in both cases. Based on the figure, we can find that the *Send to Controller* and *Post To Pushgateway* are slightly greater because metrics aggregation will not compare the last metrics values, which have more metrics that need to be sent and executed. The execution time of the 30 node 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.

Resources consumption of Acala components and member cluster. 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 Figures 8. The CPU usage[5] of Acala components is depicted in Figure 8a. 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 the Acala-Controller. Regardless of the Acala-Controller or Acala-Member, the memory consumption of both approaches is almost the same, which is around 65 MiB for Acala-Controller and 55 to 60 MiB for Acala-Member as shown in Figure 8b.

The total resource usage combines Acala-Member and Prometheus server resource consumption in a member

5. 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 [42].



(a) CPU consumption of Acala components.

(b) Memory consumption of Acala components.

(c) Total CPU consumption of member cluster.

(d) Total memory consumption of member cluster.

Fig. 8. CPU (a) and memory (b) consumption of Acala components and total CPU (c) and memory usage (d) of member cluster when scrape interval is set to 5 seconds.

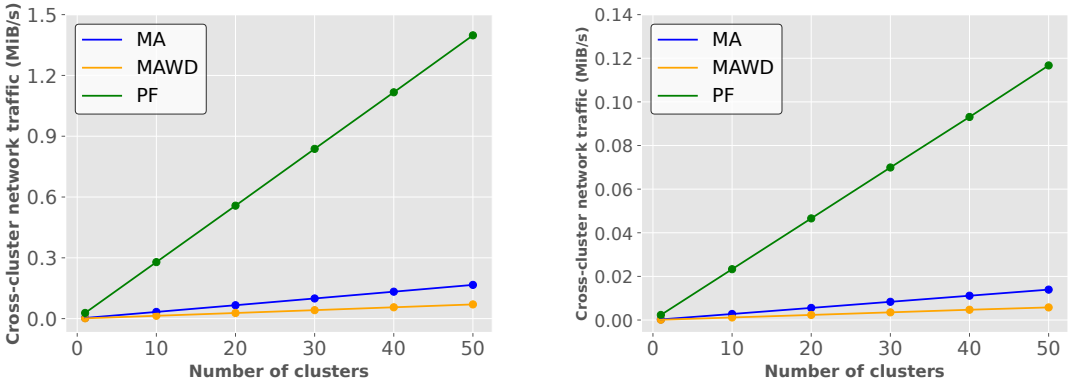
cluster. In Figure 8c, we can see that the CPU usage of Prometheus Federation is higher than both of our approaches when the number of worker nodes is greater. Although our methods require resources to run the data reduction strategies, Prometheus Federation needs to attach local labels to metrics, which also consumes resources. This is the reason why Prometheus Federation uses higher CPU resources. Figure 8d shows the results for memory. Overall, we do not see much difference in memory consumption between these three methods. For example, in the case of 30 computing nodes, the memory usage is 792 MiB, 832 MiB, and 812 MiB for metrics aggregation, metrics aggregation with deduplication, and Prometheus Federation, respectively.

5.4 Performance with Multiple Clusters

We now evaluate Acala in a larger environment where we increase the number of member clusters up to 50 clusters with 20 computing nodes each, representing up to 1,000 computing nodes. We present the same cross-cluster network traffic and resource consumption measures as in the previous section but exclude the scrape duration performance because it is a local measure within the cluster and would therefore show the same results as with a single cluster.

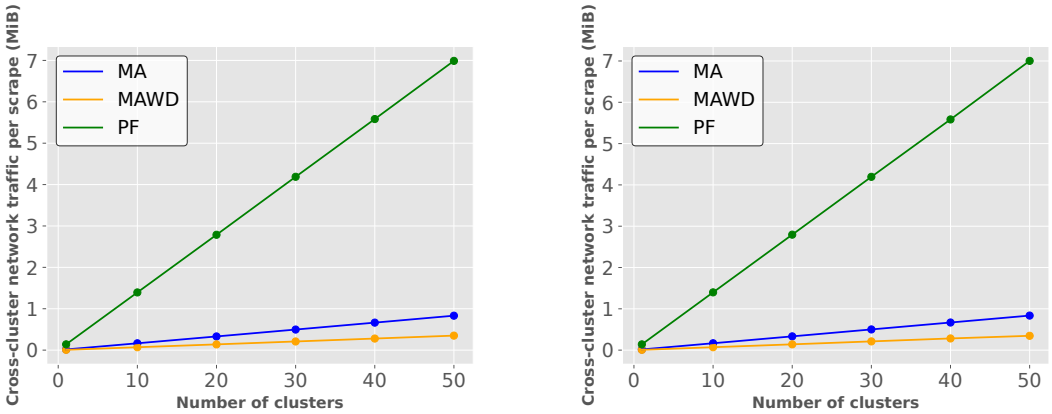
Cross-cluster network traffic. Figures 9 and 10 show the average and per-scrape network traffic when the number of clusters increases. In these two figures, subfigures (a) and (b) are the results of 5 seconds and 60 seconds scrape interval. Figure 9a presents the same trend as the previous experiment: Prometheus Federation still exhibits the greatest cross-cluster network traffic between the member and global view clusters. With 50 member clusters, Prometheus Federation uses around 1.40 MiB/s, followed by metrics aggregation (0.17 MiB/s) and metrics aggregation with deduplication (0.07 MiB/s). This means that metrics aggregation reduces the cross-cluster traffic by 88% compared with Prometheus Federation, and metrics aggregation with deduplication reduces it by 95%. Figure 9b obtains a similar reduction when scraping metrics at the 60 seconds interval, showing that Acala can effectively reduce the cross-cluster network bandwidth usage and free these precious resources to be rather used by actual user workloads.

Figures 10 show that no matter whether the scrape interval is set to 5 or 60 seconds, the cross-cluster network traffic per scrape is almost identical. We see in Figure 10a that when the scrape interval is set to 5 seconds, cross-cluster network traffic of Prometheus Federation is 4.19, 5.58, and 6.99 MiB per scrape when using 30, 40, and 50 member clusters, which represent 600, 800, and 1,000 computing nodes in total. In the same conditions, the network traffic using met-



(a) 5 seconds scrape interval. (b) 60 seconds scrape interval.

Fig. 9. Average cross-cluster network traffic in multi-cluster deployment with scrape interval set to 5 seconds (a) and 60 seconds (b).



(a) 5 seconds scrape interval. (b) 60 seconds scrape interval.

Fig. 10. Cross-cluster network traffic per scrape in multi-cluster deployment with scrape interval set to 5 seconds (a) and 60 seconds (b).

rics aggregation/metrics aggregation with deduplication is 0.50/0.21, 0.66/0.28, and 0.83/0.35 MiB. Comparing metrics aggregation and metrics aggregation with deduplication to Prometheus Federation, the reductions of network traffic are 3.69/3.98, 4.92/5.30, and 6.16/6.64 seconds which is around 90% lower than Prometheus Federation. Figure 10b shows almost identical results using a scrape interval of 60 s.

Overall, Figures 9 and Figures 10 show that our methods effectively reduce cross-cluster network traffic by an order of magnitude compared to the Prometheus Federation.

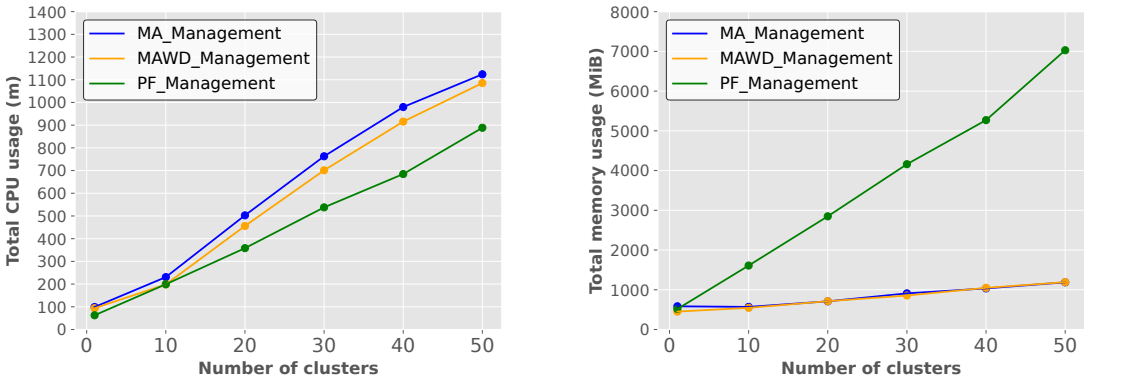
Resource usage in the management cluster. We now explore the resource usage in the management cluster with a large number of member clusters. We only show the results of total usage in the management cluster with a 5 s scrape interval. As previously discussed, the total resource usage is summed by Acala components and the Prometheus server.

Figure 11a plots the CPU resource usage as a function of the number of member clusters. All approaches see a roughly linear growth of their CPU utilization. Also, Acala's strategies require slightly more CPU than Prometheus Federation. The reason is that Acala needs to apply additional operations compared to Prometheus Federation: after scraping the metrics from member clusters, Acala-Controller then leverages HTTP POST methods to put these metrics in the Pushgateway. The metrics aggregation without deduplication is slightly higher than the metrics aggregation with deduplication because more metrics have to be scrapped.

In contrast with CPU, Acala's memory usage is much lower than that of Prometheus Federation (Figure 11b). With 50 member clusters, the memory usage in the management cluster is respectively 7,028 MiB, 1,191 MiB, and 1,187 MiB for Prometheus Federation, metrics aggregation with deduplication, and metrics aggregation. The memory reduction in both methods compared to Prometheus Federation is 5,837 MiB (83% lower) and 5,841 MiB (83% lower) for metrics aggregation with deduplication and metrics aggregation.

5.5 Impact of Aggregation on Scheduling Efficiency

To understand the impact of aggregating monitoring data with the Acala framework, we compare scheduling efficiency based on monitoring data with and without aggregation. We inject the same workloads from Google cluster-usage traces as in Section 5.2 and simulate each task execution using stress-ng, with 60 minutes of injection and 30 minutes waiting for tasks to be finished. We set the scrape interval to 5 seconds for Prometheus Federation and Acala. To schedule the workloads across these member clusters, we deploy mck8s to federate and manage clusters and



(a) Total CPU consumption of management cluster. (b) Total memory consumption of management cluster.

Fig. 11. Total CPU (a) and memory (b) usage of management cluster when scrape interval is set to 5 seconds.

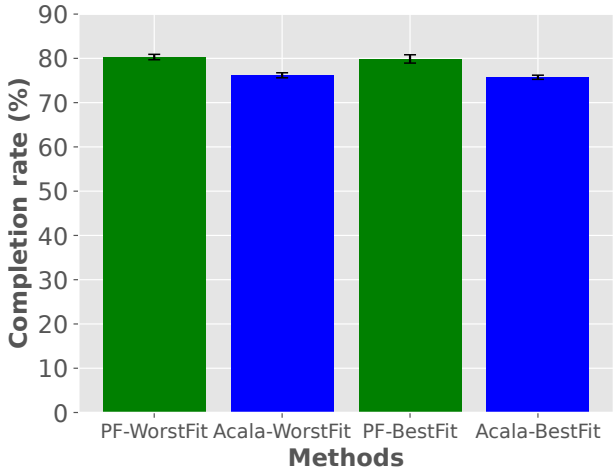


Fig. 12. Completion rate when injecting workloads in 5 member clusters.

workloads. We also modify the scheduler of mck8s to make it suitable for our experiment. The scheduler assigns each task to the cluster where the highest (worst-fit) or lowest (best-fit) number of Pods can be executed. Note that Kubernetes scheduling takes into account the sum of resource requests from running pods in each cluster node rather than their actual current resource usage. We then measure how many tasks can be completed as the metric of scheduling efficiency. We run each experiment 10 times and average the results across them.

Figure 12 depicts the completion rate when we use monitoring data for scheduling from Prometheus Federation and Acala with metrics aggregation. The two bars on the left represent the worst-fit, and the remaining are the best-fit method. The completion rate of using Prometheus Federation monitoring data is slightly higher than Acala in the worst-fit case, which are about 80.3% (Prometheus Federation) and 76.2% (Acala). On the other hand, the best-fit method selects the member cluster with the smallest available resources to enhance the resource utilization of clusters, which requires greater accuracy in monitoring data to schedule to the correct member cluster. In this difficult case, the results are similar to worst-fit, which are 79.8% (Prometheus Federation) and 75.7% (Acala). Although metrics aggregation slightly impacts the data accuracy, this scheduling experiment and previous results show that our solution can reduce cross-cluster network traffic while achieving a comparable task completion rate to the Prometheus Federation.

6 Conclusion

This article presents Acala, a monitoring framework for geo-distributed Kubernetes 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 when we increase the number of worker nodes in a member cluster. We also examine our framework in federations with large numbers of clusters, proving that the solutions suit the fog environment. Using actual deployments, we show that Acala can reduce the cross-cluster network traffic by up to 95%-97% and scrape duration by up to 55% compared to Prometheus Federation. Moreover, its resource usage remains reasonable and can even save memory resources in the management cluster. Finally, Acala does not significantly impact scheduling efficiency, which shows that reporting only aggregated metrics to the management cluster provides an accurate overview for efficient scheduling decisions.

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] Arif Ahmed, HamidReza Arkian, Davaadorj Battulga, Ali J. Fahs, Mozhdeh Farhadi, Dimitrios Giouroukis, Adrien Gougeon, Felipe Oliveira Gutierrez, Guillaume Pierre, Paulo R. Souza Jr., Mulugeta Ayalew Tamiru, and Li Wu. Fog Computing Applications: Taxonomy and Requirements. *arXiv preprint arXiv:1907.11621*, 2019.

[2] HamidReza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. Model-based Stream Processing Auto-scaling in Geo-Distributed Environments. In *Proc. ICCCN*, 2021.

[3] Alkiviadis Aznavouridis, Konstantinos Tsakos, and Euripides G. M. Petrakis. Micro-Service Placement Policies for Cost Optimization in Kubernetes. In *Proc. AINA*, 2022.

[4] Sudheer Kumar Battula, Saurabh Garg, James Montgomery, and Byeong Kang. An Efficient Resource Monitoring Service for Fog Computing Environments. *IEEE Transactions on Services Computing*, 13, 2019.

[5] Davaadorj Battulga, Mozhdeh Farhadi, Mulugeta Ayalew Tamiru, Li Wu, and Guillaume Pierre. LivingFog: Leveraging Fog Computing and LoRaWAN Technologies for Smart Marina Management (Experience Paper). In *Proc. ICIN*, 2022.

[6] Álvaro Brandón, María S Pérez, Jesus Montes, and Alberto Sanchez. FMonE: A Flexible Monitoring Solution at the Edge. *Hindawi Wireless Communications and Mobile Computing*, 2018, 2018.

[7] Franck Cappello, Eddy Caron, Michel Dayde, Frédéric Desprez, Yvon Jégou, Pascale Primet, Emmanuel Jeannot, Stéphane Lanteri, Julien Leduc, Nouredine Melab, et al. Grid'5000: A Large Scale and Highly Reconfigurable Grid Experimental Testbed. In *Proc. IEEE/ACM Intl Workshop on Grid Computing*, 2005.

[8] Gabriele Carcassi, Joe Breen, Lincoln Bryant, Robert W Gardner, Shawn Mckee, and Christopher Weaver. SLATE: Monitoring Distributed Kubernetes Clusters. In *Proc. ACM PEARC*, 2020.

[9] Yu-Wei Chan, Halim Fathoni, Hao-Yi Yen, and Chao-Tung Yang. Implementation of a Cluster-Based Heterogeneous Edge Computing System for Resource Monitoring and Performance Evaluation. *IEEE Access*, 10, 2022.

[10] Michael Chima Ogbuachi, Anna Reale, Péter Suskovics, and Benedek Kovács. Context-Aware Kubernetes Scheduler for Edge-Native Applications on 5G. *Journal of Communications Software and Systems*, 16, 2020.

[11] Jaeeun Cho and Younghan Kim. A Design of Serverless Computing Service for Edge Clouds. In *Proc. ICTC*, 2021.

[12] Cloud Native Computing Foundation. Kubernetes at the Edge: Organizations Are Using Edge Technologies, but There Is Room to Grow, cited June 2024. https://reurl.cc/aqdN1Y.

[13] Cloud Native Computing Foundation. Kubernetes Maturity Level. https://reurl.cc/eL1LrW, cited June 2024.

[14] Cloud Native Computing Foundation. Prometheus Maturity Level. https://reurl.cc/1vXeE9, cited June 2024.

[15] Vera Colombo, Alessandro Tundo, Michele Ciavotta, and Leonardo Mariani. Towards Self-Adaptive Peer-to-Peer Monitoring for Fog Environments. In *Proc. SEAMS*, 2022.

[16] Breno Costa, João Bachiega Jr, Leonardo Rebouças Carvalho, Michel Rosa, and Aleteia Araujo. Monitoring Fog Computing: A Review, Taxonomy and Open Challenges. *Elsevier Computer Networks*, 215, 2022.

[17] Shirlei Aparecida De Chaves, Rafael Brundo Uriarte, and Carlos Becker Westphall. Toward an Architecture for Monitoring Private Clouds. *IEEE Communications Magazine*, 49, 2011.

[18] Trey Dockendorf, Troy Baer, and Doug Johnson. Early Experiences with Tight Integration of Kubernetes in an HPC Environment. In *Proc. ACM PEARC*, 2022.

[19] Ali J. Fahs and Guillaume Pierre. Tail-Latency-Aware Fog Application Replica Placement. In *Proc. Springer ICSOC*, 2020.

[20] Stefano Forti, Marco Gaglianese, and Antonio Brogi. Lightweight Self-Organising Distributed Monitoring of Fog Infrastructures. *Elsevier Future Generation Computer Systems*, 114, 2021.

[21] Marcel Großmann and Clemens Klug. Monitoring Container Services at the Network Edge. In *Proc. ITC*, 2017.

[22] Tengfei Hu and Yannian Wang. A Kubernetes Autoscaler Based on Pod Replicas Prediction. In *Proc. ACCTCS*, 2021.

[23] Chih-Kai Huang and Guillaume Pierre. Acala: Aggregate Monitoring for Geo-Distributed Cluster Federations. In *Proc. ACM SAC*, 2023.

[24] Chih-Kai Huang and Guillaume Pierre. AdapPF: Self-Adaptive Scrape Interval for Monitoring in Geo-Distributed Cluster Federations. In *Proc. IEEE ISCC*, 2023.

[25] Chih-Kai Huang and Shan-Hsiang Shen. Enabling Service Cache in Edge Clouds. *ACM Transactions on Internet of Things*, 2, 2021.

[26] Chih-Kai Huang, Shan-Hsiang Shen, Chin-Ya Huang, Tai-Lin Chin, and Chung-An Shen. S-Cache: Toward an Low Latency Service Caching for Edge Clouds. In *Proc. PERSIST-IoT*, 2019.

[27] Jiaming Huang, Chuming Xiao, and Weigang Wu. RLSK: A Job Scheduler for Federated Kubernetes Clusters based on Reinforcement Learning. In *Proc. IEEE IC2E*, 2020.

[28] Dongmin Kim, Hanif Muhammad, Eunsam Kim, Sumi Helal, and Choonhwa Lee. TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform. *MDPI Applied Sciences*, 9, 2019.

[29] Kubernetes SIGs. Kubernetes Metrics Server. https://reurl.cc/RrmAGG, cited June 2024.

[30] Lars Larsson, Harald Gustafsson, Cristian Klein, and Erik Elmroth. Decentralized Kubernetes Federation Control Plane. In *Proc. IEEE/ACM UCC*, 2020.

[31] Thomas Lin, Simona Marinova, and Alberto Leon-Garcia. Towards an End-to-End Network Slicing Framework in Multi-Region Infrastructures. In *Proc. IEEE NetSoft*, 2020.

[32] Monit. Monit. https://reurl.cc/oRyx7Q, cited June 2024.

[33] Multicluster Special Interest Group. Kubernetes Cluster Federation. https://reurl.cc/6vdD5y, cited June 2024.

[34] Nagios. Nagios. https://reurl.cc/MOjkRW, cited June 2024.

[35] Javier Povedano-Molina, Jose M Lopez-Vega, Juan M Lopez-Soler, Antonio Corradi, and Luca Foschini. DARGOS: A Highly Adaptable and Scalable Monitoring Architecture for Multi-Tenant Clouds. *Elsevier Future Generation Computer Systems*, 29, 2013.

[36] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google Cluster-Usage Traces: Format + Schema. Technical report, Google Inc., 2011.

[37] Behshid Shayesteh, Chunyan Fu, Amin Ebrahimzadeh, and Roch Glitho. Automated Concept Drift Handling for Fault Prediction in Edge Clouds using Reinforcement Learning. *IEEE Transactions on Network and Service Management*, 19, 2022.

[38] Paulo Souza Jr, Daniele Miorandi, and Guillaume Pierre. Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes. In *Proc. IEEE ICFEC*, 2022.

[39] Mulugeta Ayalew Tamiru, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. mck8s: An Orchestration Platform for Geo-Distributed Multi-Cluster Environments. In *Proc. ICCCN*, 2021.

[40] The Kubernetes Authors. Kubernetes in Docker. https://reurl.cc/70d3R1, cited June 2024.

[41] The Kubernetes Authors. Pods. https://reurl.cc/QRE6X2, cited June 2024.

[42] The Kubernetes Authors. Resource Management for Pods and Containers, cited June 2024. https://reurl.cc/70d3k1.

[43] The Kubernetes Authors. Workloads. https://reurl.cc/9Gajkn, cited June 2024.

[44] The Prometheus Authors. Federation. https://reurl.cc/9Ga2E8, cited June 2024.

[45] The Prometheus Authors. Node-exporter. https://reurl.cc/GjpbAp, cited June 2024.

[46] The Prometheus Authors. Overview. https://reurl.cc/e3A1aL, cited June 2024.

[47] The Prometheus Authors. Pushgateway. https://reurl.cc/mMyvDj, cited June 2024.

[48] The stress-ng Authors. stress-ng. https://reurl.cc/p3vx6l, cited June 2024.

[49] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. Machine Learning-Based Scaling Management for Kubernetes Edge Clusters. *IEEE Transactions on Network and Service Management*, 18, 2021.

[50] Demetris Trihinas, George Pallis, and Marios D. Dikaiakos. JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud. In *Proc. IEEE/ACM CCGrid*, 2014.

[51] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. Fogernetes: Deployment and Management of Fog Computing Applications. In *Proc. IEEE/IFIP NOMS*, 2018.

[52] Zabbix. Zabbix. https://reurl.cc/GjNl43, cited June 2024.

Chih-Kai Huang is working towards the Ph.D. degree at Univ Rennes, Inria, CNRS, and IRISA. He received the M.S. degree in Computer Science and Information Engineering from the National Taiwan University of Science and Technology, Taipei, Taiwan, in 2019. His research interests cover cloud/edge/fog computing, Internet of Things, large-scale distributed systems, and SDN/NFV.

Guillaume Pierre is a Professor in Computer Science at the University of Rennes, France. Prior to this, he spent 13 years at the VU University Amsterdam. His main interests are Fog computing, Cloud computing, and all other forms of large-scale distributed systems. He is the coordinator of the FogGuru H2020 Maria-Skłodowska project, and the leader of the Magellan research team at INRIA/IRISA.