Elastic Scaling for Distributed Latency-sensitive Data Stream Operators

Tiziano De Matteis and Gabriele Mencagli
Department of Computer Science, University of Pisa
Largo B. Pontecorvo 3, I-56127, Pisa, Italy
Email: {dematteis, mencagli}@di.unipi.it

***Abstract*—High-volume data streams are straining the limits of stream processing frameworks which need advanced *parallel processing* capabilities to withstand the actual incoming bandwidth. Parallel processing must be synergically integrated with *elastic* features in order dynamically scale the amount of utilized resources by accomplishing the *Quality of Service* goals in a cost-effective manner. This paper proposes a control-theoretic strategy to drive the elastic behavior of latency-sensitive streaming operators in distributed environments. The strategy takes scaling decisions in advance by relying on a predictive model-based approach. Our ideas have been experimentally evaluated on a cluster using a real-world streaming application fed by synthetic and real datasets. The results show that our approach takes the strictly necessary reconfigurations while providing reduced resource consumption. Furthermore, it allows the operator to meet desired average latency requirements with a significant reduction in the experienced latency jitter.**

***Index Terms*—Data Stream Processing, Elastic Scaling, Continuous Queries**

I. Introduction

In a world driven by information, *Data Stream Processing* (*DaSP*) represents one of the hottest IT trending topics [1].

DaSP applications are typically executed 24-hours-a-day, seven-days-a-week to process continuous data flows coming from different sources. In this scenario, there exist natural "ebbs and flows" in the input rate and workload characteristics that must be promptly addressed by the run-time system supporting stream processing applications.

Stream processing applications must be able to autonomically scale up or down the used resources by maintaining the required *Quality of Service* (QoS) in a cost-effective manner with reduced system downtimes [2], [3]. Decisions on *when* and *how* to change the current application configuration (e.g., number of used resources) must be taken by *elastic strategies* with the following expected characteristics:

- they must be able to meet the desired QoS requirements. In addition, we focus on quality-sensitive real-time applications whose users are strongly affected by the latency of delivered results (and by its variability);
- to take a new scaling decision, the strategy should account for the *stability* of the application configuration, avoiding frequent modifications that may result in a high number of QoS violations and highly irregular QoS outcomes;
- they should be *cost-effective* by avoiding wasting resources.

In distributed environments the impact of these properties is further emphasized, due to the significant transient overhead of re-scaling decisions (state migration among nodes [2]). The contributions of this work are summarized as follows:

- we present a control-theoretic strategy based on *Model Predictive Control* (MPC) [4]. With respect to our previous works [5], [6] which were limited to multicores, the strategy and its implementation framework have been extended to distributed environments which are by far the most realistic case for stream processing frameworks;
- we evaluate our strategy in a real-world context of a latency-sensitive streaming application, by deriving the impact of the strategy in the amount of reconfigurations performed, average resource consumption and number of QoS violations experienced. A first comparison among our strategy and an existing heuristic is proposed;
- we evaluate the impact of reconfigurations in the delivered latency, by analyzing the effect of our strategy in minimizing the latency variability in terms of jitter.

The paper is organized as follows. Sect. II provides an overview on Data Stream Processing. Sect. III presents our control-theoretic approach. Sect. IV describes our distributed implementation and Sect. V shows the experimental evaluation. Finally, Sect. VII states the conclusion of this work.

II. Background

DaSP applications are modeled as direct graphs whose vertices are *operators* and arcs are *streams* [1]. The input stream conveys a sequence of data items (*tuples*), consumed and analyzed by the application. They represent occurred *events* or, more in general, any information of interest. An operator applies an intermediate computation that consumes input items and produces new tuples as new stream(s).

Stateful operators are a class of operators that keep an internal state while processing the input items. Output results depend on the value of the internal state which is continuously updated. Of special attention are the so-called *partitioned-stateful* (or *keyed*) operators [1], [2], [7], where the input stream conveys tuples belonging to multiple *logical substreams* that must be processed independently. In those cases, the data structures supporting the internal state can be separated into independent partitions, one per substream. An example is a pattern recognition operator that processes

independently tuples belonging to different stock symbols in a financial market stream.

A. Elastic Intra-operator Parallelism

In case the specified QoS requirements are not met by the current implementation (e.g., actual latency is unsatisfactory), bottleneck operators must be internally parallelized. Parallelism deserves special attention for stateful operators, in order to preserve the computation semantics and correctness.

Typically a partioned-stateful operator is parallelized by replicating the operator in multiple *worker* entities, each one processing a subset of the logical substreams [1], [7]. The logical scheme of this parallelization is sketched in Fig. 1.

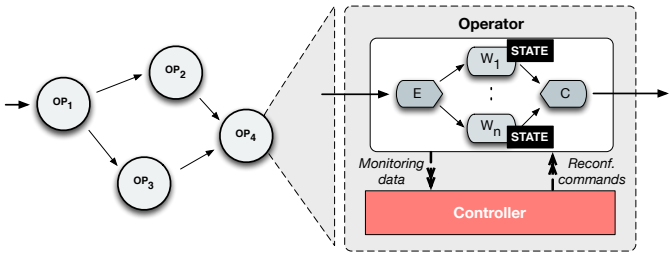


Fig. 1: Graph of streaming operators: scheme of parallel and elastic partitioned-stateful operator.

The operator receives a stream of tuples and produces a stream of results. All the tuples with the same partitioning attribute (*key*) are processed sequentially by the same worker (W_i) in the arrival order. The *emitter* (E) and *collector* (C) entities act as interfaces from/to input and output stream(s). The first is in charge of routing each input tuple to the corresponding worker using a (hash) *routing function*. The collector receives results from the workers and transmits them onto the output stream(s).

DaSP scenarios are often characterized by intensive workload fluctuations (e.g., variability in the arrival rate, processing time per tuple and changes in the frequency of keys) that must addressed in the design of parallel operators. Therefore, the parallelized operator should adapt itself in order to keep its QoS optimized according to the user expectations.

In an *elastic* scenario the parallel operator is supported by a *controller* (red part in Fig. 1), an entity able to observe the operator execution and to respond to different cases of dynamicity. The two parts interact according to a closed loop (*Monitoring, Analyze, Plan, Execute*). Based on the evaluation of a specific strategy, a set of actions are taken by the controller to change the operator configuration (e.g., distribution function, amount of parallelism). In this paper the main focus is on the design of such strategies, which assume a foremost role in achieving good quality by the overall operator execution.

III. Latency-aware Predictive Elasticity

Three fundamental goals must be accomplished by any elastic support for stream processing operators. They must provide *accuracy* in maintaining the desired QoS level required by the final user (e.g., providing a desired average latency value), *stability* of the reconfiguration process, i.e. avoid frequent reconfigurations that could have a detrimental effect on the operator performance, and being *cost-effective* using the strictly necessary computing resources to respect the given QoS constraints.

In Refs. [5], [6] we investigated the feasibility of Model Predictive Control (MPC) [4] as a viable control-theoretic approach to drive the elastic behavior of stream processing operators on multicores. Here, we extend this model for latency-sensitive operators executed on distributed architectures.

A MPC-based controller exploits a model to predict the future system behavior over a *limited prediction horizon*. The controller is *time-driven*: the adaptation strategy is re-evaluated at the beginning of fixed time intervals called *control steps*. At each step the controller executes three distinct phases described in the following three subsections.

A. Disturbances Predictions

The controller obtains the past measurements of disturbance inputs that cannot be directly controlled. They include:

- T_A, σ_A: the mean and standard deviation of the inter-arrival time of tuples over the input stream;
- $\{p_k\}_{k\in\mathcal{K}}$ current frequency distribution of the keys (logical substreams);
- $\{T_k\}_{k\in\mathcal{K}}$ mean computation time per tuple for the different keys.

To profitably apply the MPC-based technique, the controller must be able to predict the future evolution of such disturbances over a limited future horizon of some control steps. Owing to the past history updated periodically, the controller is in charge of evaluating time-series history-based filters (e.g., autoregressive moving average or neural network approaches to non-linear forecasting [8]). In the following, we indicate with a *tilde* superscript the predicted values of disturbances.

B. System Model

The elastic support uses a system model to dynamically compare the QoS that would be achieved under the predicted disturbance conditions by different operator configurations. The model captures the relation between *QoS variables* (e.g., the operator latency) and a configuration expressed as the current number of workers utilized (denoted by n).

To evaluate the expected latency achieved by an operator configuration, our MPC-based strategy exploits analytical models based on fundamental results of Queuing Theory.

We define the *ideal service rate* of the operator as the average number of tuples that the operator is able to serve per time unit assuming that it is never idle. We are interested in the inverse quantity, i.e. the *ideal service time* T_S^{id}. Under the assumption that the load is evenly distributed among the workers (the emitter functionality will update the routing function to provide this property, see Sect. IV), the ideal service time at step τ can be computed as:

$$\widetilde{T}_S^{id}(\tau) = \frac{\sum_{k\in\mathcal{K}} \widetilde{p}_k(\tau)\widetilde{T}_k(\tau)}{n(\tau)} = \frac{\widetilde{T}(\tau)}{n(\tau)} \qquad (1)$$

i.e. the ratio of the mean computation time T per tuple of any key over the current number of workers n utilized during the same control step.

Our target measurement is the latency, more formally called *response time* of the operator. It is the average time interval from when a newly received tuple arrives at the operator until the corresponding result is transmitted onto the operator output stream. It can be modeled as the sum of two quantities:

$$\mathcal{R}_Q(\tau) = W_Q(\tau) + T(\tau) \tag{2}$$

where W_Q is the *mean waiting time* that a tuple spends from its arrival to the system to when the operator starts its execution in the corresponding worker.

To estimate the current average waiting time, our idea is to model the operator as a $G/G/1$ queueing system, i.e. a server with inter-arrival time and service times having general statistical distributions. An approximation is given by Kingman's formula [9]:

$$\widetilde{W}_Q(\tau) \approx \left(\frac{\widetilde{\rho}(\tau)}{1-\widetilde{\rho}(\tau)}\right)\left(\frac{\tilde{c}_a^2(\tau)+\tilde{c}_s^2(\tau)}{2}\right)\widetilde{T}_S^{id}(\tau) \tag{3}$$

where the utilization factor of the operator during step τ is defined as $\widetilde{\rho}(\tau) = \widetilde{T}_S^{id}(\tau)/\widetilde{T}_A(\tau)$, and c_a and c_s represent the coefficient of variations of the inter-arrival and ideal service time while the ideal service time can be determined as in Eq. 1.

This model has been adopted in the past for the latency estimation of streaming operators [3] and provides good accuracy provided that the needed measurements (i.e. coefficient of variations and utilization factor of the server) can be monitored and estimated from the running computation. Furthermore, the model does not need unrealistic assumptions like fixed stochastic distributions for the arrival and service processes (like the exponential distributions adopted for simplified $M/M/1$ and $M/M/n$ models). In fact, any arrival and service distributions can be modeled provided that the needed parameters can be estimated at run-time.

We use this model to drive the MPC-based latency-aware elastic support of our distributed operator. To increase the precision of the model in highly-variable execution scenarios, we use a feedback mechanism to correct the estimation obtained by Kingman's formula according to the average prediction errors measured in the last control steps.

C. Optimization

The third step of the MPC-based strategy is to solve an optimization problem exploiting the system model. The controller does not look only for the best configuration for the next step. Instead, it is capable of analyzing the estimated behavior for a longer horizon in order to drive more stable and accurate decisions. Therefore, the result of the optimization problem is a *reconfiguration trajectory* $U^h(\tau) = (\mathbf{n}(\tau), \mathbf{n}(\tau+1), \ldots, \mathbf{n}(\tau+h-1))$ over a prediction horizon of $h \geq 1$ steps. Formally, the optimization problem is as follows:

$$\min_{U^h(\tau)} \ \mathcal{J} = \sum_{i=0}^{h-1} L\big(\widetilde{\mathcal{R}}_{\mathbf{Q}}(\tau+i), n(\tau+i)\big) \tag{4}$$

Once the optimal trajectory is found, the controller implements the first element of that trajectory as the new configuration of the operator for the next step. Then, the whole procedure is re-iterated at the next control step, using the new disturbance measurements to update the forecasts (this principle is universally known as *receding horizon* [4]).

The function L is a *step-wise cost* that can be defined to model different control objectives. The formulation that we use is the following (with $i = 0, \ldots, h-1$):

$$\begin{aligned} L\big(\widetilde{\mathcal{R}}_{\mathbf{Q}}(\tau+i), n(\tau+i)\big) = Q_{cost}(\tau+i) + & \quad \textit{QoS cost} \\ + R_{cost}(\tau+i) + & \quad \textit{Resource cost} \\ + S^w_{cost}(\tau+i) & \quad \textit{Switching cost} \end{aligned} \tag{5}$$

The *QoS cost* represents the user degree of satisfaction with the actual QoS of the operator. Since we are interested in latency-sensitive operators, we suppose that the QoS requirement is a maximum bound on the actual average response time. To model this, we use the following cost definition:

$$Q_{cost}(\tau+i) = \alpha \cdot \exp\left(\frac{\widetilde{R}_{\mathcal{Q}}(\tau+i)}{\delta}\right) \tag{6}$$

where $\alpha > 0$ is a normalization factor. The cost lies in the interval $(\alpha, e\alpha]$ for latency values within the interval $(0, \delta]$, where $\delta > 0$ is a *maximum threshold* for the response time. The idea is that such kind of cost heavily penalizes configurations with a response time greater than the threshold.

The *resource cost* models a penalty proportional to the amount of resources consumed, that is a cost proportional to the number of used workers, i.e. $R_{cost}(\tau+i) = \beta \cdot n(\tau+i)$ where where $\beta > 0$ is a normalization factor.

Finally, the *switching cost* models the penalty incurred in changing the current configuration. Although different definitions can be adopted, in this paper we analyze the effect of a switching cost capable of penalizing large changes in the used configuration. It is formally defined as follows:

$$S^w_{cost}(\tau+i) = \gamma\left(n(\tau+i) - n(\tau+i-1)\right)^2 \tag{7}$$

where $\gamma > 0$ is a normalization factor.

IV. Distributed Implementation

In this section we describe our distributed implementation of elastic partitioned-stateful data stream operators. The implementation is written in `C++` and targets distributed systems of multicore-based nodes. The design is sketched in Fig. 2. The implementation distinguishes two types of *nodes*:

- a ***master node*** executing the (global) *emitter* (*E*), (global) *collector* (*C*), and the *controller* functionalities equipped with other services (e.g., for the implementation of reconfiguration activities) as described in the sequel;
- a set of ***executor nodes***, i.e. nodes in charge of executing the workers of the distributed operator.

This distinction provides a clear separation between the nodes in charge of executing the business logic of the operator (i.e. where the worker instances are running) from the one that runs

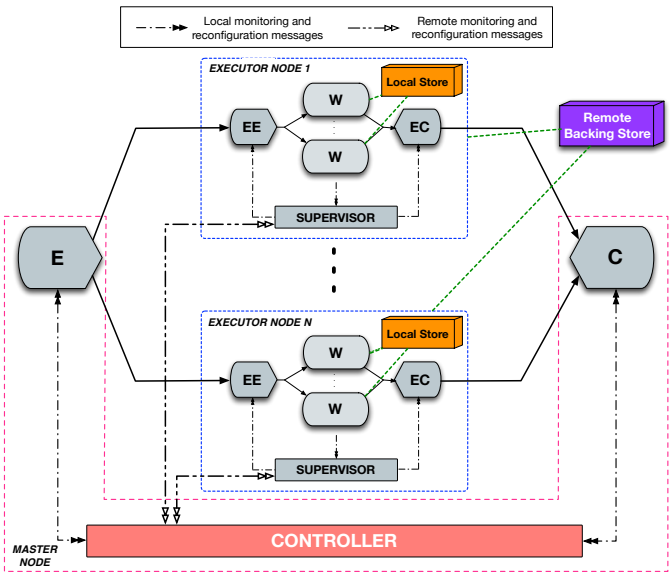


Fig. 2: Schema of the distributed operator.

support tasks to interface the whole distributed implementation with the input and the output streams.

As Fig. 2 shows, each executor node runs a sort of replica of the parallelized operator. Besides the workers (each one executed by a dedicated thread), an execution node has additional entities executed by a corresponding number of threads:

- an executor's emitter (*EE* in figure) and an executor's collector (*EC*). They are connected with the global emitter and collector and are respectively in charge of dispatching input tuples to the destination workers and gathering the produced results by sending them to the global collector;
- a *supervisor* entity in charge of collecting all the measurements from the workers execution, and of transmitting them as a single *monitoring snapshot* to the controller at the beginning of each control step. Furthermore, the supervisor implements run-time support mechanisms for spawning/killing worker instances.

The run-time system tries to package the worker instances in the minimum number of executor nodes: only when there is no space for adding additional workers within an existing executor (i.e. we are using all the available cores of the node), we start deploying a new executor onto an available node.

Incoming tuples are routed to the correct worker instances associated with the corresponding substream. Such routing is performed *hierarchically*, i.e. the global emitter maintains an *association table* between partitioning keys and existing executor nodes while each *EE* maintains the correspondence between keys and worker instances.

The global controller is in charge of evaluating the MPC-based strategy described in Sect. III. Although the strategy evaluation is centralized, all the run-time support actions and reconfiguration activities are decentralized and executed independently by the supervisors in each executor node. This is a compromise to allow the whole implementation to be easily expandable in systems with many available execution nodes.

Excluding the operational data flow, we can distinguish two different classes of messages: *local messages* exchanged within an executor node, and *remote messages* between executor nodes and master node. For remote messages we have:

- *monitoring messages*: the monitoring snapshot of an executor node is periodically transmitted by the supervisor to the remote controller;
- *reconfiguration messages* sent by the controller to the supervisors in order to implement an operator reconfiguration. Supervisors react to such messages by instantiating or removing workers within the executor node.

In the implementation, remote messages are transmitted over Posix TCP/IP sockets between the master and the execution nodes. All the data/messages remotely exchanged are serialized using the Google Protocol Buffer (`protobuf`) library [10]. Internally to an executor node, the entities (`EE`, workers and `CC`) interact by exchanging memory pointers to shared data structures using the cooperation model (lock-free single-producer single-consumer queues) provided by the `FastFlow` parallel programming frameworks [11].

A. Support to Elasticity

Reconfiguration mechanisms apply the strategic choices taken by the controller by increasing/decreasing the number of worker instances currently present in the executor nodes. As already pointed out in Ref. [2], *state migration* is the fundamental task performed by the run-time support. Once that the number of workers is changed, the input bandwidth must be redirected to them in order to keep the computational load balanced. Therefore, a subset of the existing keys, and the data structures supporting their internal state, must be properly migrated between distinct workers.

Several state migration protocols have been presented in the literature [2], [12], [13] with the goal of reducing the downtime experienced by the operator during the reconfiguration process and by preserving the computation semantics, i.e. by processing all the tuples in their arrival order. Tuples of the moving keys received during the reconfiguration phase are temporarily buffered and routed to the worker when the migration of the needed data structures is complete. In our implementation we adopt a protocol inspired by the work in Ref. [12], where, in order to avoid blocking the natural flow of the tuples non involved by the migration, new input items belonging to migrated keys are temporarily buffered in the destination workers and processed at the end of the migration.

State migration requires the transferring of some data structures. To this end, our run-time support implements a hierarchical repository with two levels, see Fig. 2:

- a ***local repository*** inside each executor node, used for state migration among the internal workers. It consists in a shared memory area where the migrated data structures (or just their pointers) are copied and acquired;
- a ***remote repository*** used for exchanging data between distributed executor nodes. It can be implemented in various ways: by back-end databases or using socket-based or MPI-based implementations [2].

Differently from some existing solutions [2], [12], [13], we decided to implement the remote repository using the *Memcached* service [14], a widely used repository for caching results of database calls (used by Facebook, Wikipedia, Flickr). Memcached is an *in-memory* store for small chunks of arbitrary data. Objects can be stored and retrieved by using an unique identifier. To utilize Memcached as remote repository, we need a Memcached server (`memcached`), i.e. an active entity maintaining the repository, and a client interacting with the server for saving/retrieving/deleting objects. In our case, the `memcached` server is in execution on the master node, and it is in charge of handling the requests arriving from the various clients, i.e. workers that copy/acquire the internal state serialized using `protobuf`) from/to the server.

V. Evaluation

The evaluation of the proposed control strategies is performed on a data stream processing application operating in the High Frequency Trading (HFT) domain, where applications are very latency-sensitive; both the magnitude and variability of latency play a decisive role for the end users. The computational kernel is sketched in Fig. 3.

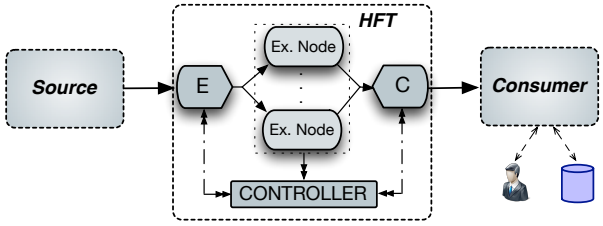


Fig. 3: Kernel of a high-frequency trading application used to discover trading opportunities in real-time.

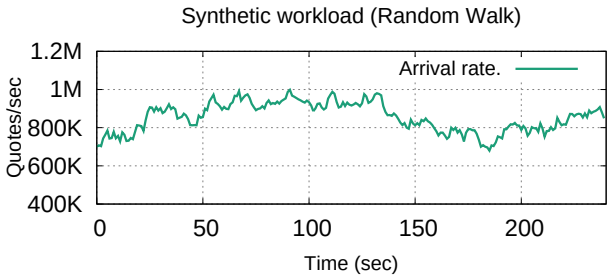
The *source* operator generates a stream of financial quotes (i.e. bid and ask proposals) represented by a record of the proposed price, volume and the stock symbol (64 bytes in total). The operator processes bids and asks grouped by the stock symbol (partitioning keys) and estimates the future volume and price for the quotes of each symbol using a sliding window of the last received tuples of the group. A count-based sliding window of size $|\mathcal{W}|$ and slide δ is maintained for each group. At each activation of a new window of data, the operator executes the *Levenberg-Marquardt* regression algorithm (C++ library `lmfit` [15]) to produce a polynomial fitting the quotes. To reduce the number of quotes to process, they are first aggregated using a resolution interval of 1 ms.

A. Distributed Architecture and Workload

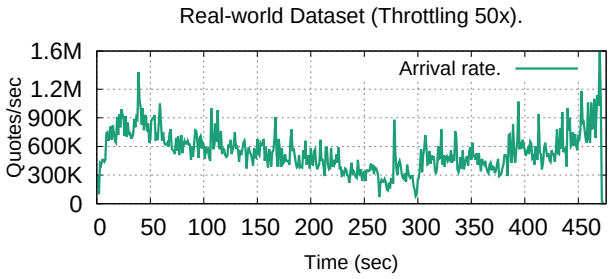
The distributed-memory system used for the evaluation is a small cluster of four identical nodes interconnected through an Infiniband network nominally working at 40Gb/s. Each node is a dual CPUs Intel Xeon E5-2699, for a total of 36 physical cores running at 2.30GHz (HyperThreading was turned off). Every machine has 192GB of RAM and runs a Linux-based operating system. The used compiler is `gcc` (version `4.8.1`), and we compile with the `-O3` optimization flag. The additional services/libraries used are: Fastflow (version `2.0.5`), Memcached (version `1.4.25`), Libmemcached (version `1.0.18`) and Google Protocol Buffer (version `proto2`).

All the entities in Fig. 2 are implemented as threads pinned on distinct cores of the nodes. For simplicity, we map the master onto a dedicated physical node while the other three nodes are used for the executors. Due to the presence of the *EE*, *EC* and supervisor entities, each executor node can run up to 33 worker instances for a maximum overall parallelism degree of the operator of 99 workers. The memcached server is executed with 8 threads in the master node, a configuration sufficient to handle all the requests in our execution scenarios. The default setting is to use windows of $|\mathcal{W}| = 2000$ tuples which are evaluate every new $\delta = 10$ tuples received.

We propose an evaluation under a synthetic workload shown in Fig. 4a, where the arrival rate follows a random-walk model while the key frequency distribution is fixed. The whole duration of the workload is of 240 seconds. Furthermore, we analyze the operator behavior in a real-world workload (Fig. 4b) obtained from a trading day of various stock markets (e.g., NASDAQ, NYSE)[1]. In this dataset we have $8,163$ traded stocks which a time-varying key probability distribution; on average the most frequent key constitutes the 0.81% of the whole generated quotes. The dataset has been accelerated 50 times to reproduce throttled input rates.



(a) Random-walk dataset.



(b) Real-world dataset.

Fig. 4: Arrival rate: synthetic and a real throttled ($50\times$) datasets.

B. Mechanisms Evaluation

The controller computes a new routing function to balance the workload among workers. This is executed periodically and at each operator reconfiguration. The necessary state migration actions generate some impact in the experienced

[1]Data available at ftp://ftp.nyxdata.com/Historical%20Data%20Samples/.

latency of the operator. To provide an evaluation of this effect, we compare two different heuristics. The first one, called *Balanced*, computes *ex-novo* a new routing table in order to balance the workload perfectly among the existing workers, possibly resulting in a large number of migrated keys. The second one is inspired by the work in Ref. [12] (*Flux*), where the operator tries to equalize the load among workers as much as possible while minimizing key movement. The idea is to use an *imbalance threshold* that represents an upper limit to the relative difference in the load between the most loaded worker and the least loaded one. In the comparison we use two versions of this heuristic, i.e. *Flux-5* and *Flux-10* using a threshold of 5% and 10% respectively.

In the test scenario the operator is not a bottleneck and the workload is balanced until timestamp 30. Then we force the input rate to suddenly change from $400K$ tuples/sec to $700K$ tuples/sec. At timestamp 31 the number of workers is changed from 30 to 60 in order to cope with the new input rate (using an additional executor node). In this case, the migration involves both the local and the remote repositories. The latency results of this analysis are shown in Fig. 5.

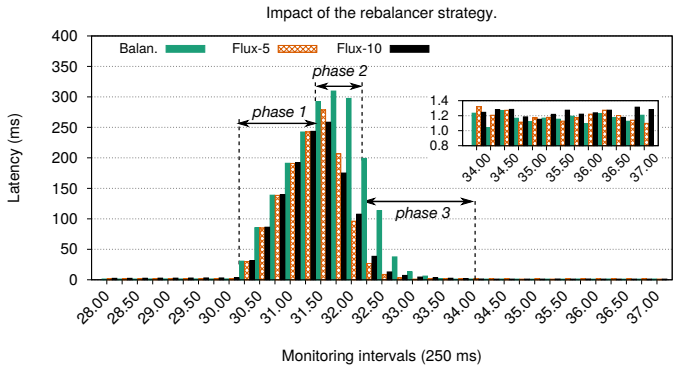


Fig. 5: Measured latency of the operator in the time instants before, during and after a reconfiguration.

In the first phase the rate changes and latency grows as the operator becomes a bottleneck. During this phase we do not measure any significant difference between heuristics. After timestamp 31 (*phase 2*) the controller triggers the reconfiguration, new workers are created and some keys are migrated. In the Balanced strategy we move $8,061$ keys ($2,690$ of which are moved using the remote repository), while with Flux-5 and Flux-10 we move 431 (389 remotely) and 418 (376 remotely) keys. The less the amount of migrated keys the lower the experienced latency peaks during this phase.

Finally, during *phase 3* workers process tuples that were buffered during the migration. For Flux-5 and Flux-10 this phase can be completed faster owing to the smaller number of pending tuples to execute, while in the Balanced heuristic the execution of the pending tuples lasts several seconds more. In the final part of the test (subplot in the figure), the Balanced strategy obtains a slightly lower latency compared to the other two heuristics, owing to better load balancing.

In conclusion, the small advantage of the Balanced strategy during the final part of the test is negligible with respect to the highest latency peaks and longest reconfiguration time. Therefore, this experiment shows that reconfigurations have substantial impact on the operator performance, and this can be reduced (but not eliminated) by minimizing of the number of migrated keys while achieving a sufficient load balancing.

C. Strategies Evaluation

In this part we evaluate the behavior of the MPC-based strategy. We study different instances of the strategy:

- ***NoSw***: MPC-based strategy without the switching cost term ($\gamma = 0$). The horizon length is set to one ($h = 1$), since it has no impact on reconfiguration decisions;
- ***Sw***: MPC-based strategy with switching cost enabled and horizon length of $h \geq 1$ steps.

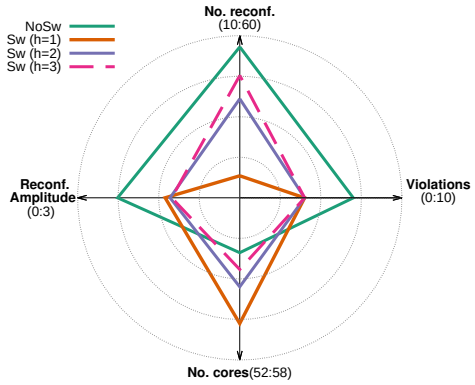
The cost parameters α, β and γ (in *Sw* case) are carefully chosen in order to: *i)* normalize the different terms of the cost function (Eq. 5); *ii)* give more priority to the QoS cost, i.e. the controller tries to reduce the number of QoS violations with minimal resource consumption.

In all the presented experiments the control step length is set of 3 seconds: for this application such time interval always guarantees the completion of the reconfiguration phase (e.g., of all the state migrations). Statistical predictions of disturbances (e.g., arrival rate) along the used horizon are computed using a set of time-series Holt Winters filters. All the experiments have been repeated 20 times by collecting the average measurements. The strategies are compared using four measures representing contrasting objectives of the adaptation process, i.e. *stability*, *accuracy*, *settling time* and *overshoot*.

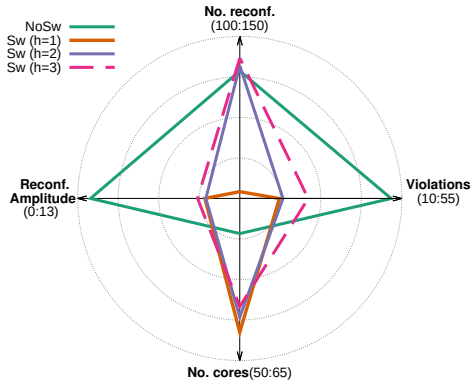
The accuracy indicates the count of QoS violations experienced by the operator using the elastic strategy, i.e. the number of control steps having an average latency higher than the desired threshold. In the experiment we use a value of $\delta = 5\ ms$ for the random walk workload and $\delta = 25\ ms$ for the real dataset (which is more challenging to sustain). Stability is expressed as the number of reconfigurations performed by the operator during the execution. The overshoot is the overall resource consumption, measured as the average number of cores utilized by the operator. Finally, the settling time is a measure of how fast the strategy is to reach a configuration satisfying the new workload conditions:strategies applying small reconfigurations (few workers added each time) obtain smaller settling time than strategies preferring large reconfigurations.

Figs. 6 reports the values of the four metrics for the *NoSw* strategy and different *Sw* strategies with various horizon lengths ($h = 1, 2, 3$). Each radar plot shows the averaged value of the measured metrics for the two different workloads.

Tab. I reports the numerical results for the real workload, which is the most interesting scenario. Here we also compare the behaviour of our strategies against: *i)* a *peak-load configuration* in which the operator is statically configured to use all the possible worker instances; *ii)* a *rule-based* approach (event-condition-action rules) [16] where the number of workers are increased/decreased every time the operator utilization goes over/under a maximum/minimum threshold. Qualitatively, we obtain the following behavior:



(a) Random walk.



(b) Real dataset.

Fig. 6: Results showing the properties obtained by different elastic strategies: accuracy, stability, overshoot and settling time. For each spoke, we indicate (in brackets) the minimum and maximum values that correspond respectively to the center and external edge of plot.

- the switching cost term allows reaching a better stability and to reduce the number of QoS violations;
- the switching cost with short horizons produces a high overshoot (cores utilized) and small settling time (average number of workers added/remove at each reconfiguration), as the operator uses more resources than necessary and it is slower in releasing/acquiring resources. This can be mitigated by using longer horizons provided that the disturbance predictions are sufficiently accurate (we limit the horizon length to $h = 3$ for this reason).

Our strategy allows the operator to obtain a number of QoS violations comparable (few additional violations) with the peak-load configuration by saving 45% of resources. The policy-based approach executes more reconfigurations and thus more QoS violations.

D. Analysis of Latency Variability

From the previous analysis we understand that the switching cost term in the formulation of the MPC optimization problem allows the controller to reduce the number of reconfigurations and their average amplitude. Reconfiguration stability and amplitude affect the quality of the latency results experienced by the user (or by the next operators in the operator graph). In many application domains, besides the average value of latency, it may be essential to keep the instantaneous latency as smooth as possible by avoiding large variations. This concept is captured by the *latency jitter* [17]. More precisely, we measure the so-called *instantaneous packet delay variation* (see RFC 3393 IETF), measured as the average variation of the latency measured for two consecutive results. This measure is a function of the processing load, queueing of tuples within the operators, and it is intuitively affected by the amount and amplitude of reconfigurations performed.

Strategy	Reconf.	Violat.	Cores	Amplit.
NoSw	139.33	52	53.25	11.88
Sw h=1	102.09	21	62.37	2
Sw h=2	140.93	22	60.98	1.93
Sw h=3	143.27	29	60	2.64
Rule based	147.2	43	46.36	3.09
Peak Load	-	19	99	-

TABLE I: Summary of the numerical results of the strategies.

Tab. II provides an interesting analysis of some derived metrics of the operator execution with the elastic strategies studied before. We focus on the real-world dataset which exhibits more wide and sudden variations in the input rate. The table reports for each strategy the jitter measured and the 95-th percentile of the latency results, in order to show an estimate of the latency variability. As it can be noticed, both the number of reconfigurations and their amplitude have substantial effect on the measured jitter. For example, despite the *NoSw* and *Sw* $h = 2$ strategies achieve a similar number of reconfigurations, the latter is able to achieve a lower jitter owing to the reduced reconfiguration amplitude. On the other side, if we consider the case *Sw* $h = 1$, which has a reconfiguration amplitude similar to *Sw* $h = 2$, it achieves the best jitter due to the lower number of performed reconfigurations. Similar conclusions can be drawn for the 95-th percentile values.

	NoSw	Sw			Rule based	Peak load
		$h = 1$	$h = 2$	$h = 3$		
Jitter	38.8	14.7	15.5	18.1	29.6	13.8
95th	341.9	56.3	63.6	78.8	153.9	45.1

TABLE II: Quality and variability of the latency results from the elastic operator. Measures are in milliseconds.

These results provide a further evidence of the effectiveness of our approach. The strategies with switching cost provide a jitter similar to the case of a peak-load configuration, where reconfigurations are never performed. Therefore, our strategies execute the strictly necessary reconfigurations to adapt to the actual incoming rate without consuming too many resources and with limited impact on latency variability.

VI. Related Works

In the recent years a large numbers of Stream Processing Engines (like Apache Storm, Apache Flink and IBM Infos-

phere Streams) have been proposed by open source projects, academia and big companies. Despite all of them exploit distributed hardware and tackle the continuous execution of DaSP applications, they do not provide elastic supports or the provided mechanisms are still embryonic.

Scaling decisions should be taken by the system automatically according to the actual workload level monitored at run-time. Some of the existing proposals are essentially *best effort* [13], [2], [18], [19] and throughput oriented. Instead, in this paper we focus on the elastic execution of operators particularly sensitive to latency.

In Ref. [20] the authors propose a method to dynamically adapt the parallelism degree to limit the length of the input queue of streaming operators. This gives bounds on the waiting time in queue but not on the experienced latency. Moreover, they assume that the input rate and processing time distributions are exponential or deterministic, assumptions that rarely meet the real world.

In Ref. [13] the authors study how to minimize latency spikes during the process of scaling decisions. This work does not enforce constraints on the average latency. As far as we know, the strategy proposed in Ref. [3] is the only past work designed to minimize latency violations. The approach is similar to the one in this paper (Queuing Theory models) but does not consider stateful operators. In this paper we integrate such models with a predictive control-based technique that takes in advance reconfiguration decisions using statistical forecasts of disturbances influencing the operator behavior.

Concerning the type of strategy used, the majority of these works propose a *reactive strategy*: the system tries to continuously match the amounts of demanded resources to react to a modification in the arrival rate. In contrast, the work in Ref. [18] is one of the few trying to apply a predictive approach leveraging the estimated knowledge of future resource and workload behavior to plan resources allocation. Our work tries to formalize this predictive behavior by introducing the horizon length, a further parameter to tune the foresight degree of the controller.

VII. Conclusions

This paper proposed an elastic strategy for latency-sensitive distributed data streams operators. The approach is aimed at controlling the average latency by taking optimal scaling decisions evaluated over limited future time horizons. Experiments proved the effectiveness of the approach in reducing the amount of QoS violations and reconfigurations performed. The final outcome is that the operator controlled with our strategy achieves more stable latency results with a limited jitter.

Future research directions of our work are oriented towards the integration of the approach in existing stream processing frameworks like Apache Storm.

Acknowledgments

This work has been partially supported by the EU H2020-ICT-2014-1 project RePhrase (No. 644235). We would like to thank Centro di Calcolo Scientifico - INFN Pisa (Director: Dott. Alberto Ciampa) for their willingness and kindness in let us use their cluster.

References

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

[2] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1447–1463, June 2014.

[3] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, June 2015, pp. 399–410.

[4] E. F. Camacho and C. A. Bordons, *Model Predictive Control in the Process Industry*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.

[5] T. De Matteis and G. Mencagli, "Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016, pp. 13:1–13:12.

[6] T. D. Matteis and G. Mencagli, "Proactive elasticity and energy awareness in data stream processing," *Journal of Systems and Software*, pp. –, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216301467

[7] T. De Matteis and G. Mencagli, "Parallel patterns for window-based stateful operators on data streams: An algorithmic skeleton approach," *International Journal of Parallel Programming*, pp. 1–20, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10766-016-0413-x

[8] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive workload classification and forecasting for proactive resource provisioning," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13. New York, NY, USA: ACM, 2013, pp. 187–198.

[9] J. F. C. Kingman, "On queues in heavy traffic," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 24, no. 2, pp. pp. 383–392, 1962.

[10] Google, "Protocol buffer," https://developers.google.com/protocol-buffers/, 2016.

[11] M. Danelutto and M. Torquati, "Structured parallel programming with "core" fastflow," in *Central European Functional Programming School*, ser. LNCS, V. Zsók, Z. Horváth, and L. Csató, Eds. Springer, 2015, vol. 8606, pp. 29–75.

[12] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin, "Flux: an adaptive partitioning operator for continuous query systems," in *Data Engineering, 2003. Proceedings. 19th International Conference on*, March 2003, pp. 25–36.

[13] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS)*. New York, NY, USA: ACM, 2014, pp. 13–22.

[14] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, Aug. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1012889.1012894

[15] J. Wuttke, "Lmfit a c library for levenberg-marquardt least-squares minimization and curve fitting," 2015. [Online]. Available: http://apps.jcns.fz-juelich.de/lmfit

[16] P. Lalanda, J. A. McCann, and A. Diaconescu, *Autonomic Computing - Principles, Design and Implementation.*, ser. Undergraduate Topics in Computer Science. Springer, 2013.

[17] D. Hay and G. Scalosub, "Jitter regulation for multiple streams," *ACM Trans. Algorithms*, vol. 6, no. 1, pp. 12:1–12:19, Dec. 2009.

[18] A. Kumbhare, Y. Simmhan, and V. Prasanna, "Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, May 2014, pp. 344–353.

[19] V. Cardellini, M. Nardelli, and D. Luzi, "Elastic stateful stream processing in storm," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 583–590.

[20] R. Mayer, B. Koldehofe, and K. Rothermel, "Predictable low-latency event detection with parallel complex event processing," *Internet of Things Journal, IEEE*, vol. 2, no. 4, pp. 274–286, Aug 2015.