The Journal of Systems and Software 000 (2016) 1–12

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Self-adaptive processing graph with operator fission for elastic stream processing

Nicolas Hidalgo*, Daniel Wladdimiro, Erika Rosas

Universidad de Santiago de Chile, Av. Ecuador 3659, Santiago, Chile

ARTICLE INFO

Article history:
Received 31 October 2015
Revised 28 April 2016
Accepted 4 June 2016
Available online xxx

Keywords:
Stream processing
Self-adaptable graph
Elastic processing
Scalable processing
S4

ABSTRACT

Nowadays, information generated by the Internet interactions is growing exponentially, creating massive and continuous flows of events from the most diverse sources. These interactions contain valuable information for domains such as government, commerce, and banks, among others. Extracting information in near real-time from such data requires powerful processing tools to cope with the high-velocity and the high-volume stream of events. Specially designed distributed processing engines build a graph-based topology of a static number of processing operators creating bottlenecks and load balance problems when processing dynamic flows of events. In this work we propose a self-adaptive processing graph that provides elasticity and scalability by automatically increasing or decreasing the number of processing operators to improve performance and resource utilization of the system. Our solution uses a model that monitors, analyzes and changes the graph topology with a control algorithm that is both reactive and proactive to the flow of events. We have evaluated our solution with three stream processing applications and results show that our model can adapt the graph topology when receiving events at high rate with sudden peaks, producing very low costs of memory and CPU usage.

1. Introduction

Nowadays, the Web is characterized by the active participation of users, reflected in the large number of blogs, social networks and other Web applications (Oberhelman, 2007). Users interact and emit data that can be seen as a flow of events at a high-velocity, with a high-volume and from high-variety of sources. Processing systems aim at analyzing these types of Big Data streams created with events at Internet-scale with near real-time results restrictions.

In near real-time processing scenarios, the widely used batch-processing model MapReduce (Lin and Dyer, 2010) is not particularly suitable because storing the data before processing produces delays. On the other hand, Stream Processing Engines (SPEs) are specially designed software solutions to process unbounded streams of unstructured events in a distributed and on-line fashion. The SPE model analyzes data without the need to store the events. This processing model is useful for many domains such as trading transactions, sensors data analysis, and bank transaction analysis, among others, where event relevance is also related to its emission time.

SPEs such as S4 (Neumeyer et al., 2010), Storm[1] and Samza[2] use a graph-based model, where vertices represent processing operations over the data, and edges represent the flow of data between these operations. Processing operators can be of any complexity; however, most of them are lightweight tasks such as filtering, counting, merging, among others.

Streaming application users define the graph topology and configure the number of replicas for the operators at the beginning of the execution. Given the dynamism of Web applications interactions, changes in the flow of data may produce an overload of the topology processing operators. Overloaded operators reduce the system's performance and may produce waste of resources or information. This work improves the graph configuration at runtime, preventing operators from becoming a bottleneck. This solution enables improving performance and scalability of the processing system.

The contribution of this article is the proposal of an elastic stream processing system that monitors the state of each processing operator independently and adapts changes to the processing graph accordingly, using both a reactive and a predictive approach. Our solution is able to predict operator's load using statistical

* Corresponding author.
E-mail addresses: nicolas.hidalgo@usach.cl (N. Hidalgo), daniel.wladdimiro@usach.cl (D. Wladdimiro), erika.rosas@usach.cl (E. Rosas).

[1] http://storm.incubator.apache.org
[2] http://samza.incubator.apache.org

http://dx.doi.org/10.1016/j.jss.2016.06.010
0164-1212/

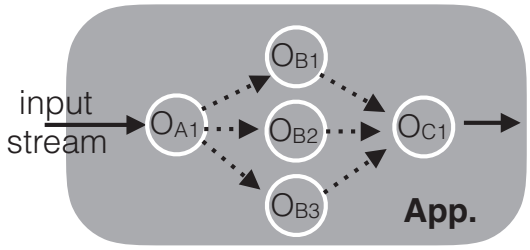


Fig. 1. SPS's processing graph.

information collected from them. We apply two simple algorithms to determine the state of the operator: short-term and mid-term algorithms. The short-term algorithm is focused on detecting peaks of traffic, while the mid-term algorithm is focused on finding patterns in traffic.

The remaining sections of this article are organized as follows: Section 2 presents some fundamentals of SPEs. Section 3 presents the design of the self-adaptable processing system, explaining the model, its components and the implemented algorithms. Section 4 shows the experiments and analysis of our proposal using different sources of events and applications. Related work on load balance and elasticity in SPEs is presented in Section 5. Finally, concluding remarks and future work is included in the last section.

2. Stream Processing Engines

SPEs are specially designed software solutions to process unbounded streams of unstructured events in a distributed and online fashion. Their processing model differs from traditional batch processing paradigm in widely-used Mapreduce (Lin and Dyer, 2010) or Hadoop (Mone, 2013) since it processes data while receiving it without requiring previous storage. Most SPEs model the applications as a graph where events flows are represented by the edges, and processing tasks (or operators), by the vertices. In the following we refer as processing graph to this representation.

Operators may be of any kind, but most of them are lightweight tasks such as filtering, counting, merge, among others. Operators receive a flow of events to process and issue processed events. Operators may receive one or more flow of events depending on the graph organization. The processing model includes a special type of operator to manage the data extraction from the source. This operator's task is to retrieve data from the source and distribute it among the operators connected to it. The data sources may provide any type of events like social interactions over a given social network, sensor measurements from a smart city application, logs entries from a network monitor, etc. Appel et al. (2012). Fig. 1 shows an example of a small processing graph that is fed with an input stream of data and has five processing operators. Of the five operators, three execute the same function (O_{B1}, O_{B2} and O_{B3}) so that the data is processed in parallel.

Operators are logically organized on the processing graph and they are deployed on physical machines. Many operators can be associated to a single processing node. The association process and the routing of events are typically performed using systems such as Zookeeper (Hunt et al., 2010), which maintains configuration information, naming, provides distributed synchronization, manages events distribution, communication and fail-over.

3. Self-adaptive processing

Although SPEs provide a flexible model to build applications, the processing graph remains static once the system is initialized. Several efforts have been made to provide graph adaptability in execution time because the volume of the stream of events is highly dynamic, usually driven by real word events. In Storm for example, the processing graph can be modeled considering different complexities, number of events, and type of events by associating a fixed number of operators to consume a given flow in parallel and perform a given task on the data. However, this decision must be made offline, and cannot be modified at runtime. A bad estimation may produce waste of resources (if we over-provision resources) or congestion and low accuracy (if we under-provision resources). Other systems, such as S4, enable associating events to a single operator based on their key value. This approach allows the programmers to avoid decisions on the number of operators to be assigned to a given task; however, this processing model also has load balance problems because of the key distribution of the events and the arrival rate of the events.

Both models suffer from load balance problems that may produce poor processing performance or low quality of results due to events loss. We provide the system with the ability to dynamically adapt its processing structure to the dynamic volume of the stream of events. In particular, we propose an elastic processing model able to assign and remove processing operators to share load. By modifying the topology of the processing graph at execution time our solution adapts its structure to the stream dynamics. Our solution is able to detect overloaded operators and creates replicas to evenly split the load among them and increase data parallelism. Balancing load by modifying the processing logic presents several advantages compared to traditional reallocations of load among the physical machines that we discuss in the following sections.

Our self-adaptive processing model was built on top of S4 processing system but it can be applied to other graph-based SPEs. On the following section we provide a brief description of the S4 processing engine.

3.1. S4 concepts

S4 (Neumeyer et al., 2010) is a general-purpose, real-time, distributed, decentralized, scalable, event-driven, modular platform that allows programmers to easily implement applications for processing continuous unbounded streams of data. S4 follows the same graph oriented processing paradigm of other SPEs. The vertices of the graph perform simple tasks on the streams of data represented by the edges. Operators connect to each other sharing their pre-processed outputs generating the processing graph (Fig. 1). S4 uses a push-processing model, where events are pushed to a given operator using a round robin scheduling by default. S4 also implements other types of scheduling, for example, basing event dispatching on their key.

S4 calls the operators Processing Elements (PEs). PEs are the basic computational units in S4. Many tasks may use standard PEs that require no additional coding; however, it is also possible to customize your PEs in order to perform more specific tasks. In S4, PEs are uniquely identified by their functionality and the type of events they consume. Additionally, a special class of PEs is also available: the Adapters that convert external streams into streams of S4 events.

Processing Nodes (PNs) are the hosts for the PEs. They are responsible for listening to events, executing operations on the incoming events, dispatching events using the communication layer, and issuing output events. Fig. 2 shows the structure of the PNs. The event listener sends messages to the Processing Element Container (PEC) that selects the responsible PEs for that particular event. PEs process the events and send them to the dispatcher that determines their next destination. Finally, supported by the communication layer, the event is issued and routed to the following PN.

Please cite this article as: N. Hidalgo et al., Self-adaptive processing graph with operator fission for elastic stream processing, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.010

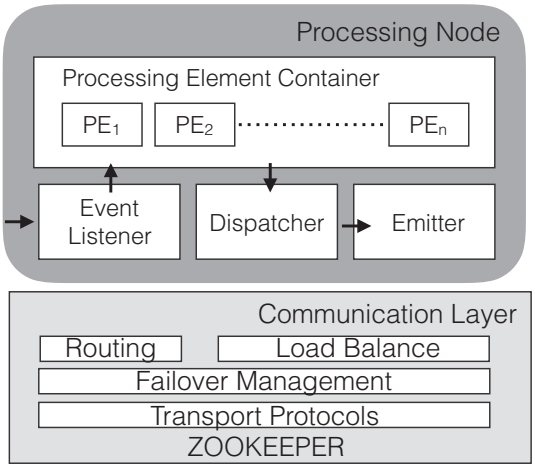


Fig. 2. S4 processing node structure.

S4 uses Zookeeper (Hunt et al., 2010) on the communication layer, which is a distributed coordination service for distributed applications. Zookeeper enables communication among nodes, load balance, and failover.

3.2. Modeling load

We collect information about the state of each operator to create a self-adaptable model. Operators that have different functions and receive different types of events have different complexities and processing times, which depends on the number of events and the available hardware resources. We evaluate all the identical operators (replicas) together in order to carry out changes in as many of them as required, to cover the amount of events they are receiving.

We have computed a load metric based on the number of events that arrive to the set of replicas that perform the same function and the amount of events this set of replicas can process. We take into account that the load changes in time and we associated these values to a time-window. Eq. 1 shows our metric, where the load L for a set of replicas of the operator O_B is computed in a time-window *tw* as the fraction between the number of events that arrive to their queues in that period of time, A_B, and the average number of events that the set of replicas are able to process when they are in a busy state, P_B. For simplicity, we assume that the value P_B does not changes over time.

$$L_B^{tm} = \frac{A_B^{tw}}{P_B} \tag{1}$$

We consider three possible states for the set of replicas of an operator based on its load level: *underloaded, balanced*, or *overloaded*. Then, a set of replicas of and operator O_X at a time-window *tw* are overloaded when its load value is $L_X^{tw} > upper\_threshold$, are underloaded when $L_X^{tw} < lower\_threshold$, and are balanced for any other value in between.

3.3. Adapting the graph

An adaptable processing model requires to dynamically modifying its structure in order to improve resource utilization. In our model, we adapt the processing graph by adding/reducing the number of replicas of an operator. Our focus is to improve processing performance by avoiding bottleneck operators and therefore also improving the utilization of the whole system.

Performance degradation is proportional to the connection degree of the operators in the graph; therefore, solving bottleneck problems is crucial for maintaining the system scalability. Replication is an efficient technique to deal with overloaded operators. When an operator becomes overloaded one or more new replicas are deployed in order to increase the service rate, improving performance. This optimization is called *fission* (Hirzel et al., 2014). Fig. 3 presents the replication of an overloaded operator O_B that creates n replicas in order to share its load parallelizing data processing. In case the deployed operators become underloaded, based on the definition previously presented, the number of replicas of the operator is reduced according to its current utilization.

When replicas are created the events are scheduled among them sending the events towards the replica that has the smallest queue of events. We have selected this type of scheduler so that overloaded operators that have a longer queue of events waiting for processing stop receiving events until the load problem is resolved. We do not constraint the number of replicas to be deployed, assuming there are enough processing resources to deal with the flow of events, which is the case in applications running over the cloud.

This model can be directly applied to state-less operators (operators that do not save a state) since the replicas can operate independently on the events. However, the application of fission for state-full operators requires the inclusion of an additional operator that consolidates the state of the operator's replicas. Details of this complementary solution can be found in Zeitler and Risch (2010); Schneider et al. (2012).

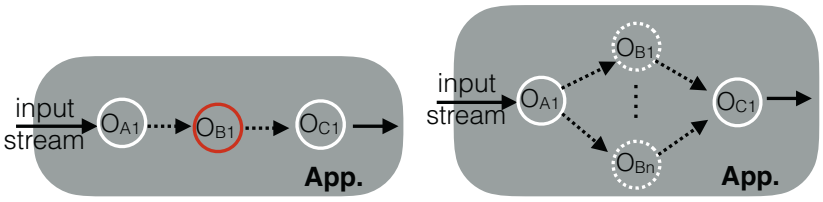
The efficiency of the approach depends on the decision of how many replicas we need to deploy to optimize graph's configuration for a given traffic. To obtain the answer we model our solution based on the Monitor, Analyze, Plan, Execute (MAPE) model (Kephart and Chess, 2003) commonly used in autonomic computing to control a system. The components of the system are presented on Fig. 4. Each module performs a specific task and is deployed as an operator of the SPE, i.e. the solution is modeled as a parallel application, whose data source comes from the operators' statistics. This approach allows the system to maintain the good properties of the SPE. From now on, we refer to the operators of our self-adaptive model as *modules* , and to the application processing operators that build the graph topology, as *operators*.

Monitor

Following the MAPE model, the first module is aimed to monitor and collect statistics from the operators. These statistics are required to estimate the load of the set of O_B operator replicas at each time-window to be monitored (Eq. 1). Each replica locally counts the number of events received in a time-window in order to compute the number of events that have arrived and the monitor requests this value periodically. Then, A_B is the sum of the number of events that arrive to each replica of operator O_B. In case of S4, all the events that one operator sends towards any of the replicas of another operator, arrive into one input queue, facilitating the monitor task.

The value P_B, which is the average number of events that the set of replicas are able to process when they are in a busy state, is computed at the beginning of the processing phase, considering the processing power of one replica through benchmarking. That is, the systems overloads the operator in order to compute the average amount of events that it is able to process in a period of time. The P_B for the set of n replicas of the operator O_B is the value measured in one replica times n. This approach does not hold for a heterogeneous system but it can be extended by performing benchmarking of all operators in all the different type of machines available. Moreover, this value should be computed more often if the system shares hardware resources with other applications that make it compete for processing power.

Please cite this article as: N. Hidalgo et al., Self-adaptive processing graph with operator fission for elastic stream processing, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.010



(a) Without Replicas (b) With Replicas

Fig. 3. Operator's replication process.

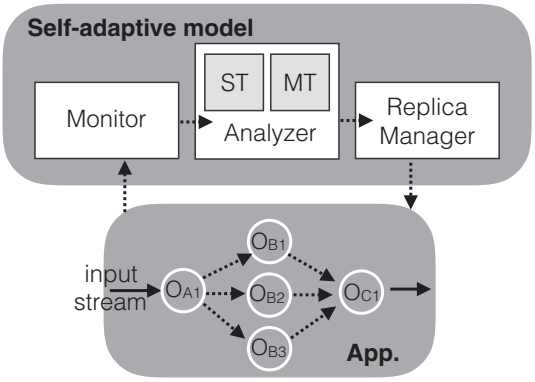


Fig. 4. Self-adaptive architecture.

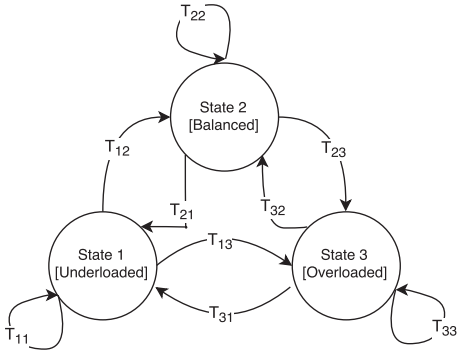


Fig. 5. Markov chain model for the mid-term algorithm.

With this information the monitor calculates the load using Eq. 1. The value *tw* is a parameter of the system, which must be set shorter enough to cope with real-time traffic variations while limiting the communication overhead. The monitor sends the aggregated load of the operators to the next module informing about their state.

Analyzer

This module is oriented to determine the state of the operator based on the information provided by the monitor module. This module is composed of two algorithms that classify the state of the operator. The first algorithm is focused on detecting sudden peaks of traffic, while the second is focused on detecting mid-term traffic variations similar to others found in the past.

The analyzer module applies the two aforementioned algorithms. The short-term algorithm is applied every t_s time and the mid-term algorithm is applied every t_m time, both being system parameter values. The time-window that defines the short-term algorithm t_s must be shorter than the time-window t_m when the mid-term algorithm is applied, to tackle the different traffic behaviors for which they were designed. The mid-term algorithm requires to collect data to provide accurate predictions.

Short-term Algorithm (ST):

Due to the dynamic behavior of the event flows, sudden peaks of events can overload operators on short-time periods. To tackle this problem we propose a short-term algorithm that evaluates operators load over short-time periods in order to detect traffic changes. The short-term algorithm needs to be fast and efficient reducing load. In order to accomplish such requirement, the algorithm classifies the operator's state based on the information provided by the monitor module. Operator state can be: *underloaded, balanced*, or *overloaded*. Short-term algorithm defines thresholds to carry out the classification of the node state. We consider the operator is overloaded when its load value is greater than an *upper_threshold*, underloaded when is smaller than a *lower_threshold*, and balanced for any other value in between. The short-term classification algorithm is presented in Algorithm 1.

Algorithm 1 Short-term algorithm.

Require: load L of the operator i (L_i).
Ensure: operator i current state Δ (Δ_i).
1: **if** $L_i >$ upper_threshold **then**
2: **return** Δ_i : *"overloaded"*
3: **else if** $L_i <$ lower_threshold **then**
4: **return** Δ_i :*"underloaded"*
5: **else**
6: **return** Δ_i :*"balanced"*
7: **end if**

Mid-term Algorithm (MT):

The second algorithm proposed predicts the operator's future load based on the past event stream behavior. In order to predict operator's load, we propose a mid-term algorithm that uses a Markov chain model (Ching and Ng, 2006). Markov chains have been widely used in literature to model traffic behavior (Gong et al., 2010; van der Veen et al., 2015; Yu et al., 2003). We propose a Markov chain (Ching and Ng, 2006) defined by three possible states: underloaded, balanced, and overloaded. Fig. 5 presents the Markov chain model. In order to define the transition probability we use the input samples (L) from a previous time-window also called the history h of the operator. The history of the operator is used evaluating the state transitions over the analyzed samples to define the transition matrix. The transition matrix allows establishing the stationary distribution in Markov chain model. The stationary distribution defines the probability of a given operator to remaining in a given state. The stationary distribution computation based on the Chapman-Kolmogórov (Papoulis, 1984) equation and the input values for the equation were established based on the work presented in Gong et al. (2010).

Algorithm 2 presents the mid-term algorithm used to predict the future operator state based on the Markov chain. The algorithm estimates the probability for the next time-window to be at one of

Algorithm 2 Mid-term algorithm.

Require: history h of operator i.
Ensure: predict operator i future state Δ_i^+.
1: $TM \leftarrow$ `createTransitionMatrix`(h)
2: $< e_1, e_2, e_3 > \leftarrow$ `getStationaryDistribution`$(TM, \#iterations)$
3: Δ_i^+ = `max`$\{< e_1, e_2, e_3 >\}$
4: **if** $\Delta_i^+ > \Delta_i + S$ **then**
5: **return** Δ_i^+
6: **else**
7: **return** $\Delta_i^+ : \Delta_i$
8: **end if**

the three possible states (overloaded, underloaded, stable). In order to reduce uncertainty in the state selection, we establish a threshold S to ensure that the next possible state is at least $S\%$ more likely to be selected than the current state. Otherwise, the system remains in its current state.

Replica manager

The final module is in charge of defining the number of replicas required for each operator, the equivalent to the executer of the MAPE model. Using the information provided by the analyzer module, the replica manager creates or removes replicas in the graph topology.

Algorithm 3 presents the replica manager behavior. In order to apply changes based on the short-term algorithm, we define that the operator must remain in the same state for n periods. This decision provides stability to the system and prevents the impact of oscillating behaviors.

Algorithm 3 Replica Manager: adding/removing replicas.

Require: Δ_i (state of operator i).
Require: algorithm window: {ST or MT}.
Ensure: replicas for operator i.
1: **if** Short-term (ST) **then**
2: **if** Δ_i = *"overloaded"* for W windows **then**
3: **return** *"add R replicas"*
4: **else if** Δ_i = *"underloaded"* for W windows **then**
5: **return** *"remove R replicas*
6: **end if**
7: **else if** Mid-term (MT) **then**
8: **if** Δ_i = *"overloaded* **then**
9: **return** *"add R' replicas"*
10: **else if** Δ_i = *"underloaded"* **then**
11: **return** *"remove R' replicas"*
12: **end if**
13: **end if**

The mid-term algorithm defines the best configuration for the next time-window t_m, larger than t_s. For this reason, setting-up the best configuration implies adding or removing a greater number of replicas from the processing graph than when applying the short-term algorithm. If the algorithm predicts an overloaded or underloaded operator, the module immediately creates or removes a predefined number of replicas, respectively.

4. Experiments

In this section we present the evaluation of our proposed model in terms of throughput, memory and CPU costs. We implemented our model over S4 and three different stream processing applications, comparing our system with scenarios with a static number of replicas. All experiments were executed in a machine with 32 cores Intel Xeon CPU E5-2650 v2 2.60 GHz with 128 GB RAM.

Table 1
Default parameters for experiments.

Parameter	Value	Module
t_w	1s	Monitor
t_s	5s	Analyzer
t_m	100s	Analyzer
upper_threshold	1.0	Analyzer ST
lower_threshold	0.5	Analyzer ST
S	0.2	Analyzer MT
R	1	Replica manager
W	2	Replica manager
R'	5	Replica manager

The baseline scenarios are:

- Static Scenario 1 (SS1): Scenario where there are no replicas of the operators.
- Static Scenario 2 (SS2): Scenario where the amount of replicas for each operator is the minimum necessary to cope with the data rate of the experiments, without loosing events or queuing.
- Static Scenario 3 (SS3): This scenario is an in-the-middle configuration, where we define the number of replicas as 50% of the replicas defined for SS2.

The default parameters selected for the elastic approach are defined in Table 1. The operators send their values to the monitor module in a time-window of 1 s. With these values, the module analyzer executes the short-term algorithm every 5 s and the mid-term algorithm every 100 s. Hence, the latter has 100 data samples to predict the next time window, value that has been used in other applications (Gong et al., 2010). We discuss the impact of the selected thresholds and number of replicas in Section 4.5.

4.1. Datasets

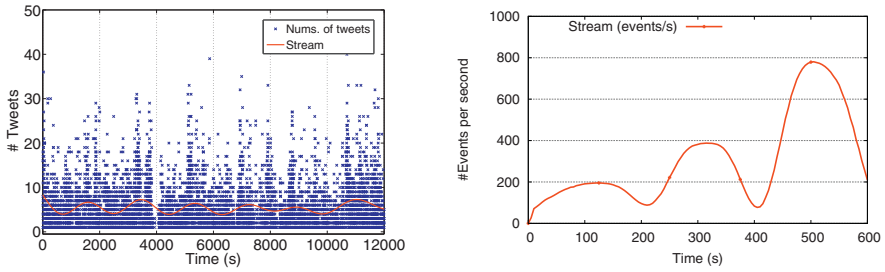
We used two datasets as input stream of data:

- *Twitter*: Stream of data composed by 4.5M tweets collected during Chilean earthquake of 2010. Fig. 6(a) shows the timestamped events and the adjusted Fourier model to obtain the stationary behaviour with 8 coefficients in the series. We extracted part of this series and built a synthetic stream with different amplifications of original timestamped version adding peaks of events. Fig. 6(b) shows the synthetic flow that defines the data rate for the experiments in a range from 0 to around 800 tweets per second. Twitter receives today an average of 6000[3] tweets per second; however, we had to adjust these values to the resources of one machine.
- *News phrases*: Stream of data composed of quotes and phrases from online news in April 2009 (Leskovec et al., 2009). Fig. 7(a) shows the timestamped events per second obtained. Similarly to the first dataset, we selected part of the real data rate distribution and amplified the amount of events per second.

4.2. Application 1

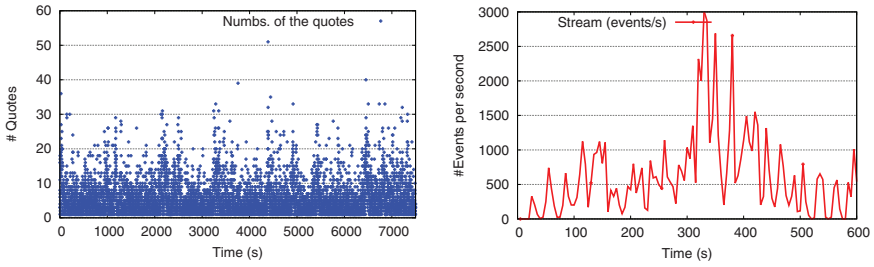
The first application is composed of 4 operators organized in a pipeline style: one operator is applied over the events after the other. Fig. 8 shows the topology of the stream processing application and uses the *Twitter* dataset as input stream source. The PE1 removes stopwords from the tweet and the PE2 detects language of the tweet using the library Apache Tika.[4] Then, the PE3 counts

[3] http://www.internetlivestats.com/twitter-statistics/
[4] https://tika.apache.org/



(a) Real Twitter stream (b) Synthetic Twitter stream

Fig. 6. Twitter stream from Chilean earthquake 2010.



(a) Real News stream (b) Synthetic News stream

Fig. 7. News stream with quotes and phrases April 2009.

Fig. 8. Topology of application 1

the words in the tweet that are contained in a bag of words previously defined. Finally, the PE4 stores the data in the non-relational database MongoDB[5].

Figs. 9 and 10 show the results of the experiments performed for Application 1, with an execution of 10 min and 60 min, respectively. The small execution window narrows the data rate distribution, in comparison with the larger execution window, producing sudden peaks of data.

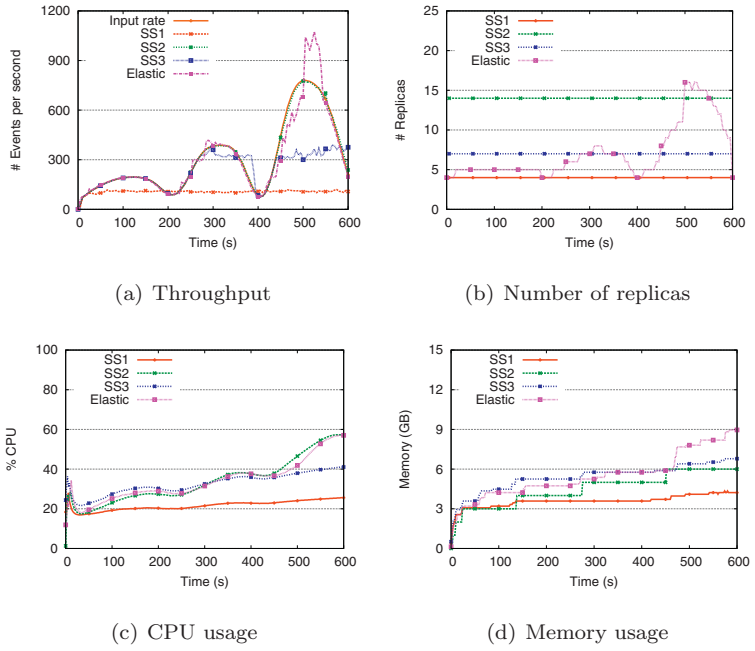
We can observe in Fig. 9(a) the throughput in all the scenarios, together with the input rate of the stream processing application. The SS2 has the ideal behaviour, processing all the incoming events. In the other extreme, the SS1 maintains a low throughput since some of the PEs become bottlenecks of the system. Our elastic approach can gracefully deal with the increasing amount of events arriving at second 300 and 500. At second 500 the mid-term algorithm intervenes increasing the number of replicas in order to deal with the events flowing. We can observe that the amount of processed events is higher than the number of incoming events around second 500 s, which occurs since the events that the operators are not able to process are stored in their queues. In the case of the 60 min execution (Fig. 10(a)), it is not necessary a sudden increase in the number of replicas, since the event rate increases at a slower pace. The amount of replicas in that case remains lower, since there is no need to deal with queuing of events (Fig. 10(b)).

In terms of resources usage, CPU and memory behave similar in the different scenarios. CPU usage is higher when the stream processing application is able to process more events (Fig. 9(c) and Fig. 10(c)). In the case of the memory, on the other hand, more complex applications produce larger queues in the system spending more memory. In Fig. 9(d) the memory of the elastic approach increases to deal with a event rate of more than 1,000 events per second after second 500. In Fig. 10(d), this was not the case, since the amount of events on the queues was smaller and the elastic approach results had a similar memory usage to the other static configurations. Finally, analyzing CPU and memory usage, we can conclude that there are no major costs in the implementation of our elastic approach, since the results are similar to the static scenarios that do not include the extra modules that our system implements.

4.3. Application 2

Fig. 11 shows the topology of the Application 2, which is composed of 5 PEs. This application has operators with two inputs and two outputs and also uses the *Twitter* dataset. Similarly to the first application, the PE1 removes stop-words from the tweet. Then, the PE2 uses a lightweight library to detect language of the tweet and splits the stream flow into two flows by sending to the PE3 the tweets detected to be written in Spanish and to the PE4 the remaining tweets. The PE4 performs a second, more accurate but slower, Spanish language detector using a dictionary. If another Spanish tweet is detected, it is also sent to the PE3, the others go directly to the last PE. The PE3 counts the words in the tweet that are contained in a bag of words previously defined. The last PE stores the data into MongoDB, receiving as input the stream from the PE3 and the PE4.

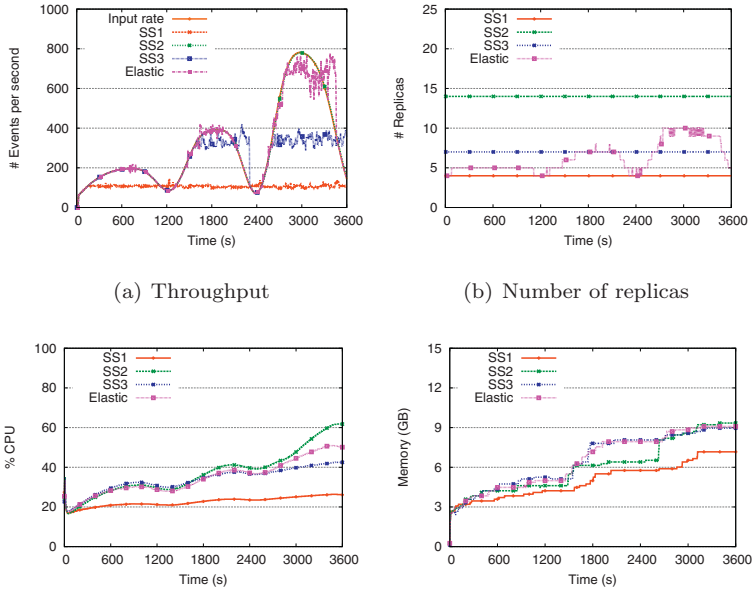
[5] https://www.mongodb.org/



(a) Throughput (b) Number of replicas

(c) CPU usage (d) Memory usage

Fig. 9. Application 1 results with execution time of 10 min.



(a) Throughput (b) Number of replicas

(c) CPU usage (d) Memory usage

Fig. 10. Application 1 results with execution time of 60 min.

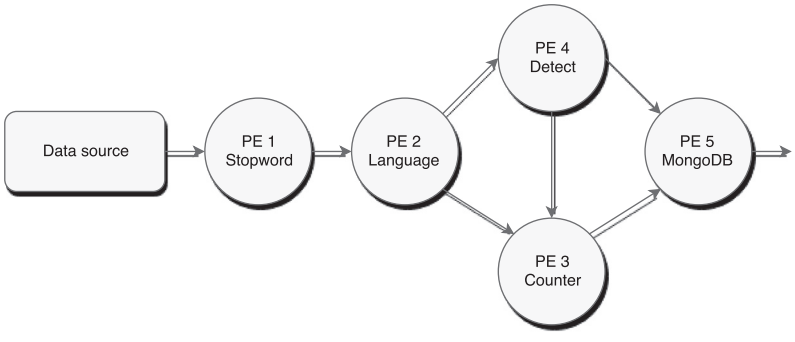
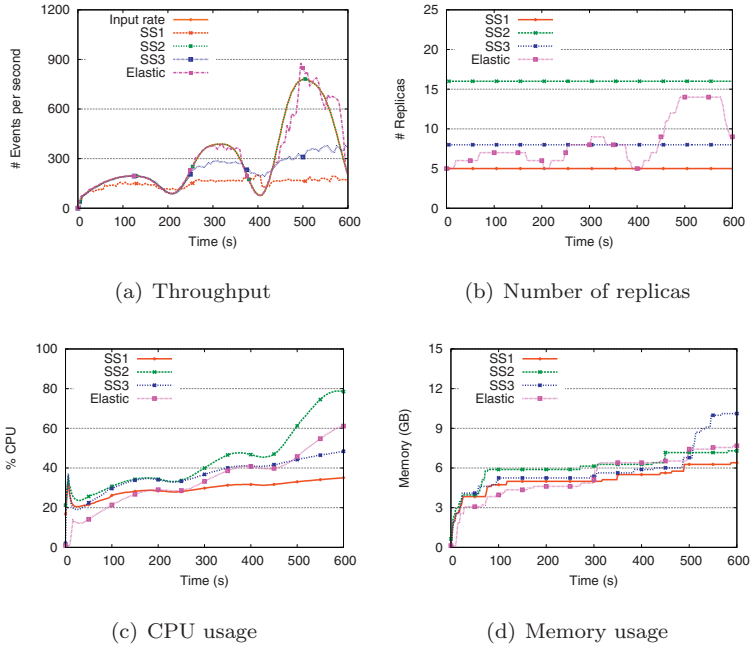


Fig. 11. Topology of application 2



(a) Throughput (b) Number of replicas

(c) CPU usage (d) Memory usage

Fig. 12. Application 2 results with execution time of 10 min.

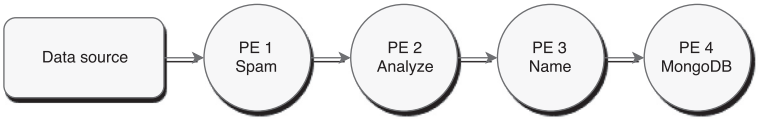


Fig. 13. Topology of application 3

Fig. 12 shows the results of the experiments for 10 minutes of execution time. Results are similar to those obtained on Application 1, but the flow division acted as an extra replica and released pressure from the bottleneck PE3. In this case, the mid-term algorithm was not required to drastically increase or decrease the number of replicas in the elastic model (Fig. 12(b). We can observe that the approach can be equally applied to graph topologies with operators that have more than one input and more than one output.

4.4. Application 3

Application 3 is composed of 4 operators, as shown in Fig. 13. This application uses the dataset of *News phrases*. The PE1 is a spam detector that uses a classifier created with the data mining software Weka[6] that is based on the work presented in Hidalgo et al. (2012). The dataset used as training set is a SMS list created by Tagg (2009). The PE2 is a sentiment classifier that considers a text positive or negative using the library SentiStrengh (Thelwall et al., 2010). Then, the PE3 detects entities inside a text using a tool for natural language processing called OpenNLP.[7] The last PE stores the data in the non-relational database MongoDB.

Fig. 14 shows the results of the experiments. Fig. 14(a) contrasts the input rate with the throughput for the different analyzed scenarios. We can observe that the SS2 is able to cope with all the traffic with 30 PEs (Fig. 14(b)). In the other hand, our elastic approach increases permanently the number of replicas in order to cope with the increasing amount of events. The mid-term algorithm increases and decreases the amount of replicas (at seconds 400 and 500 respectively), after learning about the previous peaks of data around second 300. However, given the size of the peak of events (near 3000 events per second around second 300, for example) some events are definitively lost.

CPU usage follows the number of processed events as in the other applications (Fig. 14(c)) and the memory is higher in the applications where there is more queuing of events.

Table 2
Total number of events processed.

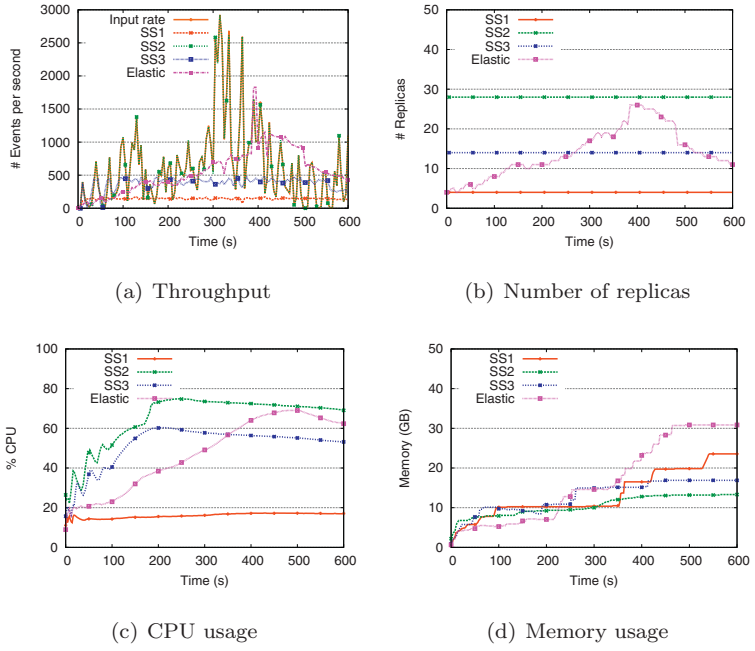
Application	Time	SS1	SS3	Elastic	SS2
App 1	10 min	63,918	144,007	185,353	187,333
App 1	60 min	380,704	873,099	1,119,554	1,224,341
App 2	10 min	90,867	139,947	186,843	187,660
App 2	60 min	534,875	852,410	1,096,732	1,112,899
App 3	10 min	84,575	227,031	329,574	382,713

4.5. Comparison and analysis of parameter selection

A summary of the number of events processed in each experiment can be seen in Table 2. The elastic model is close to the optimal values in Application 1 losing only around 1.1% of the events. This application receives a more predictable traffic. In the case of Application 3, where traffic had a larger amount of peaks of data, the loss of events reached 13.8%. Then, sudden peaks of events at high rate produce a higher number of event loss. This is explained by the window time of the monitor module, which is set to 100 s for the mid-term algorithm. This amount of time may be too long for inducing a quick reaction to the sudden peaks of data. We suggest decreasing the value of t_m when this may be the case.

Moreover, we have defined the default value for the amount of replicas created by the mid-term algorithm as $R' = 5$ and as $R = 1$

[6] http://www.cs.waikato.ac.nz/ml/weka/
[7] http://opennlp.apache.org/



(a) Throughput (b) Number of replicas

(c) CPU usage (d) Memory usage

Fig. 14. Application 3 results with execution time of 10 min.

for the short-term algorithm. Changing these values impacts how quickly the system can cope with sudden peaks of data. For example, using $R' = 8$ decreases the loss of events from 13.8% to 9.7% for Application 3. The use of this value in the Application 1, on the other hand, is not significant decreasing the event loss from 1.1% to 0.8%.

Other parameters that impact the results are the threshold values for increasing replicas or decreasing replicas in the short-term algorithm. The threshold to start replicating was set to 1.0 in the default configuration; however, we can increase the amount of processed events if we take the decision earlier (i.e. if we set the threshold to a lower value). If we set the *upper_threshold* value to 0.8 the loss of events decreases from 13.8% to 1.4% for Application 3. In Application 1, the selection of this value has no major impact decreasing from 1.1% to 0.6%.

We conclude that for traffic with high rate of events and sudden peaks of data the impact of the parameter selection is high. In this case the system requires to detect an overloaded state as soon as possible in order to increase the amount of replicas. Then, using a lower *upper_threshold* to react sooner and avoid loss of events is better.

The impact of the *lower_threshold* could not be measured in these experiments since we are executing the stream processing application over one machine with many cores. Then, we cannot observe the benefits of having a small number of replicas in the system. In future work, we expect to run this approach over the cloud, and measure the costs of having more PEs receiving events in different machines. Moreover, the impact of having a lower amount of replicas is related to the amount of memory each PE requires, and in our case the experiments were performed with lightweight PEs. PEs related with machine learning classifiers, for example, are costly to maintain in memory in an underloaded state. In this context, defining a large set of replicas for each operator would produces high memory costs.

Our elastic approach applies two algorithms: the short-term and the mid-term algorithm. The latter was activated in the experiments performed with Application 1 and Application 3. We have measured the impact of the mid-term algorithm by running our elastic approach only with the short-term algorithm. The result obtained for Application 1 in terms of the number of events loss increases from 1.1% to 2.5% (i.e. more than the double). In Application 3 without the mid-term algorithm, the number of events loss would have increased to 16.4%. Finally, we conclude that in any case (under peaks of traffic of like in Application 3 or in more stable scenarios, like in Application 1) it is important to predict traffic instead of only react to incoming events.

Our approach can be used in any SPE that use graph-based topologies, under any type of application with different datasets. However, parameters have to be carefully selected when dealing with high data rates and sudden peaks of traffic.

5. Related work

In the context of SPEs, efforts to provide elasticity of resources associated with computation are tightly linked with the graph topology of the deployed processing operators. Commercial systems such as Apache S4[8] or Storm,[9] do not provide elasticity of resources and the size of the cluster is specified at the beginning of the execution. In the case of Storm, the number of processing operators (bolts) is fixed and the system requires restarting for adaptation to event dynamics. S4, on the other hand, may also use a hash-based partition that creates processing operators at runtime for each new key-value in the data, which are mapped to the available resources. Both systems are unable to automatically adapt to the dynamics of the stream of events.

Recently, research efforts are aimed towards optimizing resource configuration and to dynamically scale SPEs in or out. We classify optimizations of current resources in three types, to improve *operator placement* in the machines minimizing communication (Xu et al., 2014), load balancing between physical nodes (Zhou et al., 2008; 2006; Wu and Liu, 2014) and maximizing system utilization (Heinze et al., 2013), *load balancing data* processing (Rivetti et al., 2015; Shah et al., 2003; Gedik, 2014), and *fault tolerance* (Madsen and Zhou, 2015; Fernandez et al., 2014; Zaharia et al., 2013). All these solutions are complementary to our proposal.

[8] incubator.apache.org/s4/
[9] http://samza.incubator.apache.org

Please cite this article as: N. Hidalgo et al., Self-adaptive processing graph with operator fission for elastic stream processing, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.010

The work of Hirzel et al. (2014) presents a catalog of optimizations for stream processing systems. Optimizations that leave the graph unchanged are load balancing, batching events and load shedding. Load shedding is used by S4 in case of data overloading; this type of technique prevents congestion; however, it decreases processing accuracy. Batching of events and load balancing increase performance without dealing with the problem of dynamic changes of input streams.

In order to achieve elasticity automatically, authors have proposed to attach/release machines so that processing operators have adequate resources to handle data. In Esc (Satzger et al., 2011) the authors proposed to dynamically attach and release machines to adjust computation capacities to the current nodes. Our work assumes unlimited resources, but it can be complemented with CPU/memory analysis in order to include resources elasticity.

Changes in the topology of the processing graph to have data parallelism have been surveyed in Hirzel et al. (2014). Operator reordering, separation, fusion and fission are the optimizations to be applied to improve performance of the stream processing system. Operator reordering acts on the selectivity of the operators, so that the traffic drops after applying one operator in the form of a pipeline. In order to apply this optimization the operators need to be commutative. The separation of an operator decouples one processing task into smaller computational steps so that they can be located into separated processing nodes. This is difficult to be done automatically. Operator fusion does the opposite: removes the pipeline between the operator if they are lightweight, reducing the overhead generated by the communication between them.

This work uses automatic data parallelism or replication, called fission by Hirzel et al. Fission of an operator is to parallelize computation over the same type of data, splitting the source stream before processing and merging the results afterwards. The merging step is only performed if the operator is stateful, meaning that it stores a state built with past processing steps. The conditions that must be met to guarantee the same results are described in Pollner et al. (2015). A similar model of fission operators is presented in Zeitler and Risch (2010); Wu and Liu (2014), and Schneider et al. (2012). StreamCloud (Gulisano et al., 2010) on the other hand, splits logical data into multiple physical data substreams that flow in parallel, preventing single-node bottlenecks.

In order to achieve resource elasticity in Storm, the authors in van der Veen et al. (2015) presented a model that monitors the Storm platform and external systems such as queues and databases. Their approach provides an initial guess of the size of the platform and then decides whether to add or remove virtual machines using a control loop. This work is focused on computing resources and does not change the processing graph topology.

Elasticity is integrated with fault tolerance in Enorm (Madsen et al., 2014) that uses a checkpointing mechanism to allow low cost and fast state migration when increasing or decreasing parallelism. This work focuses on how to adapt the system when scaling in or out, but not on the amount of parallelism required in the system. We believe their proposal can be integrated in our solution to provide integrity when replicating stateful operators. Moreover, their work uses a threshold-based approach to determine when to scale.

Another work that uses threshold-based decisions is Heinze et al. (2014b), which builds over FUGU (Heinze et al., 2013). The authors compared decisions based on local thresholds on the processing operators, global thresholds of the system and a reinforcement learning approach. The thresholds used are a lower and an upper bound plus a target utilization value. They considered a grace period after a scaling decision in order to maintain stability. The reinforcement learning approach uses the actions: scaling up, scaling down and no action. The systems uses as reward a weighted average of the difference between the current value and respective target system utilization. An extension of this approach is presented in Heinze et al. (2014a), where the authors try to minimize the number of latency violation maximizing the utilization values. In this case, there are decisions that are labeled as optional, and that can be cancelled or postponed in case the estimated latency spike is too high. Later works of the authors include fault tolerance analysis (Heinze et al., 2015b) and a prototype that allows to analyze the tradeoff between the monetary costs against the offered quality of service (Heinze et al., 2015a).

StreamCloud uses a similar elasticity protocol that sets conditions that trigger provisioning, decommissioning or load balancing. They all depend on the CPU utilization of the machines, not on the data parallelism related to the graph topology.

Congestion and throughput is measured in Gedik et al. (2014) to provide elasticity. Congestion is observed when there is a delay when sending tuples on a connection. Throughput is the number of tuples processed per second over the last adaptation period. This work only considers a reactive approach in front of changes in the workload.

Several attempts have been made to automatically scale in/out applications running on top of cloud platforms or large clusters (Lorido-Botran et al., 2014). Strategies to achieve elasticity in a system are classified in threshold-based, reinforcement learning (both have been applied to SPEs), queuing theory, control theory and time series analysis, which we apply in this work using Markov Chains.

In the field of autonomic computing it is common to use the Monitor, Analyze, Plan and Execute model, called MAPE. We follow this model to adapt a SPE to the changes in the flow of events, as the authors in van der Veen et al. (2015) and Satzger et al. (2011) applied to resource adaptation.

Broad categories for elasticity are reactive and proactive (Nikravesh et al., 2014). The reactive elasticity is based on the current state of the system and past work on SPE used this approach. The proactive approach tries to detect seasonal data. In this work we applied this type of model to a SPE.

6. Conclusion & future work

This work proposes a self-adaptive processing system that optimizes resources usage adapting the system to the dynamism of the flow of events. Based on the MAPE model, our solution is able to predict operator's load using statistical information collected from them. We apply two simple algorithms to determine the state of the operator: short-term and mid-term algorithms. Short-term algorithm detects peaks of traffics while mid-term algorithm predicts the traffic using past behavior of the flow of events.

We have evaluated our system in the context of three application and two different datasets. Results show that the elastic approach can increase the number of replicas of the operators adapting the graph topology to the data rate and sudden peaks of data. The loss of events depends on the selection of parameters of the algorithms, for example, the amount of replicas created when the system predicts an overloaded state for an operator and the threshold for considering an operator as overloaded. Moreover, we have also measured the costs of our implementation in terms of CPU and memory showing that they are low in comparison to the processing computational resources.

In the future, we plan to evaluate our solution using the benchmark datasets of LRB (Arasu et al., 2004) and to compare our solution using stateful operators in the topology to evaluate latency changes in the graph. We expect to implement the solution presented in Madsen et al. (2014) to migrate state of operators in a low cost and fast manner. Moreover, we will implement a holistic approach where replication is performed at several stages in the topology, integrating the cost of deploying processing opera-

Please cite this article as: N. Hidalgo et al., Self-adaptive processing graph with operator fission for elastic stream processing, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.010

tors over computing resources versus the accuracy achieved in the optimization process.

Acknowledgment

This work is partially supported by the University of Santiago research project PMI-USA 1204, CeBiB basal funds FB0001, and FONDEF IDEA ID15I10560, CONICYT, Chile. Nicolas Hidalgo would also like to thank DICYT-USACH 061419HC.

References

Appel, S., Frischbier, S., Freudenreich, T., Buchmann, A.P., 2012. Eventlets: Components for the integration of event streams with SOA. In: Proceedings of the 2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA), Taipei, Taiwan. IEEE, pp. 1–9.

Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R., 2004. I road: A stream data management benchmark. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30. VLDB Endowment, pp. 480–491.

Ching, W.-K., Ng, M.K., 2006. Markov Chains: Models, Algorithms and Applications. Springer US, New York.

Fernandez, R.C., Weidlich, M., Pietzuch, P., Gal, A., 2014. Scalable stateful stream processing for smart grids. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS). ACM, pp. 276–281.

Gedik, B., 2014. Partitioning functions for stateful data parallelism in stream processing. Very Large Data Bases (VLDB) J. 23 (4), 517–539.

Gedik, B., Schneider, S., Hirzel, M., Wu, K.-L., 2014. Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. 25 (6), 1447–1463.

Gong, Z., Gu, X., Wilkes, J., 2010. Press: Predictive elastic resource scaling for cloud systems. In: Proceedings of the 2010 International Conference on Network and Service Management (CNSM), pp. 9–16.

Gulisano, V., Jiménez-Peris, R., Patiño-Martínez, M., Valduriez, P., 2010. Streamcloud: A large scale data streaming system. In: Proceedings of the 2010 International Conference on Distributed Computing Systems (ICDCS), Genova, Italy. IEEE Computer Society, pp. 126–137.

Heinze, T., Jerzak, Z., Hackenbroich, G., Fetzer, C., 2014a. Latency-aware elastic scaling for distributed data stream processing systems. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS). ACM, pp. 13–22.

Heinze, T., Ji, Y., Pan, Y., Grueneberger, F.J., Jerzak, Z., Fetzer, C., 2013. Elastic complex event processing under varying query load. In: Proceedings of the First International Workshop on Big Dynamic Distributed Data (BD3). CEUR-WS, pp. 25–30.

Heinze, T., Pappalardo, V., Jerzak, Z., Fetzer, C., 2014b. Auto-scaling techniques for elastic data stream processing. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS). ACM, pp. 318–321.

Heinze, T., Roediger, L., Meister, A., Ji, Y., Jerzak, Z., Fetzer, C., 2015a. Online parameter optimization for elastic data stream processing. In: Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC). ACM, pp. 276–287.

Heinze, T., Zia, M., Krahn, R., Jerzak, Z., Fetzer, C., 2015b. An adaptive replication scheme for elastic data stream processing systems. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS). ACM, pp. 150–161.

Hidalgo, J.M.G., Almeida, T.A., Yamakami, A., 2012. On the validity of a new SMS spam collection. In: Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA), FL, USA. IEEE, pp. 240–245.

Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R., 2014. A catalog of stream processing optimizations. ACM Comput. Surv. 46 (4), 46:1–46:34.

Hunt, P., Konar, M., Junqueira, F.P., Reed, B., 2010. Zookeeper: Wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference. USENIX Association. 11–11.

Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36 (1), 41–50.

Leskovec, J., Backstrom, L., Kleinberg, J.M., 2009. Meme-tracking and the dynamics of the news cycle. In: Proceedings of the15th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), Paris, France, pp. 497–506.

Lin, J., Dyer, C., 2010. Data-Intensive Text Processing with MapReduce. Morgan and Claypool Publishers.

Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A., 2014. A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. 12 (4), 559–592.

Madsen, K.G.S., Thyssen, P., Zhou, Y., 2014. Integrating fault-tolerance and elasticity in a distributed data stream processing system. In: Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDBM). ACM, pp. 48:1–48:4.

Madsen, K.G.S., Zhou, Y., 2015. Dynamic resource management in a massively parallel stream processing engine. In: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM). ACM, pp. 13–22.

Mone, G., 2013. Beyond hadoop. Commun. ACM 56 (1), 22–24.

Neumeyer, L., Robbins, B., Nair, A., Kesari, A., 2010. S4: distributed stream computing platform. In: 2010 IEEE International Conference on Data Mining Workshops (ICDMW), Sydney, Australia. IEEE, pp. 170–177.

Nikravesh, A., Ajila, S., Lung, C.-H., 2014. Cloud resource auto-scaling system based on hidden markov model (hmm). In: Proceedings of the 2014 IEEE International Conference on Semantic Computing (ICSC). IEEE, pp. 124–127.

Oberhelman, D.D., 2007. Coming to terms with web 2.0. Ref. Rev. 21 (7), 5–6.

Papoulis, A., 1984. Probability, Random Variables, and Stochastic Processes. Mc-Graw Hill, New York.

Pollner, N., Steudtner, C., Meyer-Wegener, K., 2015. Operator fission for load balancing in distributed heterogeneous data stream processing systems. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS). ACM, pp. 332–335.

Rivetti, N., Querzoni, L., Anceaume, E., Busnel, Y., Sericola, B., 2015. Efficient key grouping for near-optimal load balancing in stream processing systems. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS). ACM, pp. 80–91.

Satzger, B., Hummer, W., Leitner, P., Dustdar, S., 2011. Esc: Towards an elastic stream computing platform for the cloud. In: Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD). IEEE, pp. 348–355.

Schneider, S., Hirzel, M., Gedik, B., Wu, K.-L., 2012. Auto-parallelizing stateful distributed streaming applications. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT). ACM, pp. 53–64.

Shah, M., Hellerstein, J., Chandrasekaran, S., Franklin, M., 2003. Flux: an adaptive partitioning operator for continuous query systems. In: Proceedings of the IEEE 19th International Conference on Data Engineering (ICDE). IEEE, pp. 25–36.

Tagg, C., 2009. A corpus linguistics study of SMS text messaging. The University of Birmingham Ph.D. thesis.

Thelwall, M., Buckley, K., Paltoglou, G., Cai, D., Kappas, A., 2010. Sentiment in short strength detection informal text. J. Am. Soc. Inf. Sci. Technol. 61 (12), 2544–2558.

van der Veen, J., Van Der Waaij, B., Lazovik, E., Wijbrandi, W., Meijer, R., 2015. Dynamically scaling apache storm for the analysis of streaming data. In: Proceedings of the 2015 IEEE First International Conference on Big Data Computing Service and Applications (BigDataService). IEEE, pp. 154–161.

Wu, X., Liu, Y., 2014. Optimization of load adaptive distributed stream processing services. In: Proceedings of the 2014 IEEE International Conference on Services Computing (SCC). IEEE Computer Society Press, pp. 504–511.

Xu, J., Chen, Z., Tang, J., Su, S., 2014. T-storm: Traffic-aware online scheduling in storm. In: Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp. 535–544.

Yu, G., Hu, J., Zhang, C., Zhuang, L., Song, J., 2003. Short-term traffic flow forecasting based on markov chain model. In: Proceedings of the IEEE Intelligent Vehicles Symposium. IEEE, pp. 208–212.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I., 2013. Discretized streams: Fault-tolerant streaming computation at scale. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP). ACM, pp. 423–438.

Zeitler, E., Risch, T., 2010. Scalable splitting of massive data streams. In: Database Systems for Advanced Applications. In: Lecture Notes in Computer Science, Vol. 5982. Springer Berlin Heidelberg, pp. 184–198.

Zhou, Y., Aberer, K., Tan, K.-L., 2008. Toward massive query optimization in large-scale distributed stream systems. In: Issarny, V., Schantz, R. (Eds.), Middleware 2008. In: Lecture Notes in Computer Science, Vol. 5346. Springer, pp. 326–345.

Zhou, Y., Ooi, B., Tan, K.-L., Wu, J., 2006. Efficient dynamic operator placement in a locally distributed continuous query system. In: On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE. In: Lecture Notes in Computer Science, Vol. 4275. Springer, pp. 54–71.

Nicolás Hidalgo obtained his Ph.D. degree in computer science at the Pierre and Marie Curie University of Paris in 2011, Paris. Currently, Mr. Hidalgo is an assistant professor at the University of Santiago, Santiago. Mr. Hidalgo also worked as a full time researcher at Yahoo! Labs, Santiago. His research areas cover highly scalable distributed systems, stream processing and fog computing.

Daniel Wladdimiro obtained his Master degree in computer science at the University of Santiago de Chile in 2015. His master thesis was focused on implementing an elastic data stream processing system based on operators fission. Nowadays, Mr. Wladdimiro works as research engineer for the Citiaps center at the University of Santiago.

Erika Rosas is an assistant professor in the Department of Informatics Engineering at the University of Santiago, Santiago. Ms. Rosas obtained her Ph.D. degree in computer science in 2011 from Pierre and Marie Curie University in Paris (Paris VI). She is a former postdoctoral researcher of Yahoo! Labs Santiago. Her research areas cover stream processing, mobile communications, and large scale networks such as P2P and social networks.

Please cite this article as: N. Hidalgo et al., Self-adaptive processing graph with operator fission for elastic stream processing, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.010