Model-based Stream Processing Auto-scaling in Geo-Distributed Environments
Hamidreza Arkian, Guillaume Pierre, Johan Tordsson, Erik Elmroth

To cite this version:

HAL Id: hal-03206689
https://inria.hal.science/hal-03206689
Submitted on 23 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Model-based Stream Processing Auto-scaling in Geo-Distributed Environments

HamidReza Arkian  
*Univ Rennes, Inria, CNRS, IRISA*  
hamidreza.arkian@irisa.fr

Johan Tordsson  
*Elastisys and Umeå University*  
johan.tordsson@elastisys.com

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

Erik Elmroth  
*Elastisys and Umeå University*  
erik.elmroth@elastisys.com

Abstract—Data stream processing is an attractive paradigm for analyzing IoT data at the edge of the Internet before transmitting processed results to a cloud. However, the relative scarcity of fog computing resources combined with the workloads’ non-stationary properties make it impossible to allocate a static set of resources for each application. We propose Gesscale, a resource auto-scaler which guarantees that a stream processing application maintains a sufficient Maximum Sustainable Throughput to process its incoming data with no undue delay, while not using more resources than strictly necessary. Gesscale derives its decisions about when to rescale and which geo-distributed resource(s) to add or remove on a performance model that gives precise predictions about the future maximum sustainable throughput after reconfiguration. We show that this auto-scaler uses 17% less resources, generates 52% fewer reconfigurations, and processes more input data than baseline auto-scalers based on threshold triggers or a simpler performance model.

Index Terms—Stream processing, auto-scaling, fog computing.

I. INTRODUCTION

The volume of data produced by Internet of Things (IoT) devices and end users is rapidly increasing. It is expected that, by 2025, 75% of all enterprise data will be produced far from the data centers [1]. These data are often generated as uninterrupted streams that must be analyzed as quickly as possible after being produced [2]. Data Stream Processing (DSP) frameworks are often used as a middleware to process such streams of data [3].

When input data are produced at the edge of the Internet, transferring them to a cloud data center where they can be processed is becoming increasingly impractical or infeasible [4]. To reduce the pressure on long-distance network links, geo-distributed platforms such as fog computing platforms are therefore being designed to extend traditional cloud systems with additional compute resources located close to the main sources of data [5]. However, managing data stream processing frameworks in geo-distributed environments remains a difficult challenge [6], [7].

A difficult and important issue faced by geo-distributed stream processing systems is that long-running IoT applications produce variable amounts of data, with significant fluctuations occurring over time [8]. Statically configuring the DSP frameworks according to their expected peak load would essentially bring these services back to a pre-cloud era where each application had to be provisioned individually with its own dedicated hardware. However, DSP frameworks were not originally designed with the necessary elasticity to dynamically adjust their resource usage to the current workload conditions [9]. As a result, any runtime DSP resource reconfiguration remains a costly operation [10].

A second issue is that stream processing applications are designed as complex workflows of data stream processing operators. Each logical operator may be replicated and distributed over multiple servers in different locations. As a consequence, stream processing applications are not monolithic entities that must be scaled up and down as a single unit, but a set of individual components that should be controlled individually.

Last but not least, fog computing networks are known to be highly heterogeneous [5]. The performance and efficiency of a fog platform is thus strongly influenced by the choice of the fog computing servers to execute any stream processing operator’s replicas, and their specific locations within the fog computing platform [11].

This paper presents Gesscale (GEo-distributed Stream au-toSCALer), an auto-scaler for stream processing applications in geo-distributed environments such as fogs. Gesscale continuously monitors the workload and performance of the running system, and dynamically adds or removes replicas to/from individual stream processing operators, to maintain a sufficient Maximum Sustainable Throughput (MST) while using no more resources than necessary. MST is a standard measure of the stream processing system’s capacity to process incoming data with no undue queuing delay [12]–[14]. Gesscale relies on an experimentally-validated performance model that gives precise estimates of the resulting performance from any poten-
tial reconfiguration [11]. This allows Gesscale to reduce the number of reconfigurations compared to a simple threshold-based auto-scaler. This is particularly important when scaling the system down, as a good performance model is the only way to accurately identify the moment when resources may be removed without violating the MST requirement.

We base our experiments on a real fog computing testbed, and the popular Apache Flink DSP engine [15]. Our evaluations show that Gesscale produces 52% fewer reconfigurations and processes more data than a baseline threshold-based auto-scaler, while using 17% fewer resources.

The remainder of paper is organized as follows. Section II presents technical background. Section III discusses related works. Then Section IV details the design of Gesscale, and Section V evaluates it. Finally, Section VI concludes.

II. BACKGROUND

A. Data stream processing in Fog platforms

Data stream processing systems were created to continuously process unbounded incoming data streams with low end-to-end latency [16]. In recent years, numerous stream processing engines (SPEs) have been proposed, including Apache Storm [17], Apache Spark [18] and Apache Flink [15]. DSP applications are designed as directed acyclic graphs of data transformations (operators) that are connected to each other by data streams, and that together form a data processing workflow. The inputs are usually composed of data stream items (tuples) that every operator consumes by applying a predefined computation and which produces new tuples that are forwarded to the next operator(s).

To guarantee a satisfactory quality of service, DSPs use a variety of techniques to parallelize the execution of their operators [19]. In most cases, replicating a stream processing operator increases its processing capacity and improves its quality of service (QoS) [20]. Therefore, many DSP performance improvement approaches focus on parallelization and elasticity strategies [3].

SPEs were initially designed for big-data analytics applications running in a cluster or a data center. Despite their distributed computation model they still lack the necessary elasticity to handle the requirements that derive from IoT/fog computing scenarios [21]. The main characteristic of fog computing environments is the geographically distribution of computing nodes, which in turn causes heterogeneous internode network latencies [22]. In such environments neither the DSP engines nor their users can easily predict the performance that would result from a reconfiguration of the DSP system using geo-distributed resources.

To handle non-stationary workloads in IoT/fog computing environments, self-adaptive DSP systems are being developed [23], [24]. They are characterized by their ability to adapt themselves to changes in their execution environment and internal dynamics to continue to achieve their objectives.

In this paper, we propose an auto-scaling system based on the MAPE (Monitor, Analyze, Plan, Execute) loop model from Autonomic Computing [25]. Using this pattern, Gesscale can locally control the adaptation of single DSP operators via a feedback controlling loop. As shown in Figure 1, Gesscale continuously monitors the workload and performance of the running application and its execution environment. When the current set of resources is insufficient to process the incoming data without delay, it selects additional nodes in the fog computing environment and rescales Apache Flink to extend to these new resources. Conversely, it releases the unnecessary resources as soon as the decreasing workload may be processed with fewer replicas.

B. Flink’s behavior in overload situations

An important aspect of any auto-scaler is to accurately detect and analyze overload situations of its controlled system. Unlike many Web-based systems that can be accurately modeled with queuing theory [26], Flink automatically slows down its operations to handle overload situations.

Flink applications are directed acyclic graphs of operators through which data are streamed and processed. When one of these operators is unable to sustain the same throughput as the others, it may compromise the entire workflow as potentially large amounts of data may need to be queued until they can be processed by the overloaded operator. As illustrated in Figure 2, when operator OPi,−1 produces data faster than its downstream operator OPi can consume, it locally buffers the undelivered tuples and starts building back pressure to slow itself down, as well as its own upstream operators. The presence of back pressure therefore indicates that at least one operator in the workflow has reached its maximum processing capacity.

Although this mechanism effectively avoids operator overload and ensures that no intermediate data is discarded for lack of queuing space, it also means that simple resource utilization metrics such as CPU utilization cannot be used to detect operator overload. Similarly, the presence of back pressure in one operator is not a sufficient indication to pinpoint operator overload, as this back pressure may either have been caused by the operator itself or propagated from downstream operators.

We therefore identify that a specific operator is experiencing overload if it does not experience back pressure itself, but all
of its upstream operators do. Note that the actual throughput of the overloaded operator is a good measure of the operator’s Maximum Sustainable Throughput (MST). Back pressure is measured as a ratio $BP \in [0; 1]$. In this work we define “high level of back pressure” as $BP > 0.5$.

C. Runtime Flink reconfigurations

Apache Flink was unfortunately not designed as an elastic platform capable of dynamically adding or removing compute resources [3]. As a consequence, when Gesscale decides to add or remove compute resources to/from a running application, it needs to stop Flink and rely of Apache Kafka to reliably buffer incoming data before entering Flink. After restarting Flink with a different resource configuration, processing may resume without any data loss. This operation however takes time during which no data can be processed.

Figure 3 depicts a Flink operator’s throughput before, during and after a resource reconfiguration. Initially the system processes incoming data as soon as it has been produced. From time $t = 650\ s$ the workload increases beyond the system’s Maximum Sustainable Throughput, which triggers a resource reconfiguration at $t = 800\ s$. We observe that the system throughput drops to zero during about 120 seconds before restarting with a significantly greater throughput. This illustrates the importance of reducing the number of resource reconfigurations as much as possible.

Note that Flink’s long reconfiguration times are not a fundamental limitation of this system. A declarative resource management capability and the corresponding scheduler are planned to be integrated in Flink-1.13, bringing concrete perspectives of significant reconfiguration time reductions in the near future [27], [28]. This new feature does not solve the problem we are addressing, but it rather makes it even more urgent to solve the auto-scaling problem.

D. Deploying Flink in geo-distributed environments

Deploying Flink in a geo-distributed fog environment requires the usage of a suitable resource orchestrator. We rely on the popular open-source Kubernetes (k8s) container orchestrator. Although it was initially designed for cluster and cloud environments, it is now being adapted to handle fog computing scenarios as well [29]–[32]. Kubernetes provides a variety of mechanisms to simplify the deployment, scaling and management of containerized applications in distributed environments by managing their complete life-cycle.

A Flink system is composed of one JobManager (in charge of managing the entire system) and a number of TaskManagers (in charge of processing data). To select in which Fog server to deploy a specific TaskManager, we maintain a map of inter-node latencies (i.e., a table of peer-to-peer network latencies between the nodes which can be obtained dynamically or provided statically), and attach the deployment requests given to Flink with Kubernetes’ nodeSelector and affinity/anti-affinity rules which constrain their deployment in the chosen server.

III. STATE OF THE ART

Numerous approaches have been proposed to control the elasticity of DSP systems in virtualized infrastructures. They differ in the environment (cluster, cloud, fog) where the DSP is deployed, the type of data that are monitored, the quality-of-service objective which is targeted, and the optimization method which is utilized [3].

Elasticity of stream processing systems has been well studied in the domain of Cloud computing where compute resources are homogeneous and connected with one another with almost negligible network latency [33]–[39]. However, working in geo-distributed environments like fog computing platforms exposes very different features where the environment is heterogeneous in terms of network latency. Heterogeneous network latency has important effects on stream processing performance which cannot be safely ignored [11].

Most works on geo-distributed stream processing auto-scaling aim to minimize the end-to-end workflow latency [40]–[44]. They use different monitored metrics and methodologies,
either at workflow-level or operator-level. A cost model is also usually considered as a side condition which describes the system cost in terms of used resources or adaptation cost [34]. Conversely, some other solutions aim to ensure sufficient throughput as their QoS objective. Threshold-based approaches compare the current throughput against a set of thresholds [33], [45]–[47]. These thresholds are usually experimentally determined using profiling methods [3]. While thresholds facilitate scaling decisions, finding appropriate threshold values for a wide range of applications and different states of a geo-distributed infrastructure can be very difficult. Conversely, in this work we aim to design an auto-scaler which does not depend on such arbitrary parameters.

Another class of solutions exploits machine learning techniques to recognize patterns from measured or profiled data [40]–[42]. Patterns are usually refined at runtime to improve precision. However, these systems must be trained for potentially very long periods of time before they can deliver accurate scaling decisions.

Finally, some auto-scalers base their decisions on a performance model to calculate the configuration that can best satisfy the defined objective. To our best knowledge, all systems in this category aim to optimize the end-to-end workflow latency [35], [36], [43], [44]. Model-based auto-scaling has the potential for delivering the most accurate decisions, depending on the prediction accuracy of the used model.

Table I summarizes and classifies these different works according to their QoS objective and the method which is used to reach this objective. As we can see, the Gesscale auto-scaler proposed in this paper is the only one which aims to optimize system throughput using a model-based approach.

IV. SYSTEM DESIGN

A. Design principles

Although a stream processing application typically consists of a complex workflow with multiple operators, the problem of auto-scaling such complex workflows can be split in a number of independent sub-problems. Because our QoS objective is to optimize throughput rather than other metrics such as end-to-end processing latency, the auto-scaling decisions can be made individually for each operator in isolation from the rest of the application. It is indeed sufficient that each operator provides sufficient throughput to process its own workload without creating undue delays which may indirectly affect other operators. We can therefore focus our attention to the auto-scaling of a single operator, with the assumption that all other operators will be auto-scaled using the same techniques.

We design the single-operator auto-scaler following the classical Monitor, Analyze, Plan, Execute (MAPE) loop architectural pattern which is the design basis for large numbers of self-adaptive systems [25].

Figure 4 shows the main architectural components of Gesscale. The core element is a MAPE-based auto-scaler which continuously monitors the performance of the running Apache Flink system, and which collaborates with a model-based performance predictor to accurately determine the required throughput and sufficient resources at operator-level. Once rescaling decisions have been made, they are implemented by a resource controller which triggers Kubernetes by sending a rescaling demand to issue the requested adaptation. The following sections respectively describe the four phases of this MAPE-based auto-scaler.

B. Monitor

Any auto-scaler bases its decisions on up-to-date information about the current system performance. In our case, we need to know the incoming data rate and current operator throughput, as well as information about the system’s computing resources in terms of their availability and network latencies with each other.

The system starts monitoring different metrics as soon as Apache Flink starts to run an application. In the initialization step, the system fetches general information such as JobID, all operators IDs, and the parallelism level of every operator. Then, it periodically monitors the current throughput and back pressure level of every operator at a 1-minute granularity. We found this periodicity implements a good trade-off between reactivity and stability of the system.

C. Analyze

The Analyze phase is in charge of determining when a scale-up or a scale-down decision is necessary, and of automatically calibrating the parameters of the operator performance model.
The Plan phase is in charge of identifying the new resource configuration which should be used to maintain the QoS objective. In a fog computing platform, the heterogeneous network performance between nodes implies that the choice of specific nodes to execute an operator influences the resulting system performance. Such decisions are taken with the help of the performance model as well as the resource controller.

1) Scaling up: Scale-up planning is presented as Algorithm 3. When a stream processing operator experiences overload as detected by the Analyze phase, we know that at least one additional resource is necessary to increase its capacity. Scale-up analysis is presented as Algorithm 1.

2) Deciding when to scale down: An operator should be scaled down when it under-utilizes its resources and it would be able to sustain the current data throughput with fewer resources. However, there is no simple metric which unambiguously indicates this condition. In particular, low CPU utilization is a poor predictor. Instead, we use the performance model to determine the MST that the operator would have if it used fewer resources than the current configuration. If this MST is greater or equal than the current operator’s throughput, then at least one resource unit can be removed without violating the QoS objective. Note that this method also allows us to determine how many resource units may be safely removed (which may be more than one in case the traffic intensity is decreasing quickly). Scale-down analysis is presented as Algorithm 2.
to resolve the situation. We can therefore only increase the number of resource units by one, and observe whether this is sufficient to reduce the operator’s overload situation. In case the problem is not solved yet, then we can repeat the operation and add other resource units one by one until the operator reaches the required throughput.

During the scale-up operations, the performance model is useful only to select which resource should be added in case more than one is currently available. In particular, in case the cost of new resource units is taken into account in the QoS objectives, the performance model can deliver accurate estimates of the future throughput with the chosen resource, which allows the system to decide whether the cost/benefit ratio is sufficient to trigger this reconfiguration.

2) Scaling down: Scale-down planning is presented as Algorithm 4. When scaling down, the goal of the auto-scaler is to remove as many resource units as possible without reducing the MST below the current system throughput value. The performance model delivers these estimates so the auto-scaler can obtain accurate information about the performance consequences of executing the operator with fewer resources.

Note that, in case of a large drop in the throughput demand, Gesscale may remove more than one resource at a time if the remaining ones are sufficient to handle the current workload. In contrast, any threshold-based auto-scaler would have to issue multiple scale-down operations one by one, thereby producing a greater number of system reconfigurations.

E. Execute

The Execute phase is in charge of executing the resource reconfiguration decisions taken by the Plan phase. It must address two challenges: first it needs to obtain an up-to-date list of available resources; this list may be fetched from Kubernetes. The resource controller then sorts the available resources based on the location of the operator i that needs to be rescaled and according to their network latencies to the existing resources. When scaling up we want to choose the resource with lowest network latency to the other resources, whereas when scaling down we want to choose the resource with greatest network latency to the other resources.

Second, the auto-scaler needs to trigger a change in the resource configuration used by Apache Flink. Unfortunately, Flink is currently not capable of dynamically integrating additional resources nor of removing some of its resources seamlessly. We therefore need to stop the Flink system and restart it with a different configuration.

The auto-scaler first stops Flink’s running job with a savepoint which checkpoints the current system state. The resource controller triggers kubectl to stop all TaskManagers by rescaling the deployment to zero replica. It then rescales again the TaskManagers deployment based on the newly requested scale and choice of resources. Finally the auto-scaler reconfigures the Flink application execution model and restarts Flink’s job from the savepoint with a different set of resources.

This operation maintains the workflow processing correctness (i.e., “it does not lose data”), but it temporarily interrupts the processing workflows. This means that resource reconstructions are expensive operations which should be issued as rarely as possible. As previously discussed, note that future versions of Flink are expected to have this capability which should significantly reduce the resource reconfiguration times. We show in the next section the importance of basing the auto-scaling decisions on an accurate performance model.

V. Evaluation

A. Experimental Setup

We conduct evaluations using a cluster of ten RaspberryPi4 single-board computers equipped with 2GB of RAM each, quad-core ARMv7 processor, and Raspbian GNU/Linux v10. This type of machines is frequently utilized to prototype fog computing systems [31], [49], [50]. The cluster is powerful enough as a testbed and it is also horizontally and vertically scalable.

We use Kubernetes across this cluster to deploy variable numbers of Flink TaskManagers as well as other required
tools. One node in the cluster acts as the Master node where we deploy Kubernetes management services as well as Flink client and the auto-scaler. A second node is used to run the MinIO data storage service and the Prometheus and Grafana monitoring tools. The remaining nodes act as separate servers of the fog computing platform.

We emulate heterogeneous network latencies between 0 and 300 ms across the fog nodes using the Linux tc (traffic control) command.

a) Flink Integration: We use Apache Flink 1.12.0. As shown in Figure 5, we deploy Flink’s JobManager and TaskManagers in separate nodes of the cluster. Without loss of generality, we configure Flink to execute a single stream processing operator in each TaskSlot, and each TaskManager is configured with a single TaskSlot.

b) Application and workload: We evaluate Gesscale using a real-world non-stationary workload derived from a dataset with records of four years of taxi operations in New York City [51]. Each record in the dataset represents one taxi trip including the date, time, and coordinate of each pickup or drop-off as well as the driver, taxi, and ride IDs. We used the first two days of May 2013, and cleaned up the data by removing duplicate and invalid entries. The resulting dataset includes roughly one million taxi rides.

We process this data set with a Data Producer and a Data Consumer. The Data Producer program (which is located out of the RPI cluster) generates a data stream of taxi ride records which are read from the dataset file and then writes them into an Apache Kafka broker. The Data Producer serves events according to their timestamps, with an adjustable speed factor. We use SpeedFactor=10 in our experiments which means that 10 s of real-world events are replayed in 1 s. To generate sufficient workload without reducing the experiment duration too much, we inject three identical records in the Flink system for each record in the dataset.

The Data Consumer, implemented as an Apache Flink operator, extracts the pickup location (latitude, longitude), calculates the Euclidean distance between this pickup location and three major tourist hotspots in New York City, and returns a record which contains the original trip information extended with the name and distance of the closest attraction.

c) Auto-scaling baselines: We evaluate the effectiveness of Gesscale and compare its performance to three other baseline algorithms which resemble systems proposed in the state of the art.

- THR-NLUnaware is a simple threshold-based auto-scaler. The scale-up threshold is defined as a CPU utilization greater than 90%, and the scale-down threshold as a CPU utilization lower than 50%. Every time one threshold is reached, THR-NLUnaware adds or removes one resource to/from the system. THR-NLUnaware is latency-unaware so it chooses resources randomly out of an unsorted list.
- THR-NLAware is another threshold-based auto-scaler which uses the same thresholds as THR-NLUnaware. In contrast, THR-NLAware is aware of the network latencies between available resources. Hence, upon a scale-up operation it chooses the available resource with lowest latency with the other nodes, and upon scale-down it removes the resource with greatest latency to the other nodes.
- MDL-NLUnaware is a simplified version of Gesscale which relies on the latency-unaware performance model presented in Equation 2. MDL-NLUnaware therefore chooses resources to be added to or removed from the system out of an unsorted list of available resources.

d) Evaluation metrics: We evaluate Gesscale and compare its performance to the three aforementioned baseline algorithms using a variety of metrics which highlight their respective performance and costs.

Accuracy is a standard metric proposed by the SPEC RG Cloud Working Group which computes the difference between the parallelism levels chosen by the evaluated algorithms and that of an “ideal” auto-scaler [52]. Figure 6 shows the behavior of our reference “ideal” autocalcer: it instantly adjusts the number of resources assigned to the considered stream processing operator to the incoming workload. As a consequence, the processed number of records/s and the number of replicas closely follow the incoming ride events’ workload. Note that this auto-scaler cannot be implemented in practice because it assumes that any change of configuration is applied immediately with no period of unavailability. According to SPEC: “The accuracy metric is divided in two sub-metrics: the under-provisioning accuracy
metric accuracy\textsubscript{U} is calculated as the sum of areas ($\Sigma_U$) where the resource demand exceeds the supply normalized by the duration of the measurement period $T$. Accordingly, the over-provisioning accuracy metric accuracy\textsubscript{O} bases on the sum of areas ($\Sigma_O$) where the resource supply exceeds the demand:\n
\[
\begin{align*}
\text{accuracy}\textsubscript{U} &= \frac{\Sigma_U}{T} \\
\text{accuracy}\textsubscript{O} &= \frac{\Sigma_O}{T}
\end{align*}
\]

The lower these two metrics are, the closer the evaluated auto-scaling algorithm is from the “ideal” strategy.

**Provisioning Timeshare:** According to SPEC: “the two accuracy metrics allow no reasoning whether the average amount of under-over-provisioned resources results from a few big deviations between demand and supply or if it is rather caused by a constant small deviation. The Provisioning timeshare metrics timeshare\textsubscript{U} and timeshare\textsubscript{O} are standard SPEC metrics computed by summing up the total amount of time spent in an under-(\Sigma_A) or over-provisioned (\Sigma_B) state normalized by the duration of the measurement period. Thus, they measure the overall timeshare spent in under- or over-provisioned states:\n
\[
\begin{align*}
timeshare\textsubscript{U} &= \frac{\Sigma_A}{T} \\
timeshare\textsubscript{O} &= \frac{\Sigma_B}{T}
\end{align*}
\]

**Excess computation time:** An important characteristic of data stream processing is that every input record will be processed eventually, assuming that the system has sufficient data buffering capacity. However, records may have to be buffered for extended durations in case the stream processing system was underprovisioned and/or it spent too much time being reconfigured. The Excess computation time measures this effect by evaluating the necessary amount of time to finished processing the remaining buffered records after the incoming workload dropped to zero at the end of the trace:

\[
\text{Excess time} = \frac{P_t - I_t}{T}
\]

where $I_t$ is the ideal execution time, and $P_t$ is actual execution time. The resulting value is also normalized by the duration of the measurement period $T$ and the lower the excess reconfiguration time, the better.

**Number of reconfigurations:** considering the cost of any resource reconfiguration, a good auto-scaler should aim to reconfigure only when this is absolutely necessary. The lower the number of reconfigurations, the smaller the amount of time during which the system cannot process incoming data.

**Cost** evaluates the amount of used resources. We use a very simple cost model where each replica is charged one cost unit per minute of execution.

\[\text{Cost} = \sum \text{replicas} \times \text{cost per minute}\]

**B. Auto-scaling effectiveness**

Figure 7 compares the behavior of the four auto-scaling algorithms. For each algorithm, the first graph depicts the incoming workload as well as the system’s processing throughput. The throughput drops are caused by reconfiguration operations. The second graph shows the number of replicas chosen by the algorithms, and the third graph shows the provisioning accuracy compared to the “ideal” auto-scaling strategy (zero values in this chart represent perfect provisioning accuracy).

We observe two periods at time $t = 30$ and $t = 180$ where the workload strongly increases before starting to oscillate. All auto-scalers react by gradually increasing the number of replicas. We can however see that the latency-aware auto-scalers eventually create fewer replicas than their latency-unaware counterparts. This is because they choose the resources that are going to provide the greatest performance gains, whereas the latency-unaware auto-scalers select resources randomly. Also, the latency-aware algorithms generate fewer reconfigurations during these phases, which results in smaller amount of incoming data being buffered during reconfiguration, and eventually lower amounts of excess computation time.

We also observe periods of workload decrease: small decreases at time $t = 100$ and $t = 220$, and much stronger decreases at $t = 120$ and $t = 270$. Here the main difference is between the threshold-based and the model-based algorithms. Threshold-based algorithms have no way to identify the new correct number of replicas so they remove replicas one by one. Conversely, the performance model gives precise indications about the necessary number of replicas, so the model-based algorithms may remove more than one replica at a time, thereby reducing the number of reconfigurations and reducing the excess computation time.

We also observe that the model-based auto-scalers identify the time when they should remove replicas, whereas threshold-based auto-scalers often trigger their reconfiguration too early, too late, or at a time when no reconfiguration is necessary. This is particularly visible in THR-NL-aware at time $t = 130$ where a decision to scale down was inaccurate and had to be
immediately compensated by two scale-up decisions. This is a fundamental difficulty faced by all threshold-based algorithms, where choosing the best set of thresholds is extremely difficult.

In short, latency-aware strategies are better during scale-up periods whereas model-based ones are better during scale-down periods. Gesscale combines latency-awareness and being model-based, and is clearly the winner of this comparison.

Table III reports the evaluation metrics for the same set of experiments. Gesscale outperforms the baseline algorithms according to all metrics except one. For instance, it generates 37% fewer reconfigurations than THR-NLAware and 52% fewer than THR-NLUnaware. Its provisioning accuracy is also 38% better in average (for both over-/under-provisioning situations), and consequently its resource usage cost is 16% lower than THR-NLAware. Eventually, although it does not manage to process all incoming data in real-time, its excess time is still 63.5% lower than THR-NLAware.

VI. CONCLUSION

Auto-scaling is an important feature for stream processing engines in geo-distributed infrastructures such as fog computing platforms. We presented Gesscale, an auto-scaler designed to allow Apache Flink to maintain a sufficient Maximum Sustainable Throughput while using no more resources than strictly necessary. Gesscale bases its decisions on a performance model which allows it to correctly anticipate the consequences of any potential reconfiguration action. Our evaluations show that Gesscale produces 52% fewer reconfigurations and processes more data than the baseline THR-NLUnaware auto-scaler, while using 17% fewer resources.
Table III: Evaluation Metrics of the Different Auto-scaling Algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>accuracy</th>
<th>accuracy</th>
<th>timeshare</th>
<th>timeshare</th>
<th>excessTime</th>
<th>#reconf.</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>THR-NLUnaware</td>
<td>1.636</td>
<td>0.742</td>
<td>51.11</td>
<td>30.28</td>
<td>0.163</td>
<td>25</td>
<td>1199.5</td>
</tr>
<tr>
<td>THR-NLAware</td>
<td>1.517</td>
<td>0.640</td>
<td>46.67</td>
<td>22.99</td>
<td>0.115</td>
<td>19</td>
<td>1193.75</td>
</tr>
<tr>
<td>MDL-NLUnaware</td>
<td>1.713</td>
<td>0.622</td>
<td>60.42</td>
<td>25.00</td>
<td>0.126</td>
<td>16</td>
<td>1270.5</td>
</tr>
<tr>
<td>Gescale</td>
<td>0.838</td>
<td>0.499</td>
<td>49.31</td>
<td>21.74</td>
<td>0.042</td>
<td>12</td>
<td>999.5</td>
</tr>
</tbody>
</table>

References