A Multi-Metric Adaptive Stream Processing System

Daniel Wladdimiro, Luciana Arantes, Pierre Sens, Nicolas Hidalgo

A Multi-Metric Adaptive Stream Processing System

Daniel Wladdimiro
Sorbonne University, Inria, CNRS, LIP6
Paris, France
daniel.wladdimiro@lip6.fr

Luciana Arantes and Pierre Sens
Sorbonne University, Inria, CNRS, LIP6
Paris, France
luciana.arantes@lip6.fr
pierre.sens@lip6.fr

Nicolas Hidalgo
Universidad Diego Portales
Santiago, Chile
nicolas.hidalgoc@mail.udp.cl

***Abstract*—Stream processing systems (SPS) have to deal with highly dynamic scenarios where its adaptation is mandatory in order to accomplish realistic applications requirements. In this work, we propose a new adaptive SPS for real-time processing that, based on input data rate variation, dynamically adapts the number of active operator replicas. Our SPS extends Storm by pre-allocating, for each operator, a set of inactive replicas which are activated (or deactivated) when necessary without the Storm reconfiguration cost. We exploit the MAPE model and define a new metric that aggregates the value of multiple metrics to dynamically changes the number of replicas of an operator. We deploy our SPS over Google Cloud Platform and results confirm that our metric can tolerate highly dynamic conditions, improving resource usage while preserving high throughput and low latency.**

***Keywords*—Stream processing systems, Adaptive SPS, Multiple metrics, Storm, Cloud.**

I. Introduction

Nowadays, applications in Cloud Computing, social networks, sensor networks, or IoT (Internet of Things) produce a large volume of data that require to be analyzed in real-time [1]. Stream Processing Systems (SPS) meet this demand offering an interface to process data 'on the fly', providing timely answers to the users. They follow a processing model based on Direct Acyclic Graph (DAG) model [2], where vertices correspond to processing operators and the edges correspond to dataflow. Operators are typically lightweight tasks such as filtering, counting, merging, etc. They can be replicated to increase processing parallelism and, therefore, the system's throughput. In runtime, the DAG is mapped to physical machines or VMs, where the operator's replicas are assigned to threads. Many SPSs such as Heron [3] or Storm [4] have been proposed in the literature in the last years.

Even if those frameworks are efficient to process dataflows, they are subject to dynamic conditions where traffic can suddenly change in terms of the number of events generated per second or distribution. Such a variation creates challenging scenarios because the SPS processing model has not been conceived to adapt itself at runtime. Consequently, it may happen that processing resources are either overestimated or underestimate by the SPS. In the former, resources might be wasted while the latter is even worst because relevant data may be lost or processing latency may increase, compromising the capacity of the SPS to provide timely responses.

978-1-6654-9550-9/21/$31.00 ©2021 IEEE

One solution to tackle the above problem is to dynamically increase the number of allocated resources, either physical [5] or logical [6], whenever the processing demand of one or more operators increases. The drawback of this strategy is the waste of system resources if the workload decreases, i.e., the system presents an overestimated amount of resources. Another existing approach is the elastic one, where the system increases or decreases the number of required resources in runtime. To this end, some SPS dynamically modify the number of replicas of the threads based on some system metrics, such as throughput [7], latency [8], or CPU usage [9]. However, they often consider just one metric which does not necessarily characterizes the real behavior of the application. For instance, the CPU utilization does not take into account the behavior of the traffic, being thus very sensitive to workload variation. On the other hand, latency does not reflect the individual load of each operator, making difficult to detect congested operators. Finally, the throughput, although it analyzes the traffic of the application, does not consider the workload.

In order to cope with the impact of the input rate stream fluctuations in operator congestion, and, consequently, operator workload variation, this work proposes an adaptive SPS that, according to a new metric that aggregates the current value of multiple metrics, dynamically changes the number of replicas of an operator. Our solution uses the Monitor, Analyze, Plan, Execute (MAPE) model [10], often applied in autonomic computing to control a system.

Our SPS implementation is an extension of Storm, rendering it adaptive. We point out that Storm supports on-demand resource scaling by unassigning all the operators from the allocated resources and then reassigning them to the new set of resources. However, such a mechanism is very costly since it interrupts the streaming computation during the reconfiguration. Contrarily, our SPS assigns, for each operator, a set of replicas, which can be either in an active or inactive state and are deployed at initialization by the Storm scheduler. Inactive replicas do not consume CPU resources but can be dynamically activated when the system detects the need for increasing the resources for the operator in question. Following the MAPE model, our SPS automatically increases or decreases the number of active operator's replicas based on a new defined metric that adapts itself to scenario fluctuations. Note that although the current implementation of our SPS was developed for applications with stateless operators, it can be

easily extended to support stateful ones.

The current contributions of this work are:

- The definition of a new metric for adaptive SPS able to improve the system response to highly dynamic scenarios.
- The implementation of an adaptive version of Storm that avoids reconfiguration downtime and uses a newly proposed metric that renders Storm adaptive.
- An evaluation of our SPS over Google Cloud Platform (GCP) aiming at validating the effectiveness of our SPS on a real cloud environment.

The rest of the document is organized as follows. Background concepts are presented in the Section II. Section III presents our adaptive SPS proposal. Section IV presents our experiments and results conducted on top of GCP with the extended version of Storm. Section V discusses existing works from the literature related to adaptive SPS. Finally, Section VI concludes and presents our future work.

II. Stream processing systems

Unlike the traditional approach of processing large amounts of data over long periods of time, SPS are designed to process high volumes of data in real-time [11]. A DAG defines the processing flow of the SPS where each vertex represents an operator, and unidirectional edges between two nodes represent the dataflow. An operator is usually a lightweight task (e.g., filtering, counting, etc.), but there also exist heavy-weight operators (e.g., classifiers based on Machine Learning models). Based on the DAG, an operator receives one or more dataflow, processes them, and sends the processed data over its output DAG edges. Furthermore, an operator can have several replicas, and each replica of the operator is associated with a thread. A data source provides the input raw data stream to be processed by the operators over the DAG. Raw data is homogeneous, composed of key-value tuples.

The data flow should be distributed to each of the replicas, and there are different approaches to do it [12]. For example, in the Shuffle Grouping, tuples are sent randomly to each replica, while in the Field Grouping, the tuple's key determines which replica will receive it. The drawback of these approaches is the potential lack of load balance. To cope with this problem, existing SPS propose other approaches such as hash-based data partition [13], partial-key based [14] or executor-centric [15] solutions.

Figure 1 shows an example of a SPS logical design (DAG), composed of the input data source, four operators, and five edges. The source is responsible for sending the raw data from the environment to the first operator O_1. After processing the data, O_1 should send its output stream to both O_2 and O_3. Thus, there are two options: partition or duplication. After processing their input data, O_2 and O_3 send their output to O_4, which terminates the workflow since it does not have an adjacent out-vertex.

The DAG must be assigned to a physical environment for deployment and subsequent execution. In addition, a scheduling algorithm is in charge of mapping operators to each of the physical resources. For instance, Storm maps each of the operators randomly to an available resource. This type of algorithm can induce load imbalance when physical resources are heterogeneous [16] or the application has complex tasks [17].

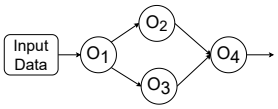


Fig. 1: The SPS logical architecture.

A. Storm

Storm [4] is an SPS framework implemented in Java. It enables the processing of unbounded dataflows following a DAG, denoted *topology*, which represents a Storm application.

There are three types of components in a topology: *Streams*, *Spouts*, and *Bolts*. *Streams* or dataflow are shared among operators. They are composed of key-value tuples. *Spouts* are responsible for capturing the input data of the topology from external sources. They structure the information to send through one or more *Streams* to the operators of the topology. *Bolts* are the operators, and similarly to *Spouts*, *Bolts* send the processed tuples through one or more *Streams*. At runtime, operators of the topology are executed by several threads called executors, which are instances of the operators.

The architecture is composed of Storm and Zookeeper clusters. The Storm cluster contains a master node, called Nimbus, and Supervisor nodes. The latter provides a fixed number of processes, called workers, that run executors. The Nimbus is responsible for distributing the application code across the cluster, scheduling executors to available workers, monitoring the state of nodes, and detecting failures. Zookeeper provides a distributed coordination service enabling communication among Storm cluster nodes, load balance, and fault tolerance.

Storm does not support runtime adaptation. On the other hand, it provides a reconfiguration command that allows the administrator to redefine the number of resources of a given topology. However, such a reconfiguration requires stopping the processing system and redeploying it, increasing processing latency and message loss. Furthermore, messages arriving during the reconfiguration will not be processed.

III. Our SPS Proposal

A. Adaptive Storm

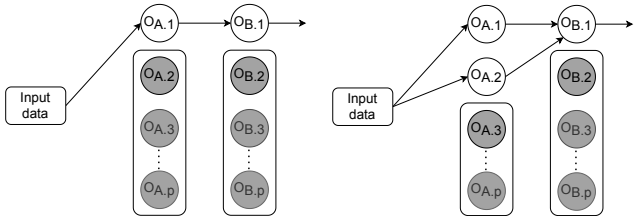
Contrarily to Storm, our adaptive version of Storm allows to dynamically increase/decrease operator replicas of the topology without the need to stop the system. This is accomplished by exploiting a pool of pre-allocated replicas as well as the MAPE model proposed by [10], which is integrated into our adaptive Storm.

MAPE model is a control loop that enables systems to adapt themselves. It consists of 4 steps: *Monitoring*, *Analysing*, *Planning*, and *Execute*. These four steps provide adaptability to the system. The Monitoring module collects statistics, which the Analysis module will analyze to determine the system's

state. Then, based on the analysis results, a plan is put in place by the Plan module. Finally, the Execute module executes, if necessary, the actions to adapt the system. In our solution, adaptation actions consist of increasing, reducing, or keeping the number of replicas based on a set of metrics, introduced in the following section.

Since our SPS needs to collect statistics from each operator, the Monitoring module invokes the Storm API to obtain such data. The collected statistics are updated over a configurable time window. In order to avoid increasing communication costs, both the adaptive SPS and the Storm API were deployed on the same node where the Storm's Nimbus master node runs.

As pointed out in Section II, Storm stops processing events when topology resources reconfiguration takes place. In order to overcome this drawback, we have defined, for each operator, a set of pre-loaded replicas (denoted pool of replicas). These replicas remain inactive until required: whenever the system detects the need to increase (or decrease) the number of replicas of an operator, it dynamically activates (or deactivates) one or more replicas of the operator's pool.



(a) One active replica per operator. (b) Activation of a replica.

Fig. 2: SPS with replica pool per operator.

Figure 2.(a) shows our SPS with two operators, O_A and O_B. Each operator has p replicas, but only one is active: $O_{A.1}$ and $O_{B.1}$, respectively. If the SPS requires to increase the number of active replicas of an operator, as shown in Figure 2.(b) for $O_{A.1}$, an inactive replica is activated, in this case, $O_{A.2}$. Once active, $O_{A.2}$ also receives/shares the input data. We should emphasize that this approach considerably reduces the reconfiguration cost compared to the traditional Storm reconfiguration.

Similarly to Storm, the shuffle grouping approach (i.e., tuples are randomly distributed to operators) is used in our SPS solution in order to distribute stream load among active replicas of an operator. In this way, each of them will receive almost the same number of events.

B. Metrics

An adaptive SPS should define metrics to characterize the state of the operators at runtime on a given scenario. Traditional metrics such as throughput, latency, and CPU are the most used in literature. In this work, we propose to integrate 3 metrics denoted *Utilization (U)*, *Execution Time (E)*, and *Queue (Q)*.

$$U = \frac{e \times \mu}{r \times tw_a} \tag{1a}$$

$$E = 1 - \frac{e_b}{e} \tag{1b}$$

$$Q = 1 - \frac{\mu}{q} \tag{1c}$$

Utilization (U): Defined by equation 1a where e represents the average execution time of an event (or tuple) by an operator, μ is the total number of processed tuples, and r is the number of active replicas of the operator. Each of these values is computed within a time window tw_a. This metric characterizes the operator load: if its value is close to 1 (resp., 0), the operator is overloaded (resp., underloaded).

Execution Time (E): Defined by equation 1b where e represents the average execution time of an event (tuple) within a time window tw_a and e_b is the execution time of an event processed by an operator without any extra load. The latter is considered as a baseline, and it is estimated by previously benchmark execution. The goal of the metric is to characterize execution degradation of operators: if the value of e is greater than e_b, the physical machines are overloaded. It is a QoS metric for the SPS which allows to detect struggle operators.

Queue (Q): Defined by equation 1c, where q is defined as the total cumulative number of events (tuples) that arrive to an operator but are not processed within tw_a. The objective of this metric is to analyze the impact of the input queue on the operator with respect to its current processing capacity. The Q tackles the input traffic behavior (traffic shape). Sudden peaks will increase the q value, generating higher values of Q. If Q value is negative, its value is set to 0.

C. MAPE implementation

The MAPE loop control is in charge of providing the self-adaptation capacity to the processing system. Figure 3 shows our implementation architecture. Each of the four MAPE steps performs a specific task:

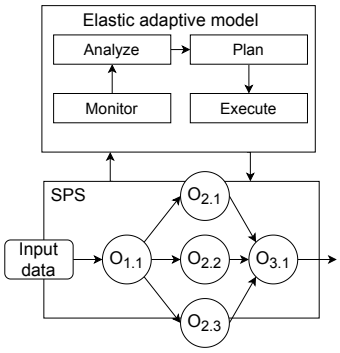


Fig. 3: Self-adaptive Storm architecture

Monitor: The *Monitor* module collects and centralizes statistics from all the operators by monitoring the nodes. These statistics are required to determine the state of every operator. In order to compute the statistics of each operator, all active replicas are taken into account. Thus, μ and q correspond to

the sum of the respective statistics values of these replicas while in the case of e, it is the average values of them.

Analyze: The *Analyze* module aims at determining the state of an operator. For this purpose, the module computes a new metric denoted δ, defined by Equation 2 where U, Q, and E are the metrics previously defined by Equation 1. By introducing weights (ω_U, ω_Q, and ω_E), the impact of the 3 metrics (U, Q, and E) can be balanced, allowing, therefore, the study of the relevance of each metric in different traffic scenarios.

$$\delta = U \times \omega_U + Q \times \omega_Q + E \times \omega_E \quad (2)$$

The δ value characterizes the overall state of an operator. Following a threshold-based approach, two bounds are defined: the upper bound δ_u and the lower bound δ_l. Considering these bounds, an operator can be in one of the three following states: *overloaded*, *stable*, or *underloaded*. These states give information about an operator's *effectiveness* and *efficiency*. *Effectiveness* is the capacity to fully process the input data, while *efficiency* is the capacity to process data by taking advantage of the available resources. If an operator is *overloaded*, it is not capable of processing all the input data, losing efficiency. On the other hand, if the operator is *underloaded*, it processes the input data effectively but not efficiently. Finally, an operator whose state is *stable* processes input data both efficiently and effectively.

Plan: Based on the previous analysis, the *Plan* module defines whether it is necessary or not to modify the system resource capacity. If the operator is overloaded or underloaded, the number of replicas should be increased or decreased respectively by k replicas. Otherwise, if the operator state is stable, the number of replicas will not change. In order to ensure system stability, we define that an operator must remain in the same state by at least two consecutive time windows before carrying out any change in the number of replicas.

As physical resources (or machines) are bounded, it is also necessary to limit the number of operator replicas that can be allocated in those resources. Hence, to decide whether or not it is possible to increase the replicas of an operator, the E metric is used: if its value is greater than δ_E, the scheduler cannot add more replicas.

Algorithm 1 presents the algorithm executed by the *Plan* module for the O_i operator, deciding if the number of replicas of O_i should be increased (or decreased) by k or remain the same.

Execute: Finally, the *Execute* module is in charge of carrying out the change in the current number of replicas of an operator, if required by the *Plan* module.

Algorithm 1 Adaptive Plan algorithm for operator O_i.

Require: Statistics Operator O_i in time window tw_a.
Ensure: Modifying the replicas of operator O_i.

```
U_i, Q_i, E_i = calculateMetrics(O_i)
δ_i = calculateMetricGeneral(U_i , Q_i , E_i)
if δ_i > δ_l then
    φ_i ← overloaded
else if δ_i < δ_u then
    φ_i ← underloaded
else
    φ_i ← stable
end if
if φ_i is the same in tw_j and tw_{j-1} then
    if φ_i = overloaded then
        if E_i > δ_E then
            Add k active replicas for O_i
        end if
    else if φ_i = underloaded then
        Remove k active replicas for O_i
    end if
end if
```

IV. Performance Evaluation

A. Testbed and parameters

The experiments were conducted on the Google Cloud Platform (GCP) using seven Virtual Machines (VMs): three in charge of Zookeeper, three as Supervisor nodes, and one for running both the Nimbus and the adaptive SPS. Two types of machines were used: a `n1-standard-1` (1 CPU, 2.2 GHz, 3.75 GB of RAM) machine for hosting Zookeeper VMs, the Nimbus, and the adaptive system, and a `n1-highcpu-16` (16 CPU, 2.2GHz, 14.4GB of RAM) machine for the Supervisors VMs. Complementary, we used for some experiments low priority machines, denoted *Preemptible*, which are cheaper than the former ones, but they are only available for 24 hours.

Table I summarizes the system parameters and their respective values.

Parameter	Description	Value
tw_a	Time window interval	25 sec
δ_u	Operator state upper limit	0.7
δ_l	Operator state lower limit	0.3
δ_E	Limit for adding replicas	0.7
ω_U	U metric weight	0.45
ω_Q	Q metric weight	0.45
ω_E	E metric weight	0.1
k	Number of active replicas to add/remove	1
p	Replica pool size	50

TABLE I: Adaptive SPS parameter and their values.

B. Study case

We deployed an application composed of four operators whose DAG is shown in Figure 4. It classifies events based on a list of keywords. These events (tweets) were previously collected from Twitter extracted with Twitter API. The four operators classify every tweet by topic, subtopic, category, and subcategory, respectively.

Fig. 4: Twitter application in SPS.

We consider two traffic scenarios: 1) a synthetic traffic that follows a Gaussian distribution and 2) a traffic modeled from real Twitter data related to COVID pandemic, composed of 237 million Tweets collected between March and September of 2020 [18]. The latter is presented in Figure 5 where the purple line represents the real data traces, while the green one models the traffic peaks.

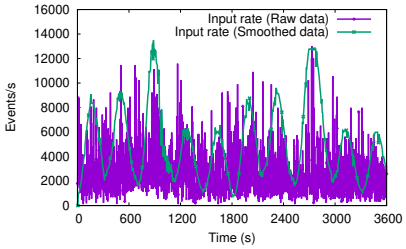


Fig. 5: Traffic shape of Covid Twitter dataset.

C. Evaluation

Our experiments have three goals: (1) the evaluation of the impact of using a pool of pre-allocated inactive operator replicas for scaling in and out the number of operator replicas when compared to Storm reconfiguration approach; (2) the evaluation of the behavior of our SPS, having twitter stream as input; (3) the evaluation of our SPS with a more complex application.

Consequently, we have defined four evaluation metrics:

- *Saved nodes*: this metric described in [19] expresses the difference in the number of used active replicas over the number of overestimated replicas. It is defined by $1 - \frac{r}{r_{over}}$, with r the number of active replicas, and r_{over} the overestimated number of replicas. If the value of the metric is negative (resp., close to 1), the number of resources is overestimated (resp., underestimated). If it is close to 0, the number of resources is well sized.
- *Throughput degradation*: this metric, also described in [19], aims at analyzing the behavior of the system in terms of throughput stability. It is defined by $\frac{|input_{rate} - output_{rate}|}{input_{rate}}$. If the metric value is close to 0, the system has good stability. On the other hand, if it is close to 1, the system is not capable to process the input rate, i.e., the system is unstable.
- *Latency*: is the average time taken by an event between the moment it entered and left the SPS (end-to-end latency). This metric is relevant since SPSs are supposed to deliver real-time processed events.
- *Difference in the number of processed events*: is the difference between the total number of processed events and the total number of received events. It is an important metric since SPSs are used to process high volumes of data, i.e., it should process as much data as possible.

D. Pool of replica vs Storm reconfiguration

We compared Storm, denoted *Storm-Default*, with our modified version of Storm that uses the pool of replicas, denoted *Storm-Pool*, described in Section III. For these experiments, we used the Gaussian-based synthetic input traffic described in Section IV-B. Note that Gaussian traffic shape allows to evaluate the capacity of the system to scale-out and scale-in.

	Saved Nodes	Throughput Degradation	Diff. Processed Events	Latency
Storm-Pool	0.2038	0.2039	0.9987	12121.92
Storm-Default	0.2041	0.4031	0.8627	913.10

TABLE II: *Storm-Pool* and *Storm-Default* metric values.

Figure 6 presents the number of replicas required by each of the two systems. We observe that the difference between both curves is not very significant. Furthermore, the difference in *Saved nodes* values of both systems shown in Table II is of 0.03% and, in terms of memory usage, *Storm-Pool* requires only 2.6% of extra memory when compared to *Storm-Default*, which is due to the pre-allocation of the pools of replicas when deploying the application.

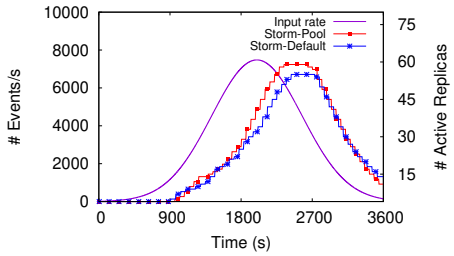


Fig. 6: Total number of replicas of *Storm-Pool* and *Storm-Default*.

Figure 7 shows the output data rate for each system. The main drawback of *Storm-Default* is the need to restart the system at each reconfiguration. We can observe that output rates drop at each reconfiguration. On the other hand, *Storm-Pool* exploits the preloaded replicas that enable the system to process events continuously while adapting itself. We can corroborate the difference in the system behaviors with the *throughput degradation* metric (Table II) that shows a difference of almost 20% between both systems.

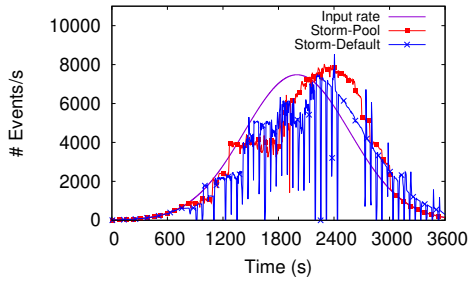


Fig. 7: Throughput of *Storm-Pool* and *Storm-Default*.

The number of cumulative processed events is shown in Figure 8. Once again, we can observe the impact of reconfiguration downtime over the performance of the *Storm-Default*:

there is a 13.6 % difference between both systems in terms of the total number of processed events. We highlight that message loss in real stream processing applications can be critical (e.g., fraud detection systems).

Therefore, based on the above results and discussions, we can conclude that Storm reconfiguration approach is not suitable for real-time applications that require timely responses.

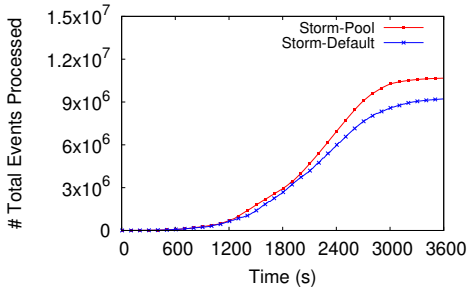


Fig. 8: Total number of processed events of *Storm-Default* and *Storm-Pool*.

We should also point out that the MAPE integrated in our SPS does not use much more extra resources since the monitored information exploited by it is the same one collected by Storm Nimbus. Furthermore, MAPE algorithms do not require much computation time because they just consist in simple mathematical calculations according to the formulas presented in Section III-B

E. Adaptive system vs replicas

The current experiments aim at evaluating the performance of our adaptive system under a real traffic scenario. To this end, we used the metrics introduced in Section IV-C. Collected from Twitter API, the input traffic rate trace is a real Twitter dataset, which we have smoothed. Our adaptive system's control loop decisions are based in δ value. Furthermore, in order to quantify the impact of U and Q in the adaptation process, we also evaluate them independently. The results of the 3 metrics are presented in Table III.

	Saved Nodes	Throughput Degradation	Diff. Processed Events	Latency
δ	0.3996	0.1092	0.8907	39687.51
U	-0.8934	0.2597	0.7402	23441.39
Q	0.4975	0.6830	0.3169	28799.60

TABLE III: Metric values of δ, U, and Q.

Figure 9 shows the number of active replicas for each metric. A considerable increase in active replicas is observed in the first third of the three experiments. Then, in the second period, since the U experiment only analyzes if another replica is necessary to improve operator utilization, it continues to increase the active replicas, which generates a decrease in the performance of the system, due to the overhead of managing a high number of replicas. Likewise, in Table III, the *Saved nodes* metric shows that the experiment with metric U requires 129.3% more active replicas than the one with δ metric. Note that for the U metric, we observe a negative value of saved nodes since sometimes the number of replicas becomes greater than the overestimated value. On the other hand, the Q metric has a 24.5% *Saved nodes* improvement over the δ metric but it succeeds to process only less than 32% of events whereas δ metric can process more than 89% of events.

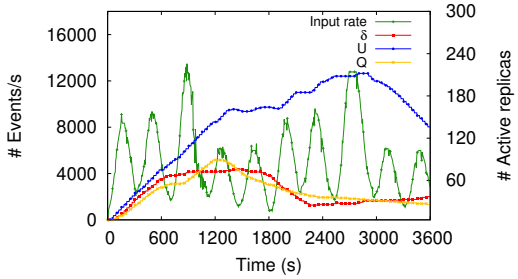


Fig. 9: Total number of active replicas of δ, U, and Q.

The behavior of δ, Q, U output rate as well as input rate are shown in Figure 10. Regarding the three metrics, they are similar in the first and second peaks but not in the third one, since the Q experiment was not able to process events similarly to the other two: the queue increased, generating an overload in the system, which led the application to crash after the fourth peak. On the other hand, both the δ and U experiments continue to process events, until the eighth peak, which generates a saturation in the system of the U experiment, making the application to crash after the ninth peak. As mentioned above, a large number of active replicas induces an overhead for handling them, decreasing the performance of the system. We also observe that there is a 15.05% difference between the δ and U regarding the *Throughput degradation* metric (Table III), which indicates a higher stability of δ experiment then U one. Also, due to the early instability of the Q experiment, there is a difference of 57.38% with respect to the δ one.

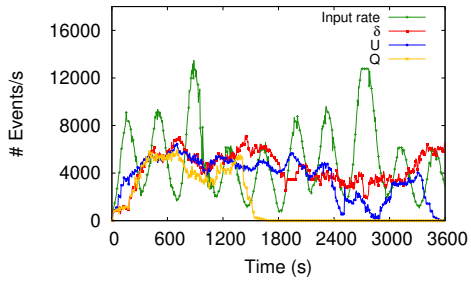


Fig. 10: Throughput of δ, U, and Q.

The total number of processed events is shown in Figure 11. Because of application crash, after $t = 1600s$ (resp., $t = 3300s$) the curve is a constant for the Q (resp., U) experiment. The difference using δ instance with respect to U and Q is 15.05% and 57.3 % respectively (Table III).

Figure 12 presents the latency of the three metrics. At $t = 1600s$, there is a strong rise in the latency for Q until the application crashes. The same happens at $t = 3300s$

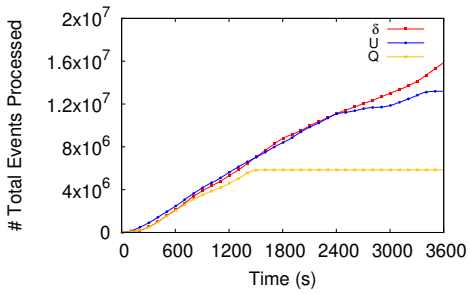


Fig. 11: Total number of processed events of δ, U, and Q.

for the U experiment. Due to the number of events and the system overload, the latest queued events can not been processed. The system becomes then saturated and is not able to continue processing. On the other hand, the latency with δ is on average higher, but the system is capable of processing a greater number of events. Therefore, although the δ instance does not have better performance in terms of latency, it is able of processing a greater amount of data without having to cope with the problem of over or under estimated number of per operator replicas, as in the case of U and Q experiments respectively. The δ latency increase relative to U and Q is 69.3% and 37.8% respectively.

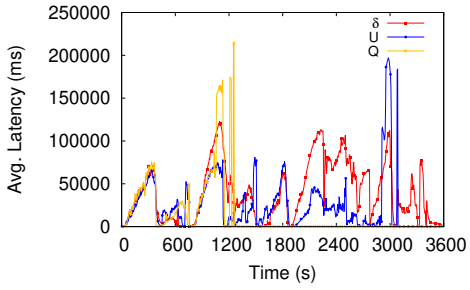


Fig. 12: Latency of δ, U, and Q.

F. A more complex application

We have also evaluated our SPS with a more complex application. Figure 13 represents an application that analyzes Twitter streaming that contains information such as news or opinions. Depending on the type of information, the flow is sent to different flows. Finally, the information is stored in a database.

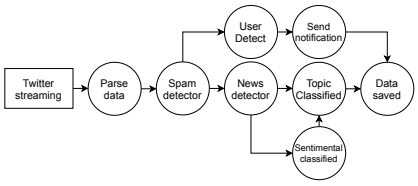


Fig. 13: A Twitter more complex application in our SPS.

We have compared our SPS with an overprovisioning system which always uses a fixed number of replicas per operator.

	Saved Nodes	Throughput Degradation	Diff. Processed Events	Latency
δ	0.5023	0.4252	0.98	179.5401
Overprovisioning	0	0	1.00	132.0272

TABLE IV: Metric values of δ and overprovisioning.

Such numbers are fixed at the beginning of the data processing and do not vary during the experiment.

Table IV presents the results of both our SPS and the overprovisioning system experiments. Due to the time gap to process events, we observe a 42.52% difference between *Throughput degradation* values of the two systems. However, such a difference is not a real problem since the numbers of processed events of the two systems are quite close, as shown in the same table. On the other hand, the difference in latency is 26.46%. We should highlight that, even if our SPS has higher values for the three previous metrics, it presents a gain of 50.23% for the *Saved nodes* metric. In other words, our SPS significantly increases the use of pre-allocated resources which is one of the main objectives of our proposal. Also, in this application, our SPS succeeded to process almost 98% of the total events while consuming 2.7 times less CPU than the static configuration which overestimates the number of replicas.

V. Related Work

There exist some SPSs, such as [20], [19], that provide automatic scalability, being, thus, capable of dynamically modifying resource allocation according to the needs of the application.

The authors in [20] present a predictive approach called AUTOSCALE which analyzes the SPS stream to predict traffic congestion on tasks (operators). For such a prediction, a queue theory principle is applied for gathering information about utilization, arrival rate, and departure rate. A centralized system then analyzes the statistics, predicting data congestion in tasks according to a sliding window. Whenever the system detects a possible congested operator, the number of replicas is increased. However, contrarily to the current article, the work does not present results in scenarios with significant variations in data flow rate.

ELYSIUM [19], implemented in Storm, scales in and out the number of replicas of the operators and, if necessary, modifies the number of workers associated with the application (horizontal and vertical scalability). It provides both a reactive and predictive approach since it analyzes the load at each time window and offers a prediction approach, based on an ANN model. Unlike our work, it has not been evaluated with a real prototype integrated in Storm.

In [21], the authors propose a hierarchical decentralized adaptive SPS. Similarly to our SPS, the implementation has been carried out in Storm, using the MAPE model in the solution. Regarding the scaling policy, the used metric is the CPU utilization of the operator replicas, defining whether a system adaptation is necessary or not. The proposed solution also analyzes the costs associated with each reconfiguration. One

of their parameters is the downtime of the system. Although the cost of re-balancing, i.e., downtime, is considered, it is still a problem in terms of overhead.

There also exist some works that use other SPS frameworks, such as Gesscale which is implemented in Flink [22]. In this work, a model is proposed in order to compute the maximum processing capacity of a physical node. For this purpose, like our approach, the SPS defines multiple different metrics which are the maximum sustainable throughput capacity of a single node, maximum network delay, and parallelization inefficiency. By applying these metrics, the model analyzes the behavior of the system in every time window and, if necessary, modifies the replicas elastically. One of the disadvantages of Gesscale solution, that our SPS does not present, is that, for reconfiguring the system, it is necessary to restart the application, which takes a considerable time (120 seconds).

VI. Conclusion and Future Work

In this article, we have proposed a DAG-based SPS for real-time processing that, for coping with input data rate fluctuation, dynamically adapts the number of active operator's replicas. Contrarily to Storm reconfiguration approach that requires to stop the processing system to scale-in or scale-out, our SPS pre-allocates, for each operator, a set of inactive replicas which are activated (or deactivated) when necessary without the Storm reconfiguration cost, as shown by the performance results conducted on GCP.

We have defined three metrics, the average load of the operator (U), the average execution time of an event (E), and the operator input queue (Q) for characterizing the state of an operator at runtime. By assigning a weight to each of these metrics, our SPS can decide whether the operator is overloaded, underloaded or stable, respectively increasing, reducing, or keeping the same number of active replicas. Performance results with Twitter input data and different evaluation metrics confirm the advantages of using the three metrics compared to a single one.

As future work, in the short term, we intend to render the parameter values of Algorithm 1 (Table I) adaptive, i.e., they would vary according to the application execution state (e.g., operators load, input rate fluctuation, etc.). Another research direction would be to implement a mechanism to predict input rate and operator load variation and then dynamically assign different weight values to the three metrics (U, Q, and E), according to such predictions.

Acknowledgement

This work was funded by the National Agency for Research and Development National Agency for Research and Development (ANID) / Scholarship Program / DOCTORADO BECAS CHILE/2018 - 72190551. This material is based upon work supported by Google Cloud. Nicolas Hidalgo wants to thank the project CONICYT FONDECYT Nº 11190314, Chile and to STIC-AmSud ADMITS Nº 20-STIC-01.

References

[1] J. Leibiusky, G. Eisbruch, and D. Simonassi, *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly, 2012.

[2] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective - Modeling, Scheduling, Load Shedding, and Complex Event Processing*, ser. Advances in Database Systems. Kluwer, 2009, vol. 36.

[3] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 239–250.

[4] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[5] J. Cervino, E. Kalyvianaki, J. Salvachua, and P. R. Pietzuch, "Adaptive provisioning of stream processing systems in the cloud," in *ICDE Workshops*. IEEE Computer Society, 2012, pp. 295–301.

[6] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. Wu, "Elastic scaling of data parallel operators in stream processing," in *IPDPS*. IEEE, 2009, pp. 1–12.

[7] Y. Tang and B. Gedik, "Autopipelining for data stream processing," *IEEE Trans. Parallel Distributed Syst.*, vol. 24, no. 12, pp. 2344–2354, 2013.

[8] W. Hummer, B. Satzger, and S. Dustdar, "Elastic stream processing in the cloud," *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, vol. 3, no. 5, pp. 333–345, 2013.

[9] A. Koliousis, M. Weidlich, R. C. Fernandez, A. L. Wolf, P. Costa, and P. R. Pietzuch, "SABER: window-based hybrid stream processing for heterogeneous architectures," in *SIGMOD Conference*. ACM, 2016, pp. 555–569.

[10] A. Computing *et al.*, "An architectural blueprint for autonomic computing," *IBM White Paper*, vol. 31, no. 2006, pp. 1–6, 2006.

[11] M. Kleppmann, *Making Sense of Stream Processing*. O'Reilly Media, Incorporated, 2016.

[12] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. H. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154 300–154 316, 2019.

[13] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *ICDE*. IEEE Computer Society, 2003, pp. 25–36.

[14] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *ICDE*. IEEE Computer Society, 2015, pp. 137–148.

[15] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang, "Elasticutor: Rapid elasticity for realtime stateful stream processing," in *SIGMOD Conference*. ACM, 2019, pp. 573–588.

[16] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm." IEEE Computer Society, 2014, pp. 535–544.

[17] Y. Xing, S. B. Zdonik, and J. Hwang, "Dynamic load distribution in the borealis stream processor," in *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, 2005, pp. 791–802.

[18] A. Gruzd and P. Mai, "COVID-19 Twitter Dataset," 2020. [Online]. Available: https://doi.org/10.5683/SP2/PXF2CU

[19] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 572–585, 2018.

[20] R. K. Kombi, N. Lumineau, and P. Lamarre, "A preventive auto-parallelization approach for elastic stream processing," in *ICDCS*. IEEE Computer Society, 2017, pp. 1532–1542.

[21] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Gener. Comput. Syst.*, vol. 87, pp. 171–185, 2018.

[22] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth, "Model-based stream processing auto-scaling in geo-distributed environments," in *30th Inter. Conference on Computer Communications and Networks*, 2021.