A predictive model for Stream Processing System that dynamically calibrates the number of operator replicas

Daniel Wladdimiro, Luciana Arantes, Nicolas Hidalgo, Pierre Sens

Compas'2022 : Parallélisme/ Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

A predictive model for Stream Processing System that dynamically calibrates the number of operator replicas

Daniel Wladdimiro[1], Luciana Arantes[1], Nicolas Hidalgo[2], and Pierre Sens[1]

[1]Sorbonne Université, CNRS, INRIA, LIP6 - Paris, France
[2]Universidad Diego Portales, Santiago, Chilli

Résumé
Based on direct acyclic graphs, where vertices and edges respectively correspond to operators and event data flow, Stream Processing Systems (SPSs) are used to process high amounts of data in real time. Operators are allocated in the resources of the infrastructure (e.g., VMs) which are usually replicated for performance sake. We propose in this paper a predictive SPS that dynamically determines the current number of replicas required for each operator based not only on the current resource utilization and data flow variation but also on the events that, due to operator's overloading, could not be processed yet and are, thus, kept in the operator's queue. Preliminary performance results with an application that processes Twitter stream, deployed on Google Cloud Platform (GCP), are presented.

1. Introduction

Nowadays, there is an accelerated increase in the volume of data created by existing applications or systems on the Web, especially because of large-scale user interactions. Hence, processing such data in real-time, delivering useful information in short periods of time, is increasingly requested by companies in different areas such as trading, security, and research, among others. To this end, they use Stream Processing Systems (SPSs) [10].
SPSs are based on directed acyclic graphs (DAG) where vertices and unidirectional edges respectively correspond to operators and event data flows [2]. An external source provides data continuously. Operators are based on light programming tasks (such as filters, counters, storage, etc.) that process the desired information in short periods in a pipeline way. Deployed in a processing infrastructure (e.g., Clouds, cluster, etc.), resources (e.g. VMs) are allocated to operators and often replicated for performance sake. However, in most SPS the number of replicas per operator is defined beforehand and does not change during execution. This static behavior might induce bottlenecks in the processing of events due to the dynamic nature of the dataflow. Sudden traffic spikes may increase some operators' load, increasing end-to-end processing latency as well. To overcome this issue it would be necessary to increase the number of such operators' replicas. On the other hand, in a down spike, resources may be underloaded, and, therefore, their number of replicas should be reduced.
This article proposes a predictive DAG-based SPS algorithm that dynamically defines the current number of per operator replicas necessary to process the input stream. The flow of events is divided into time intervals and our SPS defines, for each operator O, the events that O should process within each time interval. These events concern not only the ones that O's direct operator predecessors sent to it but also those related to previous time intervals that O could not

process yet and are thus kept in a queue. In order to process these events, the ideal number O's replicas should be estimated at each time interval by considering both the number of such events and the average event execution time. Consequently, the number of O's replicas dynamically increases or decreases over the time according to its input rate.
Our SPS extends Apache Storm [10] and uses a predictive algorithm that follows a MAPE model [3], which relies on a four-stage control loop widely used in autonomous systems.
Some preliminary experiments have been conducted on Google Cloud Platform (GCP) with an application that processes Twitter streams. We have compared our predictive adaptive SPS with the original Storm with fixed number operator replicas. Results related to some metrics, such as latency, resource utilisation, and number of processed events are presented and discussed.

2. Storm stream processing system

Storm [13] is a SPS framework implemented in Java that enables the processing of unbounded data flows. A Storm application is a DAG, denoted *topology*.
There are three types of components in a topology : *Streams*, *Spouts*, and *Bolts*. *Streams* or data flows are shared among operators following the DAG model. They are composed of key-values 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 following components of the topology. *Bolts* are the operators. Similarly to *Spouts*, *Bolts* can 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.

3. Our predictive Storm-based SPS

The aim of a predictive model is to dynamically adapt the system in order to process all input events and fast react to system adaptation requirements. Hence, the design of a predictive algorithm is based on the dynamic estimation of the number of replicas of each operator, necessary for processing all incoming events the latter receives. The prediction of the number of replicas depends on the dynamics of the event input rate.
At initialization, our SPS assigns, for each operator, a set of replicas, deployed by the Storm scheduler. Replicas can be either in an *active* or *inactive* state. An inactive replica does not consume CPU but can be dynamically activated when the system detects the need for increasing the resources for the operator in question. The concept of the initial set of replicas was proposed in [14]. Note that, if an operator has several replicas, the input assigned to it will be equally divided among its replicas.

$$r = \frac{\lambda \times et}{ti} \qquad (1) \qquad \lambda_r = \sum \lambda_k \times \theta_k \qquad (2) \qquad \lambda = \lambda_r + \lambda_q \qquad (3)$$

The number of active replicas of an operator O is dynamically recomputed by the Equation 1, where λ is the total number of events received by the operator in a time interval ti and et is the average execution time of an event by the operator. In other words, the objective of this

equation is to estimate, how many replicas would be necessary to process all λ incoming events within ti, considering that each of them is processed in et units of time.
Considering that the time interval equals to $ti = 1\ sec$, Figure 1 shows an example composed by three operators with the same input rate and initial number of replicas (equals to 2) but with different event execution times. Due to this difference, Equation 1 will render $r = 2$, $r = 1$, and $r = 4$ for O_1, O_2, and O_3 respectively. Therefore, such results inform that the number of O_1's active replicas should not change but that of O_2 (resp.,O_3) is overestimated (resp.,underestimated) and should be reduced (resp., increased) to one (resp., four).

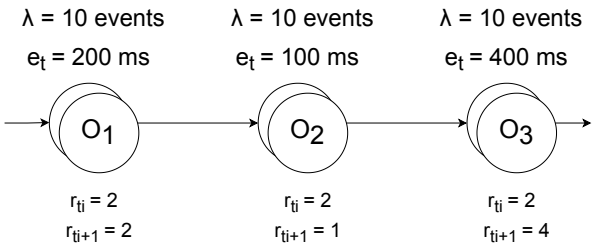
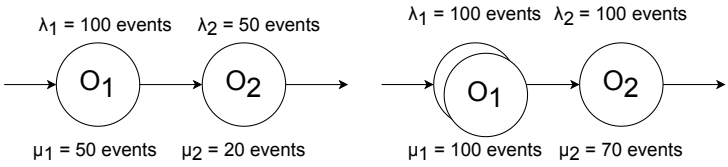


FIGURE 1 – Number of replicas according to equation (1)

Since data stream flows between adjacent operators, λ should express the dependency among them in order to avoid a bottleneck in one operator due to the debottlenecking of the previous ones in the stream data. Events received by an operator and not processed in a time interval are kept in a queue q associated with the operator.
For example, Figure 2a shows two operators O_1 and O_2, where λ and μ respectively correspond to the total number of received and sent events by an operator. Let's consider that O_1 cannot process all incoming events in the time interval ti. If the number of O_1's replicas increases, O_1 might be able to process more events. Consequently, in this case, O_2's λ would increase as well. However, it might happen that, in its turn, O_2 would not be able to process all the incoming events as shown in Figure 2b. We should point out that for long SPS DAGs, the impact of an operator's debottlenecking can induce a domino effect.

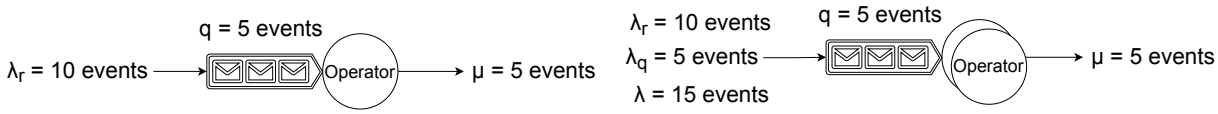


(a) Number of events in ti (b) Number of events in $ti + 1$

FIGURE 2 – Operators bottleneck

Let λ_r denote the input total number of events that an operator O receives from its direct predecessors in the SPS DAG, considering that each predecessor P sends a percentage, denoted θ_p, of its processed events to O. The θ_p parameter is necessary since not all the processed events

of P will be sent to O as P can split its output stream into several ones, sending each of them to different operators. Equation 2 expresses λ_r.
Furthermore, the number of events λ that an operator O effectively receives in a time interval ti should consider not only λ_r but also existing non-processed events kept in O's queue in previous time intervals. Otherwise, if it takes into account only λ_r, the predictive algorithm might reduce the number of O's replicas, which would not be a good decision if the number of events of the queue is high.
For instance, in Figure 3a, there are a high number of queued events in the time interval ti_k. Hence, if in the time window ti_{k+1} the number of received events decreases, the number of replicas of the operator should not be reduced since queued events must also be processed in ti_{k+1}. Equation 3, expresses λ where λ_r and λ_q respectively correspond to the total events received by the operator within a time interval and the queued events from the previous time intervals injected to the operator.



(a) Number of events in ti_k.

(b) Number of events in ti_{k+1}, considering the queue.

FIGURE 3 – Input number of events of an operator (λ)

3.1. MAPE implementation

The MAPE loop control is in charge of providing the self-adaptation feature of our SPS. Each of the four MAPE modules performs a specific task :

1. *Monitor* : module in charge of gathering and centralizing statistics from the graph operators, required to determine the number of replicas for each operator. The monitor requests, at each time interval, the values of λ, et, and the number of queued events q.
2. *Analysis* : module in charge of computing Equation 3 in order to get λ. Note that the analysis will be performed from the beginning of the graph till the last operator.
3. *Plan* : module that, based on the previous analysis and the current number of active replicas of an operator, defines whether it is necessary or not to modify the operator's current number of active replicas. Algorithm 1 shows the pseudo-code of the *Plan* modules which increase/decrease the current number of active replicas, if necessary. The $getReplicas(O)$ function returns the number of current active replicas of O.
4. *Execute* : module which is in charge of carrying out the change in the current number of replicas of an operator, if required by the *Plan* module.

4. Performance Evaluation

Testbed : Experiments were conducted on Google Cloud Platform (GCP) using eleven Virtual Machines (VMs) : three in charge of Zookeeper, seven as Supervisor nodes, and one for running both the Nimbus and the adaptive SPS. Two types of machines were used : a `n1-standard-1`

Algorithm 1 Adaptive Plan algorithm for operator O_i.

1: r_i = computeReplicas(λ_r , et, ti)
2: k_i = r_i - getReplicas(O_i)
3: **if** $k > 0$ **then**
4: Add k active replicas to O_i
5: **else if** $k < 0$ **then**
6: Remove k active replicas from O_i
7: **end if**

(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-4` (4 CPU, 2.2GHz, 3.6GB of RAM) machine for the Supervisors VMs. Complementary, we used for the experiments low priority machines, denoted *Preemptible*, which are cheaper than the former ones, but they are only available for 24 hours

Application and scenarios : We deployed an application composed of four operators which is in charge of analyzing and classifying events, as shown in Figure 4. The traffic model is based on data from Twitter related to COVID-19. Even if the database has 237 millions tweets [5], we have only used a sample for the deployment of the application.
In order to evaluate the r parameter impact, we have considered four scenarios with both the original *Storm*, with fixed number of replicas, and our SPS :
(i) Storm with no replication, $r = 1$; (iii) Storm with $r = 5$; (iii) Storm with $r = 10$; and (iv) Our SPS (adaptive replication). The scenario (iii) corresponds to a *overprovisioning* one, i.e., unnecessary allocated replicas.

Metrics : We have defined four evaluation metrics : (1) *Saved resources* (difference between the number of active replicas and the overestimated one), (2) *Difference in the number of processed events* (difference between the total number of processed events and the received ones), (3) *Throughput degradation*, and (4) *Latency*.See [14] or the appendix for more details about them.

FIGURE 4 – Twitter application in SPS.

4.1. Results

Table 1 summarizes our evaluation metric results for each of the four scenarios.
Figure 5 presents the number of replicas used by the different scenarios. We observe that our SPS can dynamically adapt the number of replicas according to the variation of the input rate. Likewise, results in Table 1 show that *our SPS* saves 21.75% and 71.75% of resources when compared to *Storm* with $r = 5$ and $r = 10$ respectively. We should also point out that such resource saving does not compromise the output quality of the system since, as confirmed by the same table, the loss of events is negligible .
The output rate (throughput) for each scenario is shown in Figure 6. For $r = 1$, since there is no replication, the performance is quite poor. For $r = 5$, although it processes a greater amount of events than the previous one, there are still not enough resources to cope with input spikes.

Therefore, not all input events are processed. On the other hand, when $r = 10$, all input events are processed but resources are underused. Finally, our SPS is able to fully process all input events. Thanks to our adaptive algorithm, even if events are queued in some time intervals, especially when spikes take place, the performance of the system does not degrade. Our SPS presents, thus, stability when processing events, which is confirmed in Table 1 by its *throughput degradation* metric value.
However, in Table 1 we observe that the latency of *our SPS* is higher when compared to the other scenarios. Such an increase is due to our replica adaptation algorithm that, by including queued events in the input data of an operator within a time interval, increases the time to process them.
Such preliminary results are quite encouraging since, even if latency increases, *our SPS* is capable of processing events, independent of input rate variation, not wasting resources.

5. Related Work

Similarly to our approach, there exist some SPS in the literature that also use time intervals to trigger resources allocation adaptations. The latter are usually based one some metrics such as CPU usage [7, 8], processing rate [12, 4], latency [6, 1], etc.
The authors in [9] present a predictive SPS called AUTOSCALE which analyzes the data stream to predict traffic congestion on tasks. Queue theory principle is applied for gathering information about utilization, arrival rate, and departure rate of the tasks. 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 our work, the article does not present evaluation results in scenarios with high variations in the data flow rate.
ELYSIUM [11] is a Storm-based SPS that 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 based on time window and an ANN model. Contrarily to our work, ELYSIUM was not been evaluated with a real prototype integrated in Storm
In [1], the authors propose a hierarchical decentralized adaptive SPS in Apache Storm, using the MAPE model to design the solution. Regarding the scaling policy, the used metric is CPU utilization of the operator replicas, which defines 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, i.e., the time necessary to restart the system which can induce much overhead. Our SPS does not present such an overhead since inactive replicas are pré-allocated at the beginning of the SPS execution.

6. Conclusion

We have presented in this article a Storm-based SPS that tolerates data flow variation by dynamically adapting the current number of operators replica. To this end, the number of events to be processed within each time interval by an operator comprise the output of its predecessors in the DAG as well as events queued in previous time intervals. Even if the application was simple, preliminary evaluation results on GCP show that our SPS was capable of processing most of the input events, without keeping extra resources if they are not necessary.
As future work, we intend to evaluate our SPS with more complex applications and also to compare it with other existing SPSs such as AUTOSCALE [9] or ELYSIUM [11].

	Saved Resources	Throughput Degradation	Diff. Processed Events	Latency
Our SPS	0.7175	0.3216	0.9989	6111.94
$r = 1$	0.9	0.5991	0.3814	47752.24
$r = 5$	0.5	0.3746	0.9965	190.73
$r = 10$	0	0	1	107.71

TABLE 1 – Metric values for *Our SPS* and *Storm* ($r = 1$, $r = 5$, $r = 10$)

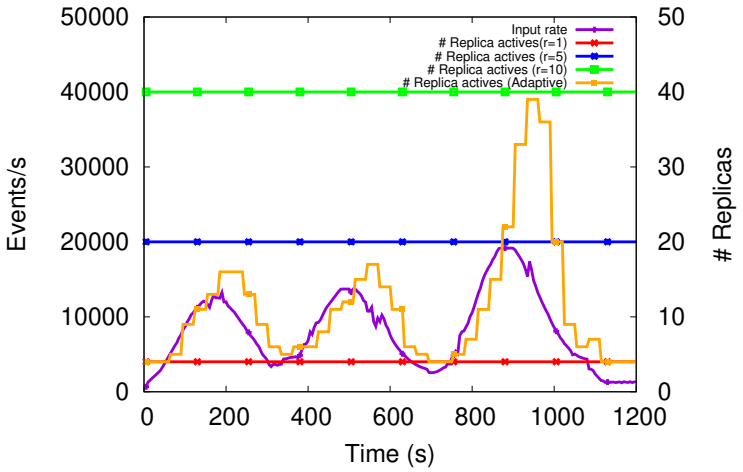


FIGURE 5 – Total number of replicas of *our SPS* and *Storm]* ($r = 1$, $r = 5$, $r = 10$)

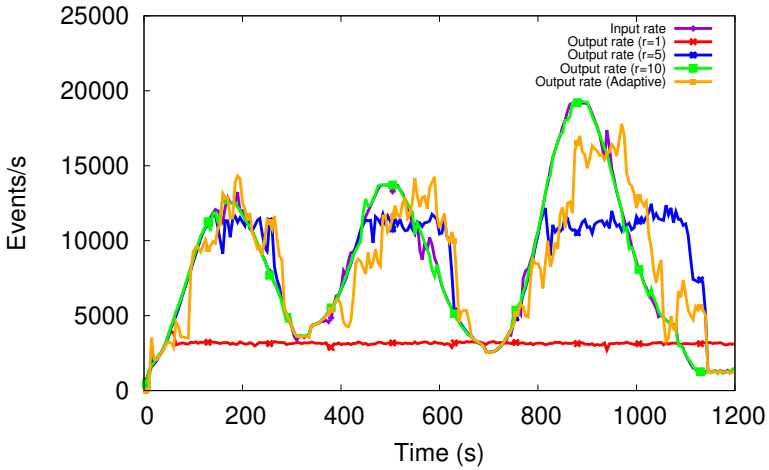


FIGURE 6 – Throughput of *our SPS* and *Storm]* ($r = 1$, $r = 5$, $r = 10$)

Metrics [14]

- *Saved resources* : this metric described in [11] 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 [11], 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.

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 N° 11190314, Chile and to STIC-AmSud ADMITS N° 20-STIC-01.

Bibliographie

1. Cardellini (V.), Presti (F. L.), Nardelli (M.) et Russo (G. R.). – Decentralized self-adaptation for elastic data stream processing. *Future Gener. Comput. Syst.*, vol. 87, 2018, pp. 171–185.
2. Chakravarthy (S.) et Jiang (Q.). – *Stream Data Processing : A Quality of Service Perspective - Modeling, Scheduling, Load Shedding, and Complex Event Processing*. – Kluwer, 2009, *Advances in Database Systems*, volume 36.
3. Computing (A.) et al. – An architectural blueprint for autonomic computing. *IBM White Paper*, vol. 31, n2006, 2006, pp. 1–6.
4. Gedik (B.), Schneider (S.), Hirzel (M.) et Wu (K.). – Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, n6, 2014, pp. 1447–1463.
5. Gruzd (A.) et Mai (P.). – COVID-19 Twitter Dataset, 2020.
6. Hummer (W.), Satzger (B.) et Dustdar (S.). – Elastic stream processing in the cloud. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, vol. 3, n5, 2013, pp. 333–345.
7. Kahveci (B.) et Gedik (B.). – Joker : Elastic stream processing with organic adaptation. *J. Parallel Distributed Comput.*, vol. 137, 2020, pp. 205–223.
8. Koliousis (A.), Weidlich (M.), Fernandez (R. C.), Wolf (A. L.), Costa (P.) et Pietzuch (P. R.). – SABER : window-based hybrid stream processing for heterogeneous architectures. – In *SIGMOD Conference*, pp. 555–569. ACM, 2016.
9. Kombi (R. K.), Lumineau (N.) et Lamarre (P.). – A preventive auto-parallelization approach for elastic stream processing. – In *ICDCS*, pp. 1532–1542. IEEE Computer Society, 2017.

10. Leibiusky (J.), Eisbruch (G.) et Simonassi (D.). – *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. – O'Reilly, 2012.
11. Lombardi (F.), Aniello (L.), Bonomi (S.) et Querzoni (L.). – Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, n3, 2018, pp. 572–585.
12. Tang (Y.) et Gedik (B.). – Autopipelining for data stream processing. *IEEE Trans. Parallel Distributed Syst.*, vol. 24, n12, 2013, pp. 2344–2354.
13. Toshniwal (A.), Taneja (S.), Shukla (A.), Ramasamy (K.), Patel (J. M.), Kulkarni (S.), Jackson (J.), Gade (K.), Fu (M.), Donham (J.) et al. – Storm@ twitter. – In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156. ACM, 2014.
14. Wladdimiro (D.), Arantes (L.), Sens (P.) et Hidalgo (N.). – A multi-metric adaptive stream processing system. – In *NCA*, pp. 1–8. IEEE, 2021.