PA-SPS: A predictive adaptive approach for an elastic stream processing system

Daniel Wladdimiro, Luciana Arantes, Pierre Sens, Nicolás Hidalgo

PA-SPS: A Predictive Adaptive Approach for an Elastic Stream Processing System

Daniel Wladdimiro[a,*], Luciana Arantes[a], Pierre Sens[a], Nicolás Hidalgo[b]

[a]*Sorbonne University, CNRS, LIP6, 4 Place Jussieu, Paris, 75005, France*
[b]*Universidad Diego Portales, Av. Manuel Rodríguez Sur 415, Santiago, Chile*

Abstract

Stream Processing Systems (SPSs) dynamically process input events. Since the input is usually not a constant flow, presenting rate fluctuations, many works in the literature propose to dynamically replicate SPS operators, aiming at reducing the processing bottleneck induced by such fluctuations. However, these SPSs do not consider the problem of load balancing of the replicas or the cost involved in reconfiguring the system whenever the number of replicas changes. We present in this paper a predictive model which, based on input rate variation, execution time of operators, and queued events, dynamically defines the necessary current number of replicas of each operator. A predictor, composed of different models (i.e., mathematical and Machine Learning ones), predicts the input rate. We also propose a Storm-based SPS, named PA-SPS, which uses such a predictive model, not requiring reboot reconfiguration when the number of operators replica change. PA-SPS also implements a load balancer that distributes incoming events evenly among

[*]Corresponding author.
Email addresses: `daniel.wladdimiro@lip6.fr` (Daniel Wladdimiro), `luciana.arantes@lip6.fr` (Luciana Arantes), `pierre.sens@lip6.fr` (Pierre Sens), `nicolas.hidalgoc@mail.udp.cl` (Nicolás Hidalgo)

replicas of an operator. We have conducted experiments on Google Cloud Platform (GCP) for evaluation PA-SPS using real traffic traces of different applications and also compared it with Storm and other existing SPSs.
Keywords: Big Data, Distributed Stream processing, Self-Adaptive System, Dynamic Traffic, Predictive Algorithm, Google Cloud Platform

1. Introduction

The amount of data produced by current Web-based systems or applications is increasing rapidly due to extensive user interactions (e.g. real-time stock trades, multiplayer game iterations, Twitter streaming data, etc.). As a result, there is a growing demand, including in trading, security, and research areas, for systems that can process such data in real-time and deliver helpful information results in short amount of time [1]. Since Stream Processing Systems (SPSs) fulfill these needs, and have been widely used to this end [2]. The goal of an SPS is to process the most of information (number of events) in order to provide high quality of results.

Numerous SPSs are based on directed acyclic graphs (DAGs) whose vertices and unidirectional edges correspond to operators and event data flows [3] respectively. An external source continuously provides data. Light programming tasks (like filters, counters, storage, etc.) are handled by operators that quickly and in pipeline-style process the data (events). In a processing infrastructure (e.g. clusters, clouds, etc.), resources (e.g., VMs) are allocated to execute the operators which are frequently replicated for performance reasons. In most existing SPSs, the number of replicas per operator is predetermined at initialization and remains constant throughout system operation.

However, the dynamic nature of data flows, which exhibit input rate fluctuations, may cause bottlenecks in the processing of events. Unexpected traffic upward spikes may overburden some operators, increasing the overall processing time. To solve this problem, more replicas of the operators are required, aiming at increasing throughput and thus mitigating the loss of events. We point out that for some applications (e.g. bank fraud detection, mitigation in natural disasters), this loss is critical. Nevertheless, a high fixed number of replicas can degrade performance since physical resources are limited. Consequently, the greater the number of operator replicas, the higher the concurrent race among threads.

In [4], the authors present a performance degradation evaluation study based on the number of replicas in SPar [5]. On the other hand, downward spikes can cause resources to become underloaded. In this case, fewer replicas should be allocated to avoid the waste of resources.

Operator's grouping technique, which defines how a stream should be partitioned among the operators, is another crucial decision of SPSs. For instance, shuffle grouping is the most common used. However, the latter does not consider the replica's load state, i.e., current events in the queue waiting to be processed [6], not ensuring, therefore, load balance among the replicas.

This article proposes a predictive DAG-based SPS, named PA-SPS, for stateless operators and skewed data stream environments. PA-SPS is an extension of the well-known SPS Storm[1]. It dynamically allocates the optimal number of replicas per operator necessary to process the input stream, defin-

[1]https://storm.apache.org/

ing the events that each operator O should process within a time interval. In order to predict the input stream and analyze the system's potential future behavior, we propose a predictive model, integrated to PA-SPS. By considering both (1) the events sent to an O by its direct operator predecessors as well as those from earlier time intervals that O could not process at the time, therefore kept in a queue and (2) event execution time, the ideal number of O's replicas for processing these events at each interval is deduced. As a result, the number of O's replicas changes over time depending on the input rate. Complementary to this approach, we propose a new grouping method which takes into account the input load of operator replicas.

Experiments on the Google Cloud Platform (GCP) have been carried out with applications that process Twitter streams, DNS traffic, or system log stream traces. We have evaluated PA-SPS with different configurations and compared it with the original implementation of Storm as well as the state of the art works like SPS DABS-Storm [7] and PSPS [8]. Results related to metrics such as latency, resource utilization, the number of processed events, and error estimation, are presented and discussed in this article.

Sections are organized as follows. Some SPS concepts and definitions are compiled in Section 2. Our solution is presented in Section 3. Results of experiments performed on GCP are presented and discussed in Section 4. Section 5 summarizes some existing work related to adaptive SPS using predictive models. Finally, Section 6 concludes the article, also presenting some future directions.

2. Stream processing systems

Processing large amounts of data in real-time is the aim of SPS, according to [9]. A DAG represents SPS processing logic. Each vertex in the DAG stands for a different operator in the SPS, and the unidirectional edges stand for the data flow. Typically, an operator is a simple task (e.g., counting, filtering, merging, etc.) that receives one or more data flows, processes them, and then sends the transformed data over its output DAG edges. There are two types of operators: *stateless* and *stateful*.

While the latter maintains a state based on previously processed events, the former handles each event independently from those before it. As a result, past events can affect how current events are processed.

Additionally, each operator replica is connected to a thread, and an operator may have multiple replicas. They can therefore process the data concurrently. A data source provides the raw data stream for the operators to process over the DAG. Raw data are homogeneous [10], i.e., a collection of key-value-identified structures (tuples).

When there are replicas, the data flow is divided and distributed among them. For instance, the Field Grouping approach uses the tuple's key to choose which replica will receive it instead of the Shuffle Grouping approach, which sends tuples to each replica randomly.

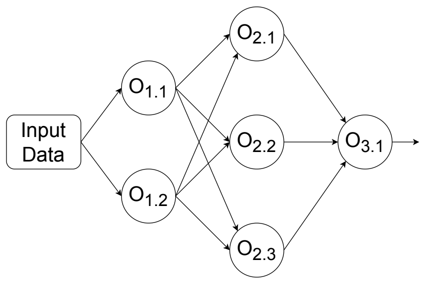
The disadvantage of the shuffle strategy is that load balance may not be achieved. Other existing SPS offer hash-based data partition [11], partial-key based [12], or executor-centric [13] solutions to deal with the distribution problem.

An SPS logical design (DAG) example with an input data source, three replicated operators, and three edges is shown in Figure 1(a). The source provides the operator O_1 with the raw data by partitioning the stream data input and sending it to each of O_1's replicas. Following data processing, O_1 transmits the processed data to O_2, its neighbor downstream in the DAG. Based on the grouping technique, each operator processes the data sent to them and then sends the processed output data to the next. In the absence of an adjacent neighbor, data flow is stopped. O_3 is the final operator to process the data in the figure.

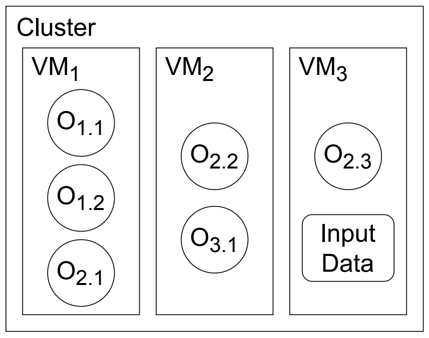
For physical execution, the processing logic (DAG) must be mapped to nodes (machines, cores, or virtual machines) of a distributed platform like a Grid, Cluster, or Cloud. The scheduling algorithm carries out the mapping. Problems with load balance could occur during this process. For instance, there is no assurance that the workload will be evenly distributed among the operators when a random policy is used (as Storm does), as one operator may be more complex than another and machine resources may be heterogeneous [14]. Unbalance problems can occur even with homogeneous computation nodes [15]. Figure 1 show an example of mapping a DAG to physical resources (VMs of a cluster).

2.1. Storm

In this work, we extend the well-known SPS Apache Storm [16]. The latter is an SPS framework implemented in Java that enables the processing of unbounded data flows. In Storm, a processing application is denoted as *topology* which has three different types of components: *Streams*, *Spouts*, and *Bolts*. According to the DAG model, operators share *Streams* or data flows (composed of key-value tuples). *Spouts* are in charge of gathering the



(a) SPS logical architecture.



(b) SPS physical architecture.

Figure 1: Mapping processing DAG to physical resources.

topology's input data from external sources. They organize the tuples before sending them through one or more *Streams* to the following topology elements. The operators are called *Bolts* and they can can send the processed tuples through one or more *Streams* much like *Spouts* can. At run-time, several threads, denoted *executors*, which are instances of the operators, execute the topology operators.

A Storm cluster comprises a master node, called Nimbus, and Supervisor nodes. The latter provides a fixed number of processes, called workers, that run executors. Nimbus is responsible for distributing the application code across the cluster, scheduling executors to available workers, collecting the statistics, monitoring the state of nodes, and detecting failures. Moreover, a distributed coordination service called Zookeeper [17] enables communication among Storm's cluster nodes.

3. PA-SPS Overview

A self-adaptive stream processing system is expected to support high throughput, low latency, and scalability while ensuring the completeness of the results. We thus propose a self-adaptive elastic stream processing system, denoted PA-SPS, an extended version of Apache Storm, that dynamically adapts itself in order to cope with highly variable input rate environments. Based on the MAPE control loop, PA-SPS monitors and predicts input rate behaviour, adapting its processing logic even in presence of stream up-spikes (or down-spikes).

PA-SPS's architecture comprises three components: an adaptive SPS, the input predictive model, and the MAPE control loop. Figure 2 presents the architecture of our solution:

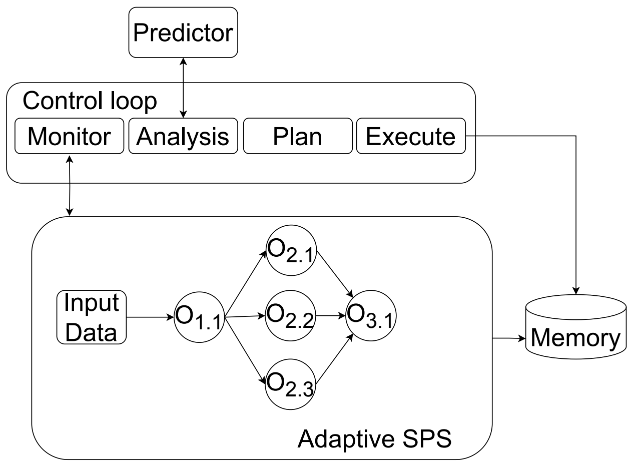


Figure 2: PA-SPS Architecture.

3.1. Adaptive SPS

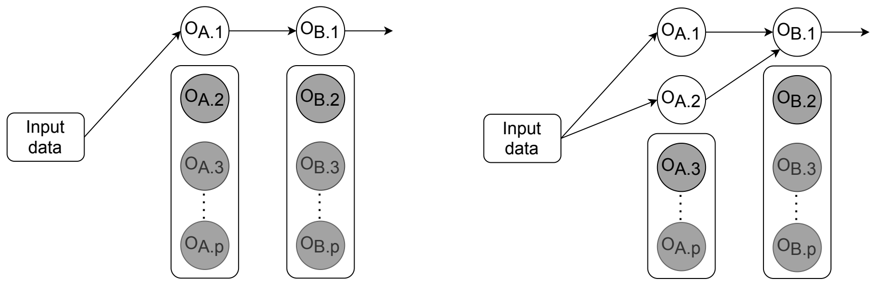
Most SPS cannot re-configure itself at runtime, requiring expert knowledge to configure the system's processing resources according to the environment requirements. Highly variable input rate environments may induce resource over-provisioning, wasting processing resources, or under-provisioning, which may induce the loss of event processing.

In order to handle such scenarios, PA-SPS can dynamically allocate/deallocate operators' replicas at runtime. Such an elasticity enables PA-SPS to adapt its processing logic, optimizing resource usage while minimizing event loss.

PA-SPS exploits two mechanisms that do not exist in the original Storm: a pool of replicas and a load-aware grouping strategy. The first one attempts to reduce adaptation reaction delay, reducing, therefore, event processing latency. Apache Storm's original version provides a rebalancing feature to reallocate the number of executors for a bolt. However, this action involves system downtime, loss of messages, and increasing end-to-end latency. The second mechanism handles operators' replicas event distribution. Grouping strategies in Storm define how events are distributed among operators. Due to the heterogeneous nature of the processing resources and operators' tasks complexity, balance issues may occur, creating bottlenecks and increasing latency.

3.1.1. Pool of replicas

At initialization, PA-SPS assigns, for each operator, a pool of replicas deployed by the scheduler. Replicas can be either in *active* or *inactive* state. The state of a replica can be modified at runtime. An inactive (resp., active) replica consumes negligible (resp., non negligible) CPU power and can be dynamically activated (resp., deactivated) whenever the prediction model of the system detects the need for increasing (resp., decreasing) the replicas for the operator in question. Thus, based on the number of available cores by VMs and the fact that, in general, each replica is associated with a thread, it is possible to set the size of pool. Figure 3(a) and 3(b) consider a DAG with operators O_A and O_B and their respective pool of replicas. In Figure 3(a) there is only one active replica per operator, while in Figure 3(b), a inactive replica of O_A has been activated.



(a) One active replica per operator. (b) Per operator replica pool example.

Figure 3: Example of pool replicas.

Because of its simplicity, performance results showed that the pool of replica is very effective, since PA-SPS is self-adaptive at runtime at a negligible cost, as evaluated and discussed in [18].

3.1.2. Load-Aware Grouping

A grouping technique specifies how a stream (tuples) should be partitioned among operators. Using traditional methods like shuffle grouping, tuples are randomly distributed across operators, ensuring that each operator receives an equal number of tuples. Due to the complexity of the tasks or the heterogeneous nature of the processing resources, load balance issues may occur. In this case, while pending events are still in the processing queue, new events can go on arriving.

$$U_{i.j}(t) = \frac{\mu_{i.j}(t) \times et_i}{td} \tag{1}$$

To overcome this problem, we propose a *load-aware grouping* strategy. The *load-aware grouping* considers the load state of active replicas in terms of $\mu_{i.j}(t)$, i.e., the number of events processed by replica j of an operator i during a time interval t, et_i, the average execution time of one event at operator i, and td, the time interval duration. Note that we define a replica j of an operator i as $O_{i.j}$.

Therefore, the proportional distribution of events considers the current utilization of active replicas. The utilization is computed following Equation 1, where U ranges between 0 and 1. A 0 and 1 values represent 0% and 100% utilization of the replica respectively. If all replicas present the same utilization value, events are sent in a round-robin fashion. Otherwise, events will be assigned to the replica with the lowest load.

Algorithm 1 shows the pseudo-code of the *load-aware grouping technique.*

Algorithm 1 Load-Aware 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.
 1: $m \leftarrow 0$
 2: for $j : 1 \rightarrow r_i$ do
 3:     if $U_{i.j} < U_{i.m}$ then
 4:         $m \leftarrow j$
 5:     end if
 6: end for
 7: if $U_{i.m} = 1$ then
 8:     $m \leftarrow$ getReplicaRoundRobin($O_i$)
 9: else
10:     $U_{i.m} \leftarrow U_{i.m} + \frac{et_i}{td}$
11: end if
12: sendEvent($O_{i.m}$)
```

3.2. Predictive model

Under highly variable input rate environments, input prediction is crucial in order to adapt the system processing logic over the time, which will keep events flowing, ensuring accurate results. In our proposal we use a prediction model for the estimation the optimal number of replicas for a given operator in the DAG. Table 1 summarizes all the parameter's notations used in our predictive model.

$$r_i(t+1) = \frac{\widehat{\lambda_i}(t+1) \times et_i}{td} \tag{2}$$

$$\widehat{\lambda_i}(t+1) = \widehat{\lambda_i^r}(t+1) + \widehat{\lambda_i^q}(t+1) \tag{3}$$

$$\widehat{\lambda_i^r}(t+1) = \widehat{\lambda_G}(t+1) \times \theta_i(t) \tag{4}$$

$$\theta_i(t) = \sum_{p \in pred(O_i)} \theta_i^p(t) \times \theta_p(t) \tag{5}$$

$$\theta_i^p(t) = \frac{\lambda_i^p(t)}{\mu_p(t)} \tag{6}$$

$$\widehat{\lambda_i^q}(t+1) = |q_i(t)| + \sum_{p \in pred(O_i)} \widehat{\lambda_p^q}(t+1) \times \theta_p(t) \tag{7}$$

The model estimates how many active replicas would be necessary for operator O_i to process input data estimation for the next time interval $t+1$ ($\widehat{\lambda_i}(t+1)$). It considers the processing capacity of O_i as the average execution time of one event at the operator (et_i). Note that the value of et_i for each operator is computed with a benchmark before the deploy of the application. At the end of each interval, the number of replicas is calculated following Equation 2. We point out that since prediction of an operator's number of

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
$\widehat{\lambda_G}(t+1)$	predicted number of events sent by input data during $t+1$
$\widehat{\lambda_i}(t+1)$	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.

replicas depends on multiple factors, dynamically determining it is not trivial. We model the calculus of this complex problem based on the previously equations. In order to simplify its understanding, we present some examples of how they are used in our model.

Let us consider that the time interval duration (td) equals 1000 ms. Figure 4 shows an example composed by three independent operators (O_1, O_2, and O_3) which have different event execution time et_i. By prediction, they will receive the same number of events $\widehat{\lambda_i}(t+1)$. In the figure scenario, at the beginning of t, all the three operators have two replicas. However, due to et_i's differences, Equation 2 will render $r_1(t+1) = 2$, $r_2(t+1) = 1$, and $r_3(t+1) = 4$ at the end of t. 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). Note that arriving events $\widehat{\lambda_i}(t)$ will be distributed among the active replicas $r_i(t)$ of O_i by applying our grouping policy presented in Section 3.1.2. The value of $\widehat{\lambda_i}(t+1)$ is determined by Equation 3, which in turn is determined by Equation 4 and 7.

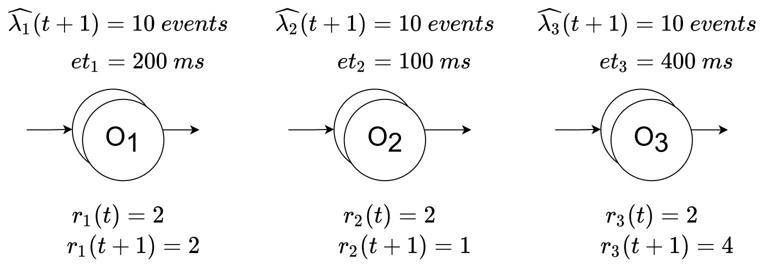


Figure 4: Example of the number of replicas calculation, according Equation 2.

In most SPS, the processing logic is represented by a DAG. The latter establishes a dependency condition where operators share a stream of events according to their location in the DAG. For example, let's use a linear DAG with two operators (see Figure 5), O_1 and O_2, and their respective values of $\lambda_i^r(t)$ and $\mu_i(t)$ (see Table 1). $\mu_1(t)$ and $\lambda_2^r(t)$ 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 $\lambda_p^r(t)$ equals $\lambda_G(t)$ ($\lambda_1^r(t)$ equals $\lambda_G(t)$). 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, $\lambda_i^r(t+1)$ too, inducing a domino effect that the prediction formulations should avoid. For example, in Figure 5, if $\mu_1(t+1)$ increased from 5 to 10 events due to replication of O_1 at the end of t, $\lambda_2^r(t+1)$ would increase as well. Hence, if the operators process all received events during $t+1$, we have that $\lambda_G(t+1) = \lambda^r_1(t+1) = \mu_1(t+1) = \lambda^r_2(t+1)$ and, consequently, all operators O_i are dependent on $\lambda_G(t)$.

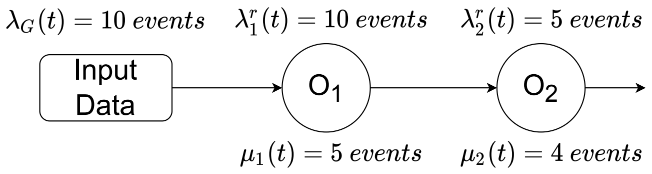
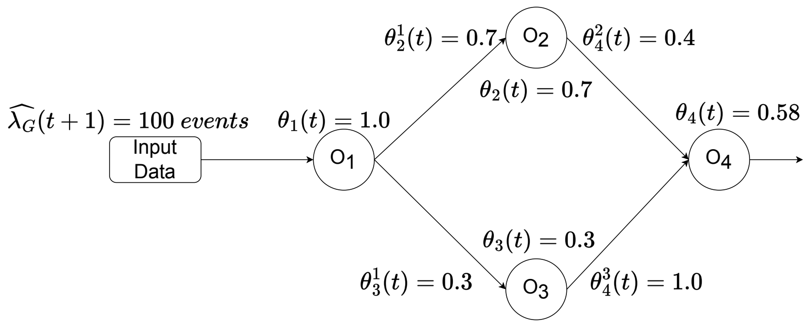

$\lambda_G(t) = 10\ events$
$\lambda^r_1(t) = 10\ events$
$\lambda^r_2(t) = 5\ events$
Input Data
O_1
O_2
$\mu_1(t) = 5\ events$
$\mu_2(t) = 4\ events$

Figure 5: DAG operators dependence example.

The above example is a borderline case. In SPS DAG, not always, all the output processed events of O^p_i, the predecessor operator of O_i, will be sent to O_i. It might happen that O^p_i 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^p_i parameter tackles this issue by informing the percentage of processed events of O^p_i sent to O_i. The latter is calculated by Equation 5 for each operator. Figure 6 shows a DAG example and a θ^p_i value estimated following Equation 5. Notice that given the dependence between the operators, it is necessary to start from the initial operator downstream to the last one. Since θ_1 is equal to 1, O_1 receives all the events sent from the input and then splits them among O_2 and O_3. In this case, the two operators do not receive all the events from their respective predecessor, O_1 in this case; therefore θ^1_2 and θ^1_3 have values 0.7 and 0.3 respectively. Finally, operator O_4 receives events from its predecessor operators O_2 and O_3. However, O_2 does not send all its processed events to O_4, but only $\theta^2_4 = 0.4$, unlike O_3 which sends all processed events to O_4 ($\theta^2_4 = 1$). The value of θ_4 is, therefore, 0.58, according to Equation 6.



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

Figure 6: DAG example of predicted received number of events according to Equation 4.

For the input prediction we apply Equation 4. $\widehat{\lambda_G}(t+1)$ is obtained by predicting the number of input events sends during $t+1$. The observation window was set to one second. The number of samples was one hundred, so as to be statistically representative and not degrade the response time of the adaptation [19]. For prediction, an asynchronous system based on REST API was implemented in Python, in which it is possible to use different predictive models. The chosen models are the following:

- *Basic*: considers that the input data values during $t+1$ will behave the same way as they did during t. This model was used in [8].
- *LR*: uses a simple linear regression similar to the one presented in [20].
- *FFT*: Fast Fourier Transform decomposes functions depending on space or time into functions depending on frequency. It allows to predict the input data by modeling its behavior as a time series [21]. The

parameters values are equal to those for defects[2] in the *Darts* library [22].

- *ANN*: uses a neural network regression model, specifically a Multi-Layer Perceptron (MLP). It implements a MLP algorithm for training and testing data sets using backpropagation and stochastic gradient descent methods [23]. The parameters values are equal to those for defects[3] in *Scikit-learn* library [24].

- *RF*: Random Forest combines ensemble learning methods with the decision tree framework to create multiple randomly drawn decision trees from the data. [25]. The parameters values are equal to those for defects[4] in *Scikit-learn* library [24].

Finally, we should consider the events received and not processed during t by O_i which are kept in $q_i(t)$. Hence, the number of input events $\widehat{\lambda}_i(t+1)$ that O_i should actually process in t is composed not only of received events $\widehat{\lambda}_i^r(t+1)$ but also of the events queued in O_i and its predecessor operators O_p^i due to the domino effect on the DAG. For this reason, we have defined $\widehat{\lambda}_i^q(t+1)$ that represents the number of events queued in O_i and the percentage of queued events that will be sent by its predecessor operators O_i^p during t. For the definition of Equation 7, we have considered that there is one queue per operator.

[2]https://unit8co.github.io/darts/generated_api/darts.models.forecasting.fft.html

[3]https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

[4]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

3.3. MAPE Control Loop

Monitoring, Analysis, Planning, and Execution is referred to as the MAPE control loop. This model is exploited in most autonomic systems. By repeating these four steps, the system can detect issues throughout the capture of data and its analysis. If a problem is found, a strategy is developed to address the issue and then executed. The MAPE control loop brings the system autonomic features such as self-configuration and self-optimization.

In our system, the MAPE model integrates the before-mentioned components where each of the four MAPE modules performs the following tasks:

1. *Monitor*: module in charge of collecting statistics from the DAG. At a predefined time interval, the monitor requests the values of $\lambda_i(t)$, et_i, and the number of queued events q_i.
2. *Analysis*: The module analyzes input data and predicts its behavior following Equation 3 and 7. Note that the analysis is performed for each operator in the DAG.
3. *Plan*: Based on the previous analysis and the current number of active replicas of an operator, the module defines whether it is necessary to modify the operator's current number. Algorithm 2 shows the pseudo-code of the *Plan* module, responsible for increasing/decreasing the current number of active replicas, if necessary. The $getReplicas(O_i)$ function returns the current active replicas of O_i.
4. *Execute*: module in charge of carrying out the change in an operator's current number of replicas, if required by the *Plan* module.

Algorithm 2 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 $O_i$.
1: $r_i(t+1) \leftarrow$ computeReplicas($\widehat{\lambda}_i(t+1)$ , $et_i$, $td$)
2: $k_i \leftarrow r_i(t+1)$ - getReplicas($O_i$)
3: if $k_i > 0$ then
4:     Add $k_i$ active replicas to $O_i$
5: else if $k_i < 0$ then
6:     Remove $k_i$ active replicas from $O_i$
7: end if
```

4. Performance Evaluation

This section presents performance results related to the evaluation of PA-SPS and its ability to adapt to the dynamics of the event stream without degrading the rate of processed events. In Section 4.1, we introduce our system settings and the use-case application which analyses tweet streaming. The implementation code is available in a public repository[5].

Evaluation experiments consist of five steps: (1) an analysis of predictive models (Section 4.2); (2) a comparison of PA-SPS with the original Storm, using a fixed number of replicas (Section 4.3) as well as with (3) the two predictive SPSs DABS-Storm and PSPS, respectively proposed in [7] and [8] (Section 4.4); (4) experiments with a complex application (Section 4.5) and finally (5) with other input stream rather than tweet stream (Section 4.6).

4.1. Experiment Environment

Experiments were conducted over the Google Cloud Platform (GCP) employing eleven Virtual Machines (VMs): three in charge of Zookeeper, seven

[5]https://github.com/dwladdimiroc/sps-storm

as Supervisor nodes, and one for running both the Nimbus and PA-SPS. Three types of machines were used: a `n1-standard-1` (1 CPU, 2.2 GHz, 3.75 GB of RAM) machine for hosting Zookeeper VMs, a `n1-standard-4` (4 CPU, 2.2 GHz, 15 GB of RAM) for Nimbus and PA-SPS, and a `n1-highcpu-8` (8 CPU, 2.2GHz, 7.2GB of RAM) machine for the Supervisors VMs.

4.1.1. Application

We have deployed an application (or topology) composed of four operators which analyzes and classifies tweet events, as shown in Figure 7. The events (tweets) were previously collected from Twitter and extracted using the Twitter API. Events classification is based on the tweets' content and the identity of the person who has published them. Classified tweets are stored in a database.

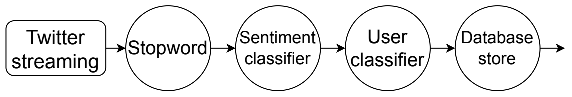


Figure 7: Twitter application in SPS.

4.1.2. Scenario

The same input stream has been used and applied for all the experiments except the last one. The traffic model is based on data from Twitter related to COVID-19, with 237 million tweets [26]. However, we have considered only a sample of these tweets in the experiments, i.e., those in periods of the datasets that present high rate variation. In other words, we select a combination of traffic spikes and under spikes to compose the input traffic for the experiments. The methodology adopted to compose the testing dataset was introduced in [27]. Figure 8 presents the dataset. The purple line represents the actual data trace behavior, while the green line is the composed traffic employed in our tests.

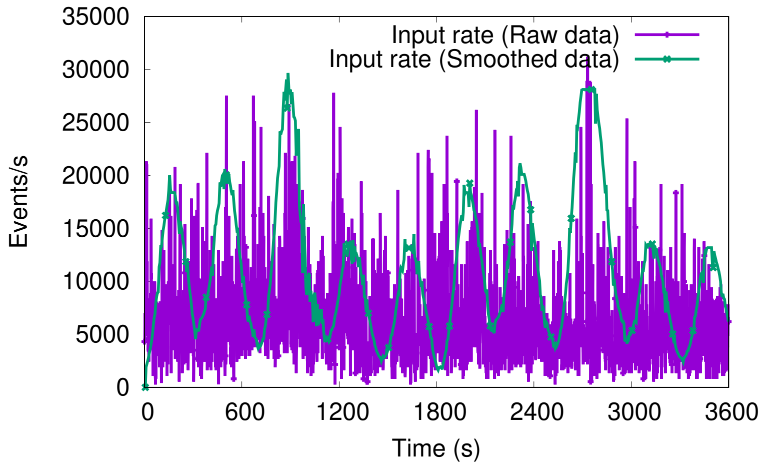


Figure 8: Traffic shape of Covid Twitter dataset.

4.1.3. Metrics

We have defined six evaluation metrics.

- *Saved resources*: proposed in [28], this metric expresses the proportion of resources (active replicas) saved with respect to a statically over-provisioned configuration. It is defined by $1 - \frac{r}{r_{over}}$, with r the number of active replicas, and r_{over} the overestimated number of replicas. r_{over} is the number of replicas needed to process all the events during the highest input rate peak of the benchmark. Note that if the value of the metric is close to 1, a high number of resources has been saved.
- *Throughput degradation*: this metric, also described in [28], 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 a good stability. On the other hand, if it is close to 1, the system is not capable to process the input rate and the system is unstable.

- *Latency*: is the average time taken by an event between the moment it enters and leaves 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.

- *Error estimation input*: is the mean absolute percentage error of the difference between the input rate and the predicted input rate during each interval.

- *Error estimation replica*: is the mean absolute percentage error of the difference between the number of replicas needed to process all events and the number of predicted replicas during each interval.

4.1.4. Parameters

Based on the study presented in [8], we have selected parameter values for which PA-SPS processes the greatest number of events without significantly degrading latency. Such values are: $td = 30s$, $t_{out} = 30s$ and $q_{size} = 100000$. For calculating the *Saved resources* metric, we have fixed $r_{over} = 32$ (i.e., $r_i = 8$).

4.2. Predictive models

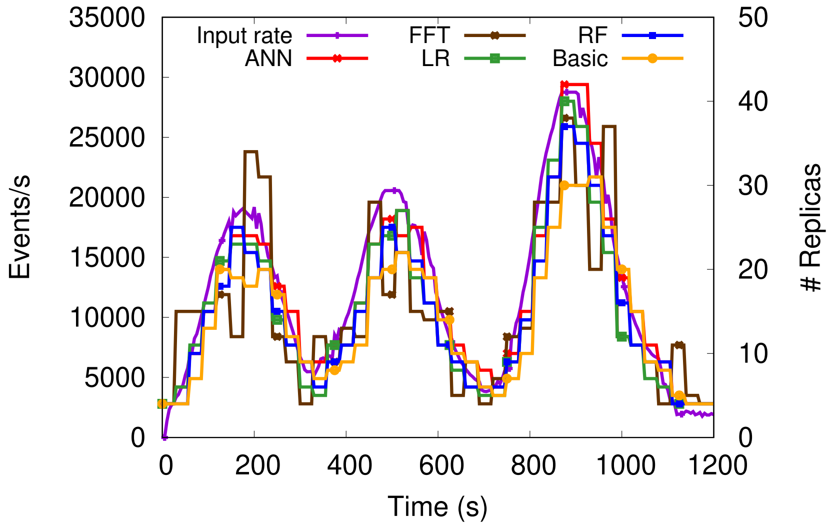
We propose in this section to discuss the use of predictive models for the calculation of $\widehat{\lambda_G}(t+1)$ and their impact on system performance.

For each predictive model, Table 2 shows the respective values for the above discussed metrics. There is no difference in processed events, except for the RF model that presents a slight decrease in the number of processed events, representing a loss of only 0.4% of the incoming events. Therefore, all models are reliable to be used in whole event processing experiments.

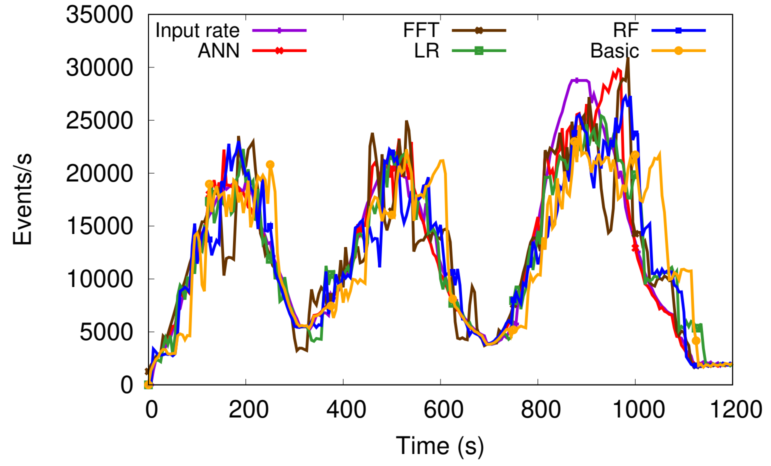
Pred. Model	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)	Error Est. Input	Error Est. Replica
ANN	0.475	0.070	1.000	355.490	0.212	0.277
FFT	0.519	0.189	1.000	1023.380	0.249	0.345
LR	0.533	0.195	1.000	663.030	0.090	0.140
RF	0.538	0.227	0.996	583.921	0.140	0.193
Basic	0.560	0.325	1.000	1295.490	0.180	0.287

Table 2: System metric values of the predictive models.

We can also observe that ANN has the lowest latency, with a difference of 39.12% compared to the second lowest latency model (RF). Such values mean that PA-SPS using ANN processes the received events faster than the others. However, in this case, it needs a larger amount of resources, as shown by the saved resources metric, where ANN has the lowest value, i.e., 15.17% worse than the Basic model metric. Thus, there is a trade-off between latency and the amount of resources: on the one hand, if the requirement is a SPS that processes all incoming events with low latency, ANN is the most suitable model; on the other hand, if the aim is the reduction of costs and the loss of events is not an issue, having an acceptable latency, RF is more suitable.



(a) Total number of replicas.



(b) Throughput of Storm.

Figure 9: Adaptation of PA-SPS using different predictive models.

Regarding the estimation error of the input rate and number of replicas, a lower estimation error can not be interpreted as better performance. Overestimating the input rate implies an overestimation of the number of replicas, but using a larger amount of resources. Consequently, the margin for event processing is larger as shown in Figure 9(b). Conversely, if the number of replicas is underestimated, the margin for processing events is smaller, making it more likely that events will be stuck and the system will be more unstable. In this case, events will probably get stuck in queues, making the SPS more unstable. Regarding accuracy, LR has the best one with an improvement of 57.54% over ANN, which does not mean that it present better performance, because there are moments when underestimating the number of replicas decreases the processing of events in the execution of the system. Unlike ANN that presents an overprovisioning of resources whenever the curve rises. On the other hand, in FFT, its estimation error has a strong impact in PA-SPS performance since it does not accurately predict the behaviour of the input rate, as shown in Figure 9(a).

4.3. Comparison with Storm

Experiments compared PA-SPS with the original *Storm* where the number r of replicas per operator is fixed. We have considered three configurations for *Storm*: no replication ($r = 1$); four replicas ($r = 4$); eight replicas ($r = 8$). The latter corresponds to the *overprovisioning* configuration where the total number of replicas, r_{over}, is equal to 32 in PA-SPS. To calculate the value of $\widehat{\lambda_G}(t + 1)$, *ANN* was used.

Table 3 summarizes the results of the different configurations. We observe that the system without replication ($r = 1$) has a very low performance, because it only processes 33.1% of the incoming events. Such a result is due to the lack of adaptation when incoming events increase. Consequently, there exists a bound for the number of events to process and the others are queued. Therefore, although such a configuration presents low resource usage, it is not recommended for performance sake.

On the other hand, the $r = 4$ configuration has only a 1.3% difference between incoming and processed events. The decrease in the amount of resources by 50% increases latency by 43.03%. Thus, once again, there is a tradeoff between performance and used resources, corroborating to our previous discussion.

Finally, above $r = 4$, PA-SPS has a 5% decrease in Saved Resources, but it is able to process all incoming events. Compared to the $r = 8$ configuration PA-SPS presents: (1) a 131.57% higher latency, whose impact should be balanced with its ability to dynamically adapt itself; (2) a difference of 7.01% in throughput degradation, which means that it greatly adapts itself in order to process most of incoming events.

System	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
PA-SPS	0.475	0.071	1.000	355.490
$r = 1$	0.875	0.515	0.331	196962.950
$r = 4$	0.500	0.855	0.987	269.750
$r = 8$	0.000	0.000	1.000	153.510

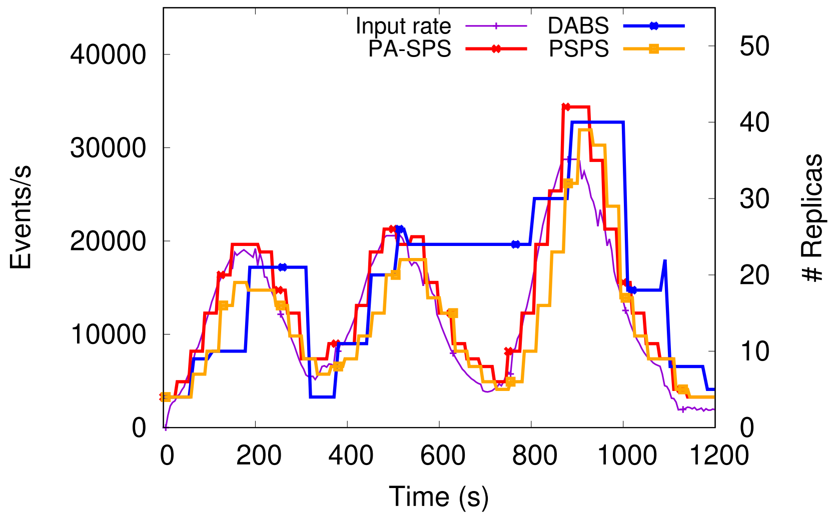
Table 3: System metric values with *PA-SPS* and different configurations in *Storm*.

4.4. Comparison with other existing SPSs

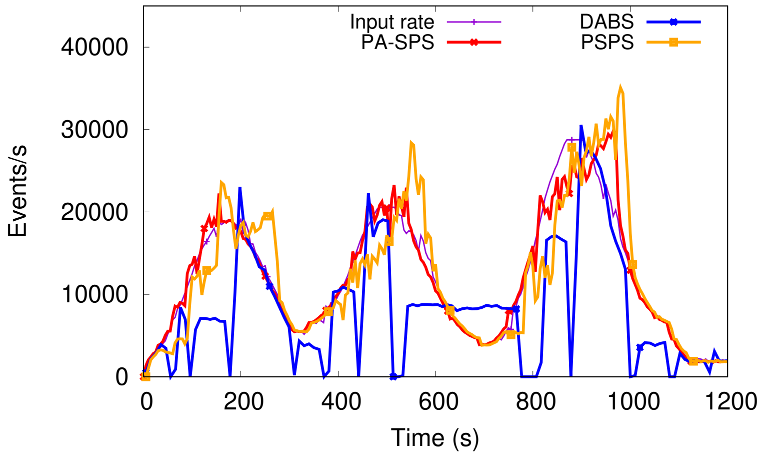
Table 4 gathers the metric values related to PA-SPS as well as PSPS and DABS-Storm (denoted DABS), proposed in [8] and [7] respectively (see Section 5). We can observe that the difference in the number of events processed by PA-SPS and PSPS is negligible, but not with DABS. The latter decreases by 17.2% of the events, a consequence of its deployment. Similarly to Storm (see Section 3.1.1), DABS needs to restart the application for reconfiguring the number of replicas, which is not the case of PA-SPS neither PSPS due to their pool of replicas. Therefore, downtime has an impact in the number of events that are processed. As we have already discussed, the increase of resources has a correlation with the decrease of latency, but there are scenarios where it does not apply. For example, in DABS, saved resources have been decreased by 16.63% when compared to PA-SPS, but its latency is 291.36% higher. Such a behavior can be explained since DABS overestimates resources in non-critical intervals, as observed between $t = 800$ and $t = 900$ in Figure 10(a), which is useless in the case of curve peaks, where more resources are needed. On the other hand, corroborating the previous correlation, the main difference between PSPS and PA-SPS is latency and saved resources. The former increases 490.42% (resp. 15.32%) latency (resp. saved resources) that the latter. This difference has aend-to-end n impact also in the stability of the SPS, as shown in Figure 10(b). Correctly estimating the input rate for PA-SPS implies the dynamic allocation of the necessary resources to process the incoming events and thus the decrease in throughput degradation of 61.20% when compared to PSPS.

System	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
PA-SPS	0.475	0.071	1.000	355.490
PSPS	0.561	0.183	0.998	2098.910
DABS	0.396	0.284	0.828	1391.280

Table 4: PSPS and DABS-Storm metric values.



(a) Total number of replicas.



(b) Throughput of *Storm*.

Figure 10: Comparison with *DABS*, *PSPS* and *PA-SPS*.

4.5. Complex application

We have also evaluated PA-SPS with a more complex application, whose DAG is represented in Figure 11. It analyzes Twitter streaming containing information such as news or opinions. Depending on the type of information, the stream can be split. Results are stored in a database. In the experiments, $r_{over} = 40$.

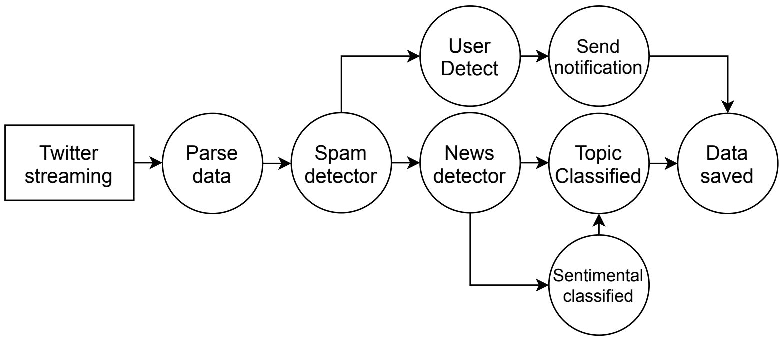


Figure 11: A Twitter more complex application in *PSPS*.

Table 5 shows evaluation results obtained with PA-SPS and Storm with a fixed per operator number of replicas of five ($r = 5$). In PA-SPS, we observe a high reduction of used resources with 68.8% fewer active replicas, when compared to Storm. Such a decrease has an impact on the physical used resources: CPU consumption of PA-SPS (resp. Storm) is in average, 9.57% (resp. 14.66%). This difference happens because each replica is associated with a thread. Therefore, with a fixed number of 5 replicas, Storm requires more CPU than PA-SPS where the number of replicas dynamically varies.

Pred. Model	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)
PA-SPS	0.688	0.031	1.000	209.270
$r = 5$	0.000	0.000	1.000	31.990

Table 5: Complex application metric values.

4.6. Other datasets

We consider the following datasets: Twitter raw stream, DNS traffic data stream, and distributed system logs.

Similarly to the application presented in Section 4.1.1, we have deployed two other ones with linear topology, composed of four operators, that respectively analyze and classify events based on DNS traffic and distributed system logs datasets.

4.6.1. Twitter raw stream

In order to evaluate the impact of having used in the previous experiments Twitter smoothed traces instead of the original one (*raw data*, described in 4.1), we conducted an experiment with the same application but with Twitter raw data as input. We have considered $r_{over} = 32$.

Table 6 shows the results obtained, which have similar values with Table 2, related to smoothed data. PA-SPS, regardless the model, has processed most of the received events, with only 1.2% (resp.,1.3%) of events not processed by LR (resp., RF). Also, the lowest latency corresponds to ANN, although the difference in latency with respect to the second best model (Basic) is 0.47%. By using a more unstable input rate, the estimation error of the models increases as the input behaviour is more complex to predict. Since PA-SPS also becomes more unstable, the throughput degradation increases, as not all events received are processed. FFT presents the highest difference because the input rate does not have a stationary behaviour. Consequently, there is a large percentage of error in the prediction of the input and the number of replicas which degrades performance.

4.6.2. DNS traffic data stream

We have deployed an application with linear topology as shown in Figure 12, composed of four operators which analyzes and classifies events based on

Pred. Model	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)	Error Est. Input	Error Est. Replica
ANN	0.395	0.213	1.000	1044.510	0.424	0.466
FFT	0.153	0.579	1.000	12285.990	0.903	1.282
LR	0.421	0.261	0.988	1366.680	0.397	0.391
RF	0.539	0.381	0.987	2610.330	0.253	0.398
Basic	0.513	0.251	1.000	1049.490	0.312	0.407

Table 6: Twitter raw stream metric values.

DNS traffic. For the experiments, we have used the dataset presented in [29] with fixed $r_{over} = 12$ (i.e., $r_i = 3$).

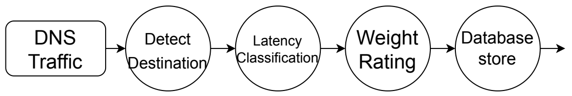


Figure 12: DNS application in PSPS.

The aim of this experiment is to verify the adaptation ability of PA-SPS with an input with a different fluctuation than the previous inputs and then analyse the behaviour of it with each predictive model. Table 7 summarizes the obtained results. The processing capability of PA-SPS is reconfirmed since each proposed model has none or a negligible difference in the number of processed events. The highest difference percentage is around of 2.5% (RF) when compared to LR.

Figure 13(b) shows both the input rate and the throughput. Despite the high dynamics of the input rate, PA-SPS is able to adapt its number of resources in order to process the largest number of events in each time interval. In this experiment, the model with the best performance is FFT, having the lowest value of latency and throughput degradation. On the contrary, the input prediction error is the highest. Considering the values of saved resources, we can conclude that there was an overestimation of the input which led to an overestimation of resources as shown in Figure 13(a). If a model with lower resource utilisation is required, RF is a good choice, given that it has a difference of 22.08% of the saved resources value with respect to FFT. It is worth remarking that due to the above difference, FFT throughput degradation and latency decrease by 29.37% and 27.25% respectively.

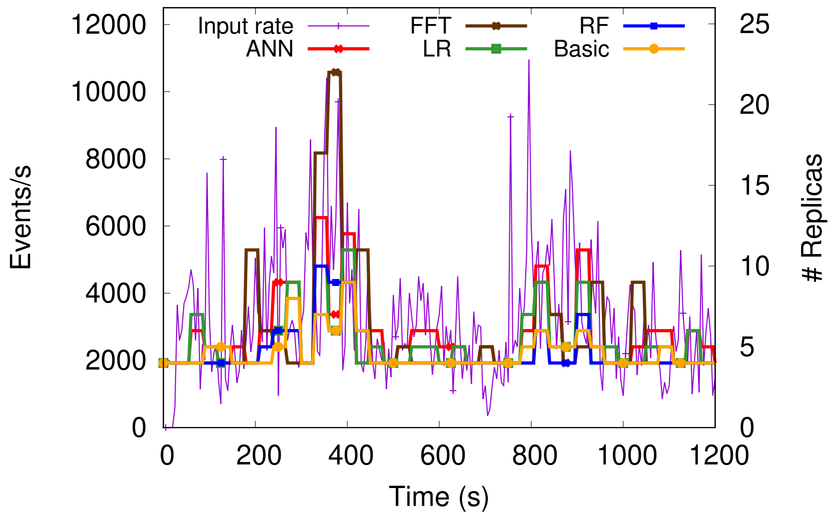
Pred. Model	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)	Error Est. Input	Error Est. Replica
ANN	0.515	0.381	0.998	446.840	0.294	0.872
FFT	0.498	0.337	0.995	397.140	0.367	1.674
LR	0.561	0.350	1.000	464.010	0.216	0.714
RF	0.608	0.436	0.975	545.910	0.112	0.583
Basic	0.604	0.511	0.984	487.600	0.090	0.738

Table 7: DNS scenario metric values.

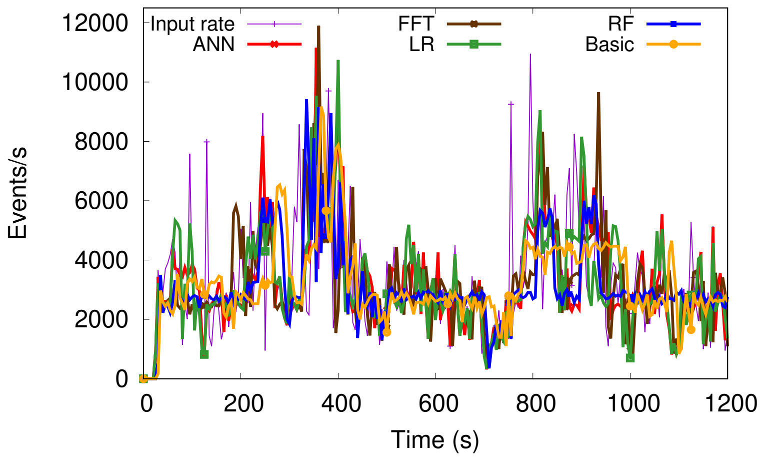
4.6.3. Distributed system logs

We deployed an application with linear topology as shown in Figure 14, composed of four operators for parsing and determine events based on distributed system logs. For the experiments, we use the dataset presented in [30] and fixed $r_{over} = 32$ (i.e., $r_i = 8$).

Table 8 summarizes the obtained results. The processing capacity of PA-SPS is once again confirmed, where each proposed model has a none or a negligible difference of processed events and the highest difference percentage of events processed is around of 1.52% (ANN). Basic presents the best performance, both in terms of resource usage and latency. Figure 15(a) shows the amount of used replicas, where we can observe that the amount used by Basic does not vary much. Although there are high peaks of the input rate



(a) Total number of replicas.



(b) Throughput of *Storm*.

Figure 13: Comparison with different predictive models used DNS traffic.

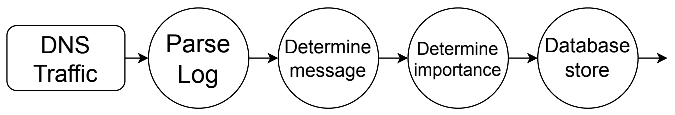


Figure 14: Log application in *PA-SPS*.

(see Figure 15(b)), as in $t = 700$, they are short for periods. Thus, it is more appropriate to use a constant amount of replicas rather than to adapt the SPS many times according to the input rate behaviour.

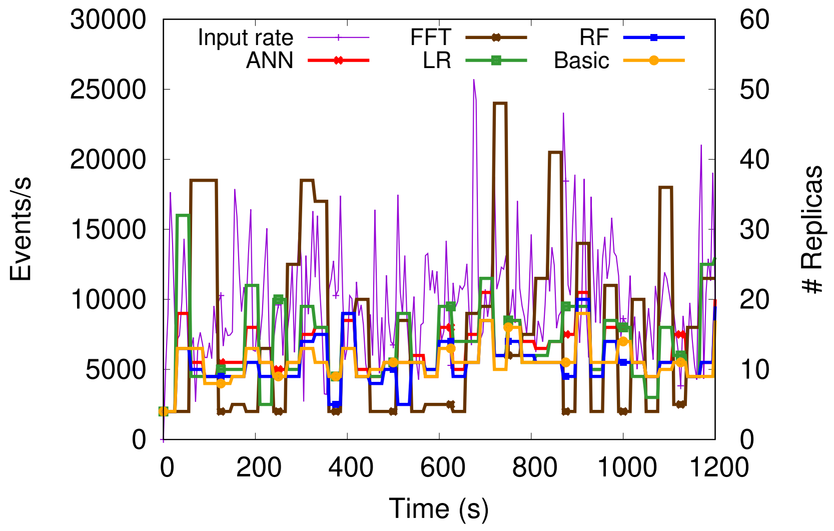
Pred. Model	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency (ms)	Error Est. Input	Error Est. Replica
ANN	0.603	0.236	0.987	905.290	0.355	0.537
FFT	0.503	0.572	1.000	9184.060	0.746	1.191
LR	0.565	0.252	0.994	1021.800	0.412	0.413
RF	0.661	0.335	0.998	1673.560	0.243	0.394
Basic	0.655	0.306	0.989	855.970	0.250	0.449

Table 8: Log scenario metric values.

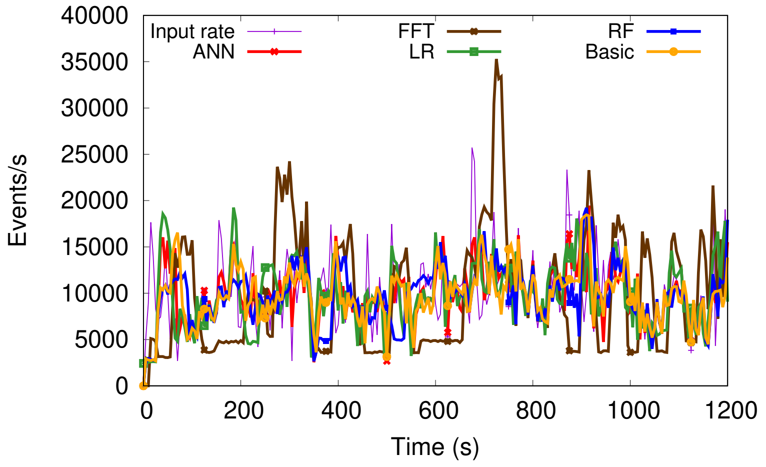
4.7. Discussion

PA-SPS presents better performance than the other two adaptive SPSs (*DABS* or *PSPS*), confirming its ability to dynamically adapt the number of replicas based on the variation of input behavior. Experimental results showed a 17.2% increase in the total number of events processed and an 83.01% reduction in latency.

We also observed that the most appropriate predictor model depends on the scenario (e.g. the input rate behavior, the application, the dataset,etc.). For instance, if the input rate behaviour is stationary, time series based models such as *FFT* may perform better. Contrarily, if the input rate behaviour is too dynamic, it would be better a simple model, such as *Basic*, whose number of replicas does not vary much since, in this case, events queued in up-spike periods could be processed in down-spike periods. Finally, it is also important to take into account cost/performance trade-off. Models such as *ANN* that overestimate the number of replicas, on the one hand, increase



(a) Total number of replicas.



(b) Throughput of *Storm*.

Figure 15: Comparison with different predictive models used log traces.

the costs associated with the system but, on the other hand, improve performance.

5. Related Work

In this section, we focus on existing works of the literature that, similar to ours, propose predictive SPS solutions.

DABS-Storm, a congestion prevention SPS, is presented in [7]. Its aim is to reduce the degradation of the accuracy of the results. To this end, 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. Such a metric that estimates the level of activity is defined by predicting the system input by using a regression function as well as taking into account pending events. The capacity of the operators is also estimated, considering both the physical capacity of the machine where the operator is located and latency of the system. As DABS-Storm has been implemented in Storm, its operators reconfiguration approach carries the drawback of Storm reconfiguration downtime cost, contrarily to PA-SPS that avoids it with the pre-allocated pool of inactive replicas.

The authors in [31] propose a predictive model implemented in Borealis SPS [32], taking into account not only the input rate as a metric, but also the capacity of the nodes as well as data processing complexity. Then, the model provides an equation that characterizes the workload of the system and determines the amount of required parallelism for processing events. Therefore, its objective is both the balance of the workload between the nodes and the reduction of latency. Although the system is capable of scaling-out, it does not perform scale-in, so it does not consider the reduction of allocated resources that PA-SPS provides.

ELYSIUM [28] 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. Unlike PA-SPS, ELYSIUM was not been evaluated with a real prototype integrated in Storm.

Based on look-ahead approach, PLAStiCC is a predictive scheduling proposed in [33]. Its model analyzes the system performance through the balance of resource overload. Furthermore, as it has been conceived to run on clouds, allocated resources can have different costs. Therefore, the model considers not only the workload of the system, but also the costs associated with the increase in resources. Contrarily to our experiments which were conducted on the public cloud GCP with real Twitter traces, PLAStiCC uses for evaluation the cloud simulator CloudSim [34], as well as synthetic dataflows.

The Elastic-PPQ SPS [35] proposes to analyze the system at short and medium/long terms. The first one performs an analysis on the events that arrive in a time interval while the second one takes into account longer periods to perform a more complex analysis, using Fuzzy Logic Controller. For this purpose, an autonomous system, based on QoS, manages the system resources according to a runtime strategy, which considers the complexity of the system components. In this way, the parallelism of the tasks, associated with a set of threads, can increase or decrease. For the evaluation and validation of system load analysis, both synthetic and real data were used. Although the solution is quite robust, since it is implemented in FastFlow [36] framework, its focus is more on high performance processing than on distributed data processing.

In [37], the authors propose a hierarchical decentralized adaptive SPS in Apache Storm, using the MAPE model to design the solution. Regarding scaling policy, the adopted 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. PA-SPS does not present such an overhead since inactive replicas are pre-allocated at the beginning of the SPS execution.

The predictive MEAD SPS [38] was implemented in Flink [39]. Operator auto-scaling takes place based on Markovian Arrival Processes approach, where the system load is analysed according to a queuing model. The SPS proposes a MAPE-K for the control flow. Evaluation experiments use both synthetic and real environments. However, even if the authors state that the MEAD supports operators scaling-out, such a feature has not been implemented. On the other hand, similarly to PA-SPS and other works that use Flink, such as [40], reconfiguration does not induce performance degradation.

A SPS adaptation model, which minimises system reconfiguration costs, is proposed in [41]. It 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, the model decides whether it is necessary to modify the amount of resources, considering the cost of such an adaptation.

6. Conclusion

This work proposes PA-SPS, a predictive Storm-based SPS model, which dynamically adapts the active number of operator replicas according to the behaviour of the input data. Based on a MAPE model, our solution predicts the number of operator replicas using statistics collected from operators. To this end, we defined a set of equations and predictive models of the input rate as well as a Load-Aware grouping strategy which is based on current replicas load.

Compared to both DABS or PSPS, evaluation results confirm the effectiveness of the dynamic replica adaptation of PA-SPS. In the experiments, latency decreases by 74.44% and saved resources increase 19.94%, when compared to DABS, and latency decreases by 83.06%, when compared to PSPS. On the other hand, we observed that the most appropriate predictor model depends on the type of input rate behavior.

As future work, we are going to evaluate PA-SPS with various benchmark datasets, such as [42] or [43]. Their use will enable us to characterize the different input rate behaviors and define the most appropriate predictive model. Given the design and implementation of PA-SPS, the system accepts the use of new models as Markov Chain [44] or Wavelet [45] providing the possibility of other predictive solutions. We also plan to evaluate PA-SPS using stateful operators and then propose an operator state replication model.

7. Acknowledgement

This work was funded by the 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 Nº11190314, Chile. The authors acknowledge the support of the French Agence Nationale de la Recherche (ANR), under grant ANR-22-CE25-0008-01 (SkyData project).

References

[1] A. Özal, A. Ranganathan, N. Tatbul, Real-time route planning with stream processing systems: a case study for the city of lucerne, in: GIS-IWGS, ACM, 2011, pp. 21–28.

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

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

[4] A. Vogel, Self-adaptive Abstractions for Efficient High-level Parallel Computing in Multi-cores, Ph.D. thesis, University of Pisa, Italy, 2022.

[5] D. Griebler, Domain-specific language & support tools for high-level stream parallelism, Ph.D. thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil, 2016.

[6] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, B. Sericola, Efficient key grouping for near-optimal load balancing in stream processing systems, in: F. Eliassen, R. Vitenberg (Eds.), Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015, ACM, 2015, pp. 80–91.

[7] R. K. Kombi, N. Lumineau, P. Lamarre, N. Rivetti, Y. Busnel, Dabs-storm: A data-aware approach for elastic stream processing, Trans. Large Scale Data Knowl. Centered Syst. 40 (2019) 58–93.

[8] D. Wladdimiro, L. Arantes, P. Sens, N. Hidalgo, 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), 2022, pp. 120–129.

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

[10] H. Andrade, B. Gedik, D. Turaga, Fundamentals of Stream Processing: Application Design, Systems, and Analytics, Cambridge University Press, 2014.

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

[12] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, M. Serafini, The power of both choices: Practical load balancing for

distributed stream processing engines, in: ICDE, IEEE Computer Society, 2015, pp. 137–148.

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

[14] Y. Xing, S. B. Zdonik, 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.

[15] J. Xu, Z. Chen, J. Tang, S. Su, T-storm: Traffic-aware online scheduling in storm, in: 2014 IEEE 34th International Conference on Distributed Computing Systems, IEEE Computer Society, 2014, pp. 535–544.

[16] 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.

[17] P. Hunt, M. Konar, F. P. Junqueira, B. C. Reed, Zookeeper: Wait-free coordination for internet-scale systems, in: USENIX Annual Technical Conference, USENIX Association, 2010, p. 11.

[18] D. Wladdimiro, L. Arantes, P. Sens, N. Hidalgo, A multi-metric adaptive stream processing system, in: NCA, IEEE, 2021, pp. 1–8.

[19] L. Wynants, W. Bouwmeester, K. Moons, M. Moerbeek, D. Timmerman, S. Van Huffel, B. Van Calster, Y. Vergouwe, A simulation study

of sample size demonstrated the importance of the number of events per variable to develop prediction models in clustered data, Journal of Clinical Epidemiology 68 (2015) 1406–1414. doi:https://doi.org/10.1016/j.jclinepi.2015.02.002.

[20] D. C. Montgomery, E. A. Peck, G. G. Vining, Introduction to linear regression analysis, John Wiley & Sons, 2021.

[21] H. J. Nussbaumer, H. J. Nussbaumer, The fast Fourier transform, Springer, 1981.

[22] J. Herzen, F. Lässig, S. G. Piazzetta, T. Neuer, L. Tafti, G. Raille, T. V. Pottelbergh, M. Pasieka, A. Skrodzki, N. Huguenin, M. Dumonal, J. Koscisz, D. Bader, F. Gusset, M. Benheddi, C. Williamson, M. Kosinski, M. Petrik, G. Grosch, Darts: User-friendly modern machine learning for time series, J. Mach. Learn. Res. 23 (2022) 124:1–124:6.

[23] M. Riedmiller, A. Lernen, Multi layer perceptron, Machine Learning Lab Special Lecture, University of Freiburg (2014) 7–24.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

[25] S. J. Rigatti, Random forest, Journal of Insurance Medicine 47 (2017) 31–39.

[26] A. Gruzd, P. Mai, COVID-19 Twitter Dataset, 2020. URL: https://doi.org/10.5683/SP2/PXF2CU. doi:10.5683/SP2/PXF2CU.

[27] P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, D. A. Patterson, Characterizing, modeling, and generating workload spikes for stateful services, in: SoCC, ACM, 2010, pp. 241–252.

[28] F. Lombardi, L. Aniello, S. Bonomi, L. Querzoni, Elastic symbiotic scaling of operators and resources in stream processing systems, IEEE Trans. Parallel Distrib. Syst. 29 (2018) 572–585.

[29] M. MontazeriShatoori, L. Davidson, G. Kaur, A. H. Lashkari, Detection of doh tunnels using time-series classification of encrypted traffic, in: IEEE Intl Conf on Dependable, Autonomic and Secure Computing, IEEE, 2020, pp. 63–70.

[30] S. He, J. Zhu, P. He, M. R. Lyu, Loghub: A large collection of system log datasets towards automated log analytics, CoRR abs/2008.06448 (2020). URL: https://arxiv.org/abs/2008.06448. arXiv:2008.06448.

[31] C. Balkesen, N. Tatbul, M. T. Özsu, Adaptive input admission and management for parallel stream processing, in: DEBS, ACM, 2013, pp. 15–26.

[32] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. B. Zdonik, The design of the borealis stream processing engine, in: CIDR, www.cidrdb.org, 2005, pp. 277–289.

[33] A. G. Kumbhare, Y. Simmhan, V. K. Prasanna, Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds, in: CCGRID, IEEE Computer Society, 2014, pp. 344–353.

[34] R. N. Calheiros, R. Ranjan, C. A. F. D. Rose, R. Buyya, Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services, CoRR abs/0903.2525 (2009).

[35] G. Mencagli, M. Torquati, M. Danelutto, Elastic-ppq: A two-level autonomic system for spatial preference query processing over dynamic data streams, Future Gener. Comput. Syst. 79 (2018) 862–877.

[36] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: high-level and efficient streaming on multi-core, Programming multi-core and many-core computing systems, parallel and distributed computing (2017).

[37] V. Cardellini, F. L. Presti, M. Nardelli, G. R. Russo, Decentralized self-adaptation for elastic data stream processing, Future Gener. Comput. Syst. 87 (2018) 171–185.

[38] G. R. Russo, V. Cardellini, G. Casale, F. L. Presti, MEAD: model-based vertical auto-scaling for data stream processing, in: CCGRID, IEEE, 2021, pp. 314–323.

[39] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink™: Stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (2015) 28–38.

[40] H. Arkian, G. Pierre, J. Tordsson, E. Elmroth, Model-based stream processing auto-scaling in geo-distributed environments, in: 30th Inter. Conference on Computer Communications and Networks, 2021, pp. 1–10.

[41] M. Borkowski, C. Hochreiner, S. Schulte, Minimizing cost by reducing scaling operations in distributed stream processing, Proc. VLDB Endow. 12 (2019) 724–737.

[42] A. Shukla, S. Chaturvedi, Y. Simmhan, Riotbench: An iot benchmark for distributed stream processing systems, Concurr. Comput. Pract. Exp. 29 (2017).

[43] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, R. Tibbetts, Linear road: A stream data management benchmark, in: VLDB, Morgan Kaufmann, 2004, pp. 480–491.

[44] S. L. Zeger, B. Qaqish, Markov regression models for time series: A quasi-likelihood approach, Biometrics 44 (1988) 1019–1031.

[45] M. Küçük, N. Ağirali˙ oğlu, Wavelet regression technique for streamflow prediction, Journal of Applied Statistics 33 (2006) 943–960.