Compas'2023 : Parallélisme/ Architecture/ Système
LISTIC/USMB - Annecy France, 4-7 juillet 2023

PRESPS : a PREdictive model to determine the number of replicas of the operators in Stream Processing Systems

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, Chili

Résumé
Stream Processing Systems (SPS) are used to process large amounts of data in real time, which are designed as directed acyclic graph (DAG). The vertices correspond to operators and the edges to data stream. Each component is deployed distributed in an infrastructure, which is parallelized. In this paper we propose a predictive model to dynamically and predictively determine the number of replicas required by each SPS operator, based on the input data, per operator event queue and execution time. We have performed preliminary experiments of our solution with another existing solution using a Twitter Stream, deployed on Google Cloud Platform (GCP).

1. Introduction

Online applications create a tremendous amount of data that requires to be analysed in a timely manner. To cope such requirements, specialised systems has been developed, called Stream Processing Systems (SPS) [1].

SPSs are based on directed acyclic graphs (DAG) where the vertices correspond to operators and edges to event streams [5]. Also, an external source sends continuous data to the system for processing. The operators are lightweight tasks (such as filters, counters, classifiers, etc.) which together design a pipeline for event processing. A distributed infrastructure, such as a cloud or a cluster, must be used to deploy operators which, in their turn, may deploy replicas in order to parallelize data processing. However, such a procedure usually requires stopping the system to reorganize the processing graph, resulting in performance degradation. The dynamic nature of the analyzed traffic flows induces bottlenecks and message loss, among others, in the presence of traffic spikes while waste of allocated resources in the absence of spikes.

In this work, we propose to dynamically adapt the stream processing systems graph by increasing or decreasing the number of operator replicas according to the input traffic. To this end, PRESPS extends Apache Storm and uses a predictive algorithm for its adaptation. Following the MAPE control loop, we deploy or stop replicas in real time based on the input traffic and operator's metrics.

We have conducted experiment over the Google Cloud Platform (GCP) using Twitter data flows, also comparing our model with other predictive Storm-based SPS [8].

2. Storm Stream Processing System

Storm [11] is an 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-value tuples. *Spouts* are responsible for capturing the input data of the topology from external sources. They structure the tuples sending them through one or more *Streams* to the next 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.

3. PRESPS

The aim of a predictive model is to dynamically adapt the system in order to process the largest 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.
PRESPS configures a fixed number of replicas for each operator. Replicas can be either in an *active* or *inactive state*. *Inactive* replicas do not consume CPU but can be further activated to cope with traffic spikes. This pool of replicas concept was proposed in [13]. Note that, if an operator has several replicas, the input assigned to it will be equally divided among its replicas.

$$r_i(t+1) = \frac{\widehat{\lambda}_i(t+1) \times et_i}{td} \quad (1) \qquad \widehat{\lambda}^r_i(t+1) = \lambda_G(t) \times \theta_i(t) \quad (2)$$

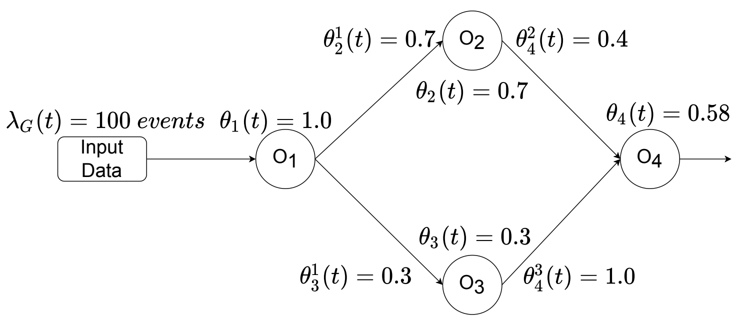
$$\theta_i(t) = \sum_{p \in pred(O_i)} \theta^p_i(t) \times \theta_p(t) \quad (3) \qquad \theta^p_i(t) = \frac{\lambda^p_i(t)}{\mu_p(t)} \quad (4)$$

$$\widehat{\lambda}^q_i(t+1) = |q_i(t)| \quad (5) \qquad \widehat{\lambda}_i(t+1) = \widehat{\lambda}^r_i(t+1) + \widehat{\lambda}^q_i(t+1) \quad (6)$$

For the prediction of the number of replications it is necessary to obtain statistics of the system. Table 1 summarizes all the notations used by our predictive model where $\widehat{v}$ designates a *predicted* value of variable v.
Execution time et_i is determined by a time interval t whose duration is td and is calculated for each operator O_i. The value of et_i has been calculation with a benchmark at the beginning of the deployment of the application.
At the end of each interval, the number of active replicas of an operator O_i for the next time interval is dynamically recalculated by Equation 1. The objective of this equation is to estimate how many active replicas would be necessary for O_i to process all $\widehat{\lambda}_i(t+1)$ estimated events within $t+1$, considering that O_i processes each event in et_i units of time.
Since operators of the SPS are related to each other by the DAG, there exists a dependency between sent and processed events, if we use a linear SPS DAG with two operators, O_1 and O_2, and their respective values of $\lambda^r_i(t)$ and $\mu_i(t)$ (see Table 1). $\mu_1(t)$ and $\lambda^r_2(t)$ are equal since operator O_1 has sent all the events it has processed to its single successor O_2. If i is the initial single DAG operator, then $\lambda^r_p(t) = \lambda_G(t)$ ($\lambda^r_1(t) = \lambda_G(t)$). Note that the increase of O_p's number of active replicas at the end of the interval t has a direct impact in O_p's successors, since, in this case, $\mu_p(t+1)$ increases and thus, $\lambda^r_i(t+1)$ too, inducing a domino effect that the prediction formulations should avoid.

Parameter	*Description*
O_i	operator i
t	time interval number
td	time interval duration
et_i	average execution time of one event by O_i
$q_i(t)$	queue of events received and not processed by O_i at the end of t
$\lambda_G(t)$	number of events sent by input data during t
$\lambda_i^r(t)$	number of events received by O_i during t
$\lambda_i^p(t)$	number of events received by O_i sent from O_p during t
$\mu_i(t)$	number of events processed by O_i during t
$\theta_x(t)$	percentage of events processed of $\lambda_G(t)$ by O_x during t
O_i^p	predecessor operator of O_i in the SPS DAG
$\theta_i^p(t)$	percentage of events produced by O_i^p sent to O_i during t
$\widehat{\lambda_i}(t+1)$	predicted number of events to process by O_i during $t+1$
$\widehat{\lambda_i^r}(t+1)$	predicted number of events received by O_i during $t+1$
$\widehat{\lambda_i^q}(t+1)$	predicted number of queued events to be processed by $O_i$during $t+1$
$r_i(t+1)$	number of replicas of O_i computed at the end of t

TABLE 1 – Parameters notation and their description.



	O_1	O_2	O_3	O_4
$\widehat{\lambda_i^r}(t+1)$	100	70	30	58

FIGURE 1 – DAG example of predicted received number of events, according to Equation 2

In a SPS execution, not always all the output processed events of O_i^p, the predecessor operator of O_i, will be sent to the latter. It might happen that O_i^p splits, filters, or replicates the events into several streams, sending each of them to one of its different successor operators in the DAG. The θ_i^p parameter informs the percentage of processed events of O_i^p sent to O_i. Its value is calculated by Equation 4.

Figure 1 shows a DAG SPS with the respective values of Equation 2 for each operator. We observe that, since θ_1 is equal to 1, O_1 receives all the events sent from the input date and then splits them among O_2 and O_3. As these two operators do not receive all the events from its predecessor O_1, θ_1 and θ_1 have values 0.7 and 0.3 respectively. Finally, operator O_4 receives the events of its predecessor O_2 and O_3. However, O_2 does not send all its processed events to O_4, but only $\theta_4^2 = 0.4$, unlike O_3 which sends all processed events to O_4 ($\theta_4^2 = 1$). The value of θ_4 is, therefore, 0.58, according to Equation 3.

Events received and not processed by O_i are kept in $q_i(t)$. Hence, the number of input events $\lambda_i(t)$ that O_i should actually process in t is composed not only of received events $\lambda_i^r(t)$ but also the events queued in O_i, which correspond to $\lambda_i^q(t)$. $\lambda_i^q(t)$ is defined as the number of

events queued to process in O_i during t, considering there is one queue per operator. Thus, the predicted value of $\widehat{\lambda_i^q}(t+1)$ is defined by the events queued by O_i at the end of the time interval t as defined in Equation 5.
The calculation of $r_i(t+1)$ (Equation 1) requires the value of $\widehat{\lambda_i}(t+1)$ (Equation 6), which in its turn is defined by $\widehat{\lambda_i^r}(t+1)$ (Equation 2) and $\widehat{\lambda_i^q}(t+1)$ (Equation 5). In order to obtain the value of $\widehat{\lambda_i^r}(t+1)$, the value of $\theta_i(t)$ for each O_i needs to be computed. Therefore, given the dependence between the operators, it is necessary to start from the initial operators to the last one. Finally, having obtained the predicted values, the number of replicas $r_i(t+1)$ for each operator O_i can be calculated. See the appendix A for an example about them.

Algorithm 1 Adaptive Plan algorithm for operator O_i.

Require: Statistics Operator O_i in time interval t.
Ensure: Modifying the current number of active replicas of operator O_i.

```
r_i(t+1) ← computeReplicas(λ̂_i(t+1) , et_i, td)
k_i ← r_i(t+1) - getReplicas(O_i)
if k_i > 0 then
    Add k_i active replicas to O_i
else if k_i < 0 then
    Remove k_i active replicas from O_i
end if
```

3.1. Grouping

In shuffle grouping, the received input events are evenly distributed among active replicas without considering that the loaded replicas can receive new events while pending events are in their queue. A replica j of an operator i is defined as $O_{i,j}$

Parameter	*Description*
$\mu_{i,j}(t)$	number of events processed by $O_{i,j}$ during t
$U_{i,j}(t)$	utilization rate of $O_{i,j}$ computed at the end of t

$$U_{i,j}(t) = \frac{\mu_{i,j}(t) \times et_i}{td} \tag{7}$$

TABLE 2 – Parameters notations of the grouping algorithm.

To overcome such a constraint, we propose the *Load balancing grouping* strategy, which considers the load of active replicas in each time interval (t). In this way, the distribution of events is proportional to the load of active replicas, computed using Equation 7, where U is a value between 0 and 1 : 0 informs that the replica has no load, and 1 the 100% utilization of the replica. If the value of U is the same for all replicas, the subsequent events are sent in round-robin. If not, these new events are sent to the replica with the lowest load. Algorithm 2 shows the pseudo-code of the *Load balancing grouping*.

3.2. 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 :

Algorithm 2 Load balancing grouping for operator O_i.

Require: Statistics of r_i replicas of O_i in interval t.
Ensure: Replica $O_{i.m}$ that should process the event.
$m \leftarrow 0$
for $j : 1 \rightarrow r_i$ **do**
  if $U_{i.j} < U_{i.m}$ **then**
    $m \leftarrow j$
  end if
end for
if $U_{i.m} = 1$ **then**
  $m \leftarrow$ getReplicaRoundRobin(O_i)
else
  $U_{i.m} \leftarrow U_{i.m} + \frac{et_i}{td}$
end if
sendEvent($O_{i.m}$)

1. *Monitor* : a module in charge of gathering and centralizing statistics from the DAG. At each time interval, the monitor requests the values of $\lambda_i(t)$, et_i, and the number of queued events q_i.
2. *Analysis* : a module in charge of computing Equation 6 in order to get $\lambda_i(t)$. 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 to modify the operator's current number of active replicas. Algorithm 1 shows the pseudo-code of the *Plan* module, responsible for increasing/decreasing the current number of active replicas, if necessary. The $\mathtt{getReplicas}(O_i)$ function returns the number of current active replicas of O_i.
4. *Execute* : a 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 our 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-8` (8 CPU, 2.2GHz, 7.2GB of RAM) machine for the Supervisors VMs.

Application and scenarios : We deployed an application composed of four operators which is in charge of analyzing and classifying events, as shown in Figure 2. The traffic model is based on data from Twitter related to 2016 USA presidential election. The sample of selected tweets considers periods of the datasets that present high variation. In other words, we select a combination of traffic spikes and under spikes. The methodology for the creation of the testing data set is presented in [2]. For the evaluation, we have compared PRESPS with DABS-Storm, an adaptive SPS, proposed in [8].

Metrics : We have defined four evaluation metrics : (1) *Saved resources* (the difference between the number of active replicas and the overestimated one), (2) *Difference in the number of processed events* (the difference between the total number of processed events and the received ones), (3) *Throughput degradation*, and (4) *Latency*. See [13] or the appendix B for more details about them.

4.1. Results

Table 3 gathers the metric values related to PRESPS and DABS-Storm. Regarding the used resources, PRESPS improves the number of them by 41.77% when compared to DABS-Storm. In addition, the throughput degradation of PRESPS is 35.73% lower than DABS. The most important differences are in latency and the number of processed events. As DABS needs to restart the system at every reconfiguration, operator queues are emptied and their events are dropped. On the other hand, in PRESPS, there exists a pool of replicas and it is only necessary to activate or deactivate the pre-allocated replicas at each reconfiguration, keeping the existing pending events of the queues. Hence, since in DABS-Storm the queued events are not processed during reconfiguration, we observe a decrease in both the number of processed events and latency : DABS processes 17.06% fewer events than PRESPS and latency is decreased by 33.71%.
Figure 3 shows the number of replicas used by the two SPS. DABS and PRESPS can dynamically adapt the number of replicas according to the input rate. However, DABS downtime at each reconfiguration has a direct impact in the throughput, as we can observe in Figure 4, inducing a higher instability and a decrease in the number of processed events.

5. Related Work

In the state of the art, there are different solutions regarding parallelization and elasticity in SPS, works such as [6], [10] and [7].
DABS-Storm, a congestion prevention SPS, is presented in [8]. Its aim is to reduce the degradation of the quality of the results. To this end, a metric is used to estimate the level of activity of the operators. A monitor gathers statistics about the operators activity and then, based on a metric, decides if the amount of resource allocated to each operator should be modified or not.
In [4], the authors propose a hierarchical decentralized adaptive SPS in Storm. 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.
In the work of [3], a SPS adaptation model is proposed which minimises system reconfiguration costs. In this way, the system uses several metrics to predict the future behaviour of the system, which are based on time series and EKF model. Therefore, through the knowledge of the system, a decision is taken whether it is necessary to modify the amount of resources, also considering the cost of such an adaptation.

6. Conclusion

In this paper we have presented a model capable of predicting the input data of the system, thus analysing the number of replicas needed per operator according to the variations of the data flow. To this end, we have analysed both the dependence of the operators in the DAG and their queued events in previous time windows. The results show a decrease in latency as well as an increase in processed events and system stability and decreased latency compared to the other solution. As future work, we would like to evaluate our SPS with other types of datasets or benchmark [12]. Also, we could use other types of predictive models.

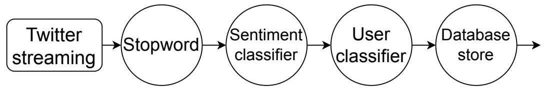


FIGURE 2 – Twitter application in SPS.

System	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
PRESPS	0.5617	0.1831	0.9987	2098.91
DABS	0.3962	0.2849	0.8283	1391.28

TABLE 3 – PRESPS and DABS-Storm metric values.

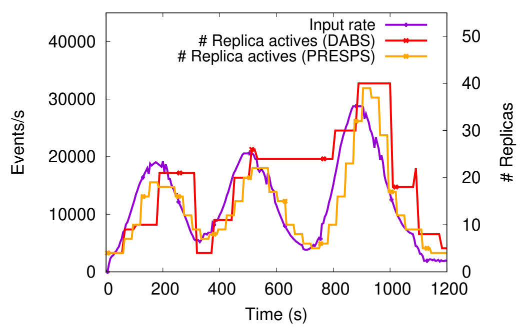


FIGURE 3 – Total number of replicas of *PRESPS* and *DABS-Storm*.

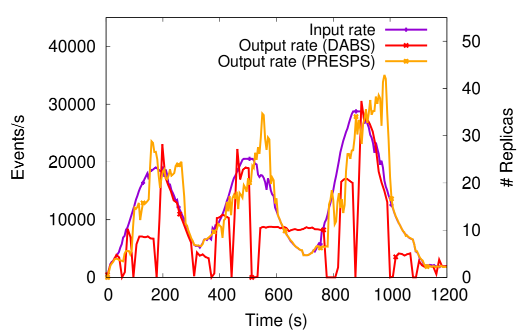


FIGURE 4 – Throughput of *PRESPS* and *DABS-Storm*.

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 the Google Cloud Research Credits program with the award GCP19980904. Nicolas Hidalgo wants to thank the project ANID FONDECYT N° 11190314, Chile and to STIC-AmSud ADMITS N° 20-STIC-01.

Bibliographie

1. Andrade (H.), Gedik (B.) et Turaga (D.). – *Fundamentals of Stream Processing : Application Design, Systems, and Analytics*. – Cambridge University Press, 2014.
2. Bodík (P.), Fox (A.), Franklin (M. J.), Jordan (M. I.) et Patterson (D. A.). – Characterizing, modeling, and generating workload spikes for stateful services. – In *SoCC*, pp. 241–252. ACM, 2010.
3. Borkowski (M.), Hochreiner (C.) et Schulte (S.). – Minimizing cost by reducing scaling operations in distributed stream processing. *Proc. VLDB Endow.*, vol. 12, n7, 2019, pp. 724–737.
4. 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.
5. 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.
6. Fernandez (R. C.), Migliavacca (M.), Kalyvianaki (E.) et Pietzuch (P. R.). – Integrating scale out and fault tolerance in stream processing using operator state management. – In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pp. 725–736, 2013.
7. Kahveci (B.) et Gedik (B.). – Joker : Elastic stream processing with organic adaptation. *J. Parallel Distributed Comput.*, vol. 137, 2020, pp. 205–223.
8. Kombi (R. K.), Lumineau (N.), Lamarre (P.), Rivetti (N.) et Busnel (Y.). – Dabs-storm : A data-aware approach for elastic stream processing. *Trans. Large Scale Data Knowl. Centered Syst.*, vol. 40, 2019, pp. 58–93.
9. 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.
10. Tang (Y.) et Gedik (B.). – Autopipelining for data stream processing. *IEEE Trans. Parallel Distributed Syst.*, vol. 24, n12, 2013, pp. 2344–2354.
11. 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.
12. van Dongen (G.) et Van den Poel (D.). – Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, n8, 2020, pp. 1845–1858.
13. Wladdimiro (D.), Arantes (L.), Sens (P.) et Hidalgo (N.). – A predictive approach for dynamic replication of operators in distributed stream processing systems. – In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 120–129, 2022.

A. Example of prediction in our SPS

Figure 5 shows a linear DAG SPS, with operators O_1, O_2, and O_3. The values of the parameters obtained in the time interval t are presented in Table 4 (a). The calculation of $r_i(t+1)$ (Equation 1) requires the value of $\widehat{\lambda_i}(t+1)$ (Equation 6), which in its turn is defined by $\widehat{\lambda_i^r}(t+1)$ (Equation 2) and $\widehat{\lambda_i^q}(t+1)$ (Equation 5). In order to obtain the value of $\widehat{\lambda_i^r}(t+1)$, the value of $\theta_i(t)$ for each O_i needs to be computed. Therefore, given the dependence between the operators, it is necessary to start from the initial operators to the last one. The value of each θ_i^p is defined in Figure 5, which were calculated according to Equation 4. We will assume that the value $\lambda_G(t) = \widehat{\lambda_G}(t+1)$ Finally, having obtained the predicted values, the number of replicas $r_i(t+1)$ for each operator O_i can be calculated. Such values are presented in Table 4 (b).

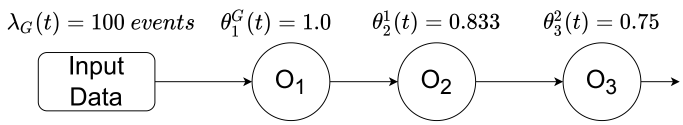


FIGURE 5 – Predictive analysis example for the SPS DAG.

	et_i	$\lambda_i^r(t)$	$\lambda_i^q(t)$	$\mu_i(t)$	$q_i(t)$	$\theta_i(t)$
O_1	16.6 ms	100	40	140	0	1.0
O_2	25 ms	117	10	120	7	0.833
O_3	100 ms	90	20	90	20	0.625

(a) Parameters values calculated for a time interval t.

	$\widehat{\lambda_i^r}(t+1)$	$\widehat{\lambda_i^q}(t+1)$	$\widehat{\lambda_i}(t+1)$	$r_i(t+1)$
O_1	100	0	100	2
O_2	84	7	91	3
O_3	63	20	83	9

(b) Parameters values predicted using Equation 1.

TABLE 4 – Analysis example for DAG presented in Figure 5.

B. Metrics [13]

- *Saved resources* : this metric described in [9] 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 [9], 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.