The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-75178-8_9

Towards Hierarchical Autonomous Control for Elastic Data Stream Processing in the Fog

Valeria Cardellini ✉, Francesco Lo Presti,
Matteo Nardelli, and Gabriele Russo Russo

Department of Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata, Italy
{cardellini,nardelli}@ing.uniroma2.it, lopresti@info.uniroma2.it,
gab.russorusso@gmail.com

Abstract In the Big Data era, Data Stream Processing (DSP) applications should be capable to seamlessly process huge amount of data. Hence, they need to dynamically scale their execution on multiple computing nodes so to adjust to unpredictable data source rate. In this paper, we present a hierarchical and distributed architecture for the autonomous control of elastic DSP applications. It revolves around a two layered approach. At the lower level, distributed components issue requests for adapting the deployment of DSP operations as to adjust to changing workload conditions. At the higher level, a per-application centralized component works on a broader time scale; it oversees the application behavior and grants reconfigurations to control the application performance while limiting the negative effect of their enactment, i.e., application downtime. We have implemented the proposed solution in our distributed Storm prototype and evaluated its behavior adopting simple policies. The experimental results are promising and show that, even with simple policies, it is possible to limit the number of reconfigurations while at the same time guaranteeing an adequate level of application performance.

Keywords: Data stream processing, Self adaptive, Hierarchical control, MAPE loop

1 Introduction

Data Stream Processing (DSP) applications can continuously collect and process data generated by an increasing number of sensing devices, to timely extract valuable information in many application domains, including health-care, energy management, logistic, and transportation. These scenarios pose new challenges to DSP systems in terms of strict latency requirements in face of variable and high data volumes to process. To deal with operator overloading, a commonly adopted stream processing optimization is data parallelism, which consists in scaling-out or scaling-in the number of parallel instances for the operators, so that each instance can process a subset of the incoming data flow in parallel.

Recently, since data sources are in general geographically distributed (e.g., in IoT scenarios), we also have witnessed a paradigm shift with the deployment and execution of DSP applications over distributed Cloud and Fog computing resources, which *de facto* bring applications closer to the data, rather than the other way around, to improve application latency and make better use of the ever increasing amount of resources at the network periphery. Nevertheless, this very idea makes it difficult to control DSP application performance. Most of the approaches proposed in the literature (as detailed below) have been designed for cluster environments with a centralized control component overlooking the DSP operations. These solutions typically do not scale well in a distributed environment given the spatial distribution, heterogeneity, and sheer size of the infrastructure itself. While scalable decentralized solutions have been proposed, e.g., [12], their inherent lack of coordination might result in frequent reconfigurations which negatively affect the application performance due to continuous system downtime.

In this paper, to take the best of the two worlds, we propose a hierarchical distributed approach to the autonomous control of elastic DSP applications in Fog-based environment. Our contributions are as follows. We present in Section 2 a hierarchical distributed architecture for the autonomous control of elasticity, named *Elastic and Distributed DSP Framework* (EDF). The control is organized according to the Monitor, Analyze, Plan and Execute (MAPE) reference model for self-adapting systems. Specifically, the proposed architecture relies on a high-level centralized MAPE-based *Application Manager* that coordinates the run-time adaptation of subordinated MAPE-based *Operators Managers*, which, in turn, locally control the adaptation of single DSP operators.

As a second contribution, we present in Section 3 a simple reference control strategy for each component, we name the *local* (for the Operator Managers) and *global* policy (for the Application Manager), respectively. The first monitors and analyzes the operator performance to determine whether it needs to be reconfigured by scaling the number of replicas or by migrating a replica. The global policy identifies the most effective reconfigurations proposed by the Operator Managers, accepting or declining the proposed reconfigurations in order to control their number, and hence the application downtime.

As a third contribution, we have implemented EDF on our extension [1,2] of Apache Storm and evaluated the proposed solution on our prototype. We implemented two simple policies: the local policy employs a threshold approach to request operator reconfigurations to the Application Manager; the global policy adopts a token bucket scheme to control the number of allowed reconfigurations in any control interval. As shown in Section 4, our results are promising and show the effectiveness of the proposed solution in achieving a good trade-off between application performance and reconfiguration cost.

Related work Run-time adaptation of DSP applications achieved through elastic data parallelism is attracting many research and industrial efforts. Most approaches that enable elasticity are often implicitly organized as self-adaptive systems based on the MAPE model. Some works, e.g., [4,6,7], exploit best-effort

threshold-based policies based on the utilization of either the system nodes or the operator instances. The basic idea is that when the utilization exceeds the threshold, the replication degree of the involved operators is modified accordingly. Other works, e.g., [5,10,11,16], use more complex centralized policies to plan the scaling decisions. Lohrmann et al. [10] propose a strategy that enforces latency constraints by relying on a predictive latency model based on queueing theory. Stela [16] relies on throughput-based metric to identify those operators that need to be scaled-out/in. Heinze et al. [8] estimate latency spikes caused by operator reallocations through a model and use it to define a heuristic placement algorithm. In [1] we present a centralized optimization problem for the runtime elasticity management of DSP applications that minimizes migration costs while satisfying the application QoS requirements. Differently from the above works that present reactive scaling strategies, De Matteis and Mencagli [3] propose a proactive strategy that takes into account a limited future time horizon to choose the reconfigurations. However, all these works rely on a centralized planner for the run-time adaptation of DSP applications, that may suffer from network latencies in a geo-distributed operating environment. Mencagli [11] presents a game-theoretic approach where the control logic is distributed on each operator, but it is not integrated in a DSP system.

As regards the deployment of DSP applications in geo-distributed environments, we extended Apache Storm [2] with a self-adaptive and distributed placement heuristics [12], but it suffers from frequent and uncoordinated reconfigurations. SpanEdge [13] is implemented in Apache Storm, but it does not support operator migrations. Saurez et al. [14] propose a new Fog-specific programming model supporting the migration of application components.

2 System Architecture

2.1 Problem Definition

A DSP application can be regarded as directed acyclic graph (DAG), where data sources, operators, and sinks are connected by streams. An operator is a self-contained processing element that carries out a specific operation (e.g., filtering, POS-tagging), whereas a stream is an unbounded sequence of data (e.g., tuple). We distinguish between stateless and stateful operator whether the operator computes the output data using only the incoming data or also some internal state information, respectively. For the execution, multiple replicas can be used to run an operator, where each replica processes a subset of the incoming data flow. By partitioning the stream over multiple replicas, running on one or more computing nodes, the load per replica is reduced, which yields lower application latency. Since the load can vary over time, the number of replicas can change at run-time as to optimize some non-functional requirements. As infrastructure on which DSP applications are executed, we consider computing resources that are scattered in a geo-distributed environment as Fog computing.

For the execution, a DSP application needs to be deployed on computing resources, which will host and execute the operators. Since DSP applications are

usually long-running, the operators can experience changing working conditions (e.g., fluctuations of the incoming workload, variations in the execution environment). To preserve the application performance within acceptable bounds, their deployment should be adapted at run-time, through migration and scaling operations. A *migration* moves an operator replica to another computing resource, so to balance resource utilization. A *scaling* operation changes the replication degree of an operator: a scale-out decision increases the number of replicas when the operator needs more computing resources, whereas a scale-in decreases the number of replicas when the operator under-uses its resources. The drawback of reconfigurations is that they cause application downtime; hence, if applied too often, they negatively impact the application performance.

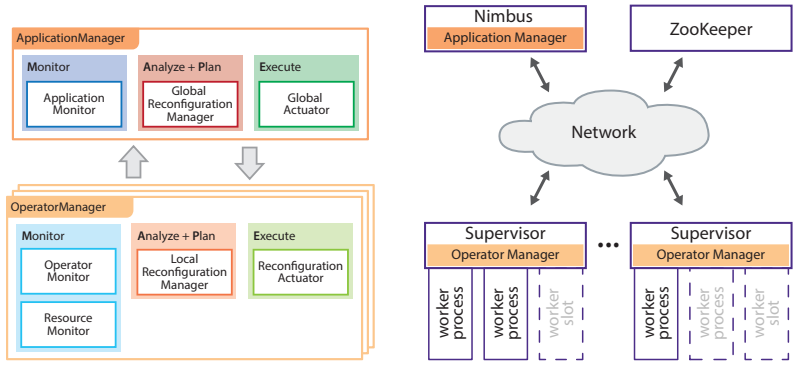
Being in charge of the application execution, the DSP system (e.g., Storm) can control the application performance. To agree on satisfying execution conditions, the user and the DSP system provider stipulate a Service Level Agreement (SLA). We consider that the SLA specifies as Service Level Objective (SLO) the maximum acceptable response time $R_{\max}$, that is the worst end-to-end delay from a data source to a data sink, and the maximum tolerable downtime during normal execution conditions. The latter indicates how often the application can be reconfigured when its response time is far from the critical value $R_{\max}$.

2.2 Hierarchical Architecture

The MAPE loop represents a prominent and well-know reference model to organize the autonomous control of a software system, where four components (Monitor, Analyze, Plan, and Execute) are responsible for the primary functions of self-adaptation. When the controlled system is geo-distributed as in Fog computing, a MAPE loop where analysis and planning decisions are centralized on a single component may not be sufficient for effectively managing the adaptation, because of the network latencies among the system components. As described by Weyns et al. in [15], different patterns to design multiple MAPE loops have been used in practice by decentralizing the functions of self-adaption.

When studying the strategies for placing DSP applications in a geo-distributed environment, we observed that a fully decentralized approach as in [2], where a multiplicity of peer MAPE loops autonomously manages the operator placement, may negatively affect the application performance, because of too frequent and uncoordinated decisions. This situation can be exacerbated when scaling operator decisions are involved besides those regarding the operator placement.

To address such lack of coordination in the multiple MAPE loops, in this paper we present a hierarchical distributed architecture, named *Elastic and Distributed DSP Framework* (EDF), for the autonomous control of elastic DSP applications in a Fog environment. The proposed solution is organized according to the hierarchical pattern for decentralized control described in [15], where higher-level MAPE components control subordinate MAPE components. Specifically, our proposal revolves around a two layered approach with separation of concerns and time scale between layers. Figure 1a illustrates the conceptual ar-



(a) The EDF conceptual architecture: hierarchical MAPE loops

(b) Implementation of the system architecture in Storm

Figure 1: System architecture

chitecture of EDF, highlighting the hierarchy of the multiple MAPE loops and the system components in charge of the MAPE loop phases.

At the lower level (i.e., at the per-operator grain) and a faster time scale, the *Operator Manager* is the distributed entity in charge of controlling the adaptation of a single DSP application operator/subset of the DSP application operators through a local MAPE loop. It monitors the system logical and physical components used by the operator(s) through the *Operator Monitor* and the *Resource Monitor*, and then, through the *Local Reconfiguration Manager*, it analyzes the monitored data and determines if and which local reconfiguration action (among operator scale-in, scale-out, or migration) is needed. When the Operator Manager determines that some adaptation should occur, it issues an operator adaptation request to the higher layer.

At the higher level (i.e., at the per-application grain) and a slower time scale, the *Application Manager* is the centralized entity that coordinates the adaptation of the overall DSP application through a global MAPE loop. By means of the *Application Monitor* it oversees the global application behavior. Then, through the *Global Reconfiguration Manager* it analyzes the monitored data and the reconfiguration requests received by the multiple Operator Managers, and decides which reconfigurations should be granted. These decisions are then communicated by the *Global Actuator* to each Operator Manager, which can, finally, execute the operator adaptation actions by means of the its local *Reconfiguration Actuator*.

The EDF architecture is general enough to not limit the specific internal policies and goals that can be designed for each component in the two layers. For example, the planning components can be either activated periodically or on event-basis, can rely on optimization problem formulation or heuristics with the goal to minimize the application response time, maximize its availability or a combination of the two. As a proof-of-concept of the proposed architecture, we

present, in Section 3, simple heuristic adaptation policies whose overall adaptation goal is to preserve the application performance, avoiding unnecessary or too frequent reconfigurations which might result in excessive application downtime.

We have implemented the proposed EDF architecture in Apache Storm, an open source, real-time, and scalable DSP system. Figure 1b shows the high-level instantiation of the EDF components on the Storm architecture. Due to space limitations, we omit a description of the basic Storm architecture and refer the reader to Section 6 in [1], where we also describe how to support in Storm elasticity mechanisms, including the migration of stateful operators. To obtain monitoring information (including network latencies) we rely on Distributed Storm [2].

3 Multi-level Elasticity Policy

The proposed two-layered architecture for self-adaptive DSP elasticity control identifies the different macro-components (i.e., Application Manager and Operator Managers) that, by means of abstraction layers and separation of concerns, cooperate to adapt the deployment of DSP applications at run-time. By properly selecting each component internal policy, the proposed solution can address the needs of different execution contexts, which can comprise applications with different requirements, infrastructures with different computing resources, and different user preferences. For example, specific policies can execute the application by minimizing its response time, maximizing its availability, or limiting the adaptation efforts (i.e., executing the application in a best-effort manner). The Operator Manager works at the granularity of a single DSP operator and implements what we called a *local policy*. By monitoring and analyzing the performance of each operator replica, the local policy can plan a reconfiguration of number and location of the operator replicas. Specifically, by scaling the number of replicas, the operator exploits parallelism to quickly process its incoming data, whereas by migrating some of the operator replicas, the operator better distributes the incoming load among computing resources. The Operator Manager sends the planned reconfiguration to the Application Manager, which runs periodically and decides, according to its so called *global policy*, which reconfiguration should be enacted. The global policy works at the granularity of the whole application, thus it coordinates the reconfigurations so to limit them and avoid deployment oscillations, if needed. On the basis of the monitored application performance and the stipulated SLA, the global policy identifies the most effective reconfigurations proposed by the Operator Managers: it accepts or declines each reconfiguration with the aim to adapt the DSP application to changing working conditions while meeting the SLA.

3.1 Local Policy

The Operator Manager local policy implements the Analyze and Plan phases of the decentralized MAPE loop, which controls the execution of a single DSP operator. Running on a decentralized component, this policy has only a local

view of the system, which results from the monitoring components (i.e., Operator Monitor and Resource Monitor). The local view consists of the status (i.e., resource utilization) of each operator replica and of a restricted suitable set of computing nodes (i.e., located in the neighborhood). By analyzing this information, the policy can plan a reconfiguration of the operator deployment, either by changing the number of replicas, or by migrating some of them. The proposed reconfiguration plan is then communicated to the centralized Application Manager which, based on all the Operator Manager's reconfiguration plans and the global policy, determines which plan can be executed and which not.

Reconfiguration Plan. A reconfiguration plan is expressed through the following information: *adaptation actions*, *reconfiguration gain*, and *reconfiguration cost*[1]. We consider two types of adaptation actions: replica migration and operator scaling. Actions can be of the form: "move replica α of *op* from r_i to r_j", "add a new replica to *op* on r_i", or "remove replica α of *op* from r_i", where *op* and r_i denote an operator and a computing resource, respectively. The *reconfiguration gain* is a function, adopted by every Operator Manager, which captures the benefits of the planned adaptation action. It can express, for instance, the reduction of the operator's processing latency, the reduction of monetary cost for running the operator, or the improvement of some utility function. In this paper, we assume a simple gain function that induces an order relation among the reconfiguration actions, namely `scale-out` > `migration` > `scale-in`. The *reconfiguration cost* expresses the cost of reconfiguring the system. In this paper, we express it in terms of application downtime. It results from the time required to add/remove an operator replica, to relocate the operator code, and to migrate its internal state (if any). We now discuss the two types of adaptation action.

Replica Migration. A computing resource can host replicas of one or more operators, which, in turn, are controlled by dedicated Operator Managers. When the computing resource becomes overloaded, the hosted replicas can experience a performance degradation. To overcome this issue, an Operator Manager proposes to move some of the operator replicas away from the resource.

We adopt a reactive and threshold-based policy in order to decide when and how to perform the migration. The local policy analyzes the monitoring data coming from the computing resources that host at least one operator replica. We denote with U_r the overall CPU utilization of the resource r. When U_r is above a critical value $U_{\max}$, the policy plans to migrate at most one operator replica to a new location. The latter is identified in two steps. First, the policy sorts the known neighbor resources according to their distance, measured in terms of network delay. Then, it selects the new location using a randomized approach: the closer the resource, the higher the probability of being selected. The policy checks if the new selected location has room to run the migrating replica; in negative case, a new resource is selected from the sorted list.

Reconfiguration Cost. If the operator is stateless, the migration of a replica can be easily performed by terminating the replica on the old location, moving

[1] For the sake of simplicity, we assume that the local policy proposes, for an operator, a single reconfiguration decision (i.e., migration, scaling) at a time.

its code to the new location, and restarting it. On the other hand, if the operator is stateful, we also need to efficiently migrate its internal state, so to preserve the integrity and consistency of the outputted streams. Our migration protocol follows a pause-and-resume approach with the help of a data store as staging area for the replica internal state (details on our migration protocol in [1]).

Operator Scaling. When an operator replica receives an increasing workload, it can saturate the capacity of the hosting computing resource. To prevent the performance penalty associated to overloading, the Operator Manager proposes to add an additional replica and redistribute the incoming workload accordingly. Conversely, when the incoming workload decreases, the Operator Manager can reduce the number of replicas in order to decrease the number of allocated resources, and redistribute the workload among the remaining ones. Let us denote by S_α the resource utilization of the hosting resource by replica α, which measures the fraction of CPU time used by α. We adopt a simple threshold-based scale-out policy to each replica. When the utilization of α exceeds a usage threshold $S_{\text{s-out}} \in [0, 1]$ (i.e., $S_\alpha > S_{\text{s-out}}$), the Operator Manager proposes to add a new replica. Its placement is computed relying on the same strategy used for the replica migration. Conversely, the Operator Manager proposes a scale-in operation, which removes one of the running n replicas, when the sum of their utilization divided by $n-1$ is significantly below the usage threshold, i.e., when $\sum_{\alpha=1}^{n} S_\alpha/(n-1) < cS_{\text{s-out}}$, being $c < 1$. The replica to be removed is randomly chosen between the two replicas with the highest utilization.

Reconfiguration Cost: If the operator is stateless, a scaling operation implies only to start or stop a replica. Conversely, if the operator is stateful, we also need to reallocate its internal state among the new set of replicas. We assume that each replica can work on a well-defined state partition [5]. A scale-out operation redistributes equally the partitions among replicas, whereas a scale-in operation aggregates the partitions from the merged replicas.

3.2 Global Policy

The Application Manager global policy implements the Analyze and Plan steps of the centralized MAPE loop. Its main goal is to satisfy the DSP application SLA, while minimizing the allocated resources (or their cost). To this end, it monitors the application response time and analyzes its behavior with respect to the SLO specified in the SLA. In the planning phase, the policy determines which reconfiguration plans, proposed by the decentralized Operator Managers, should be enacted as to improve performance while controlling the number of application reconfigurations (which cause application downtime). In this paper, we consider a simple global policy scheme which is exemplified in Figure 2. Time is divided in control intervals of fixed length T. During each interval, the global policy collects reconfiguration requests from the Operator Managers: these requests can take different forms, e.g., replica migrations (the continuous arrows in the figure), operator scale-out (the dotted arrow) and operator scale-in (the dashed arrow). At the end of each interval, the policy determines how many and which reconfigurations should be enacted by the Operators Managers. In order

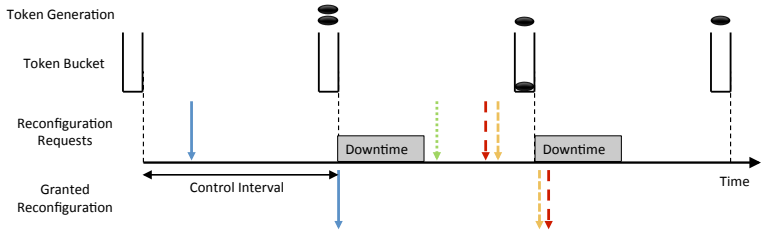


Figure 2: Global policy behavior

to control the number of reconfigurations, and hence the downtime, we adopt a simple token bucket scheme whereby each reconfiguration consumes a token. Tokens are generated at the end of each control interval T and are accumulated in a token bucket, which has a finite capacity (i.e., when the bucket is full, it cannot store any other token). The number of reconfigurations allowed at the end of each control interval is thus limited by the number of available tokens. If the number of requests is higher than the number of available tokens, the global policy has to identify the most valuable reconfigurations to accept. As simple scheme, the policy uses a greedy approach by prioritizing the requests according to the gain to cost ratio; the higher this index, the better the reconfiguration.

In the proposed scheme, a key role is played by the token generation rate. Ideally, when the application response time is well within the SLO (defined by $R_{\max}$), reconfigurations should be limited since performance is guaranteed and the possibly sub-optimal behavior is preferable to the downtime caused by reconfigurations. On the other hand, should the performance degrades, the system should be more prone to reconfigure itself. As such, the token generation frequency depends on how far is the response time from $R_{\max}$, with increasing token generation rates as performance gets close to $R_{\max}$.

4 Evaluation

We evaluate EDF equipped with the proposed proof-of-concept policies, using Apache Storm 0.9.3 on a cluster with 5 worker nodes and one further node to host Nimbus and ZooKeeper (details in [1]). Each node has a dual CPU Intel Xeon E5504 (8 cores at 2 GHz) with 16 GB of RAM.

The reference application solves a query of DEBS 2015 Grand Challenge [9], where data streams originated from the New York City taxis are processed to find the top-10 most frequent routes during the last 30 min. Figure 3 shows the application DAG. *Data source* reads the dataset from Redis; *parser* filters out irrelevant and invalid data. Then, *filterByCoordinates* forwards only the events related to a specific area to *computeRouteID*, which identifies the routes covered by taxis. So, *countByWindow* computes the route frequency in the last 30 minutes, supported by *metronome* that defines the passing of time. Finally, *partialRank* and *globalRank* compute the top-10 most frequent routes.

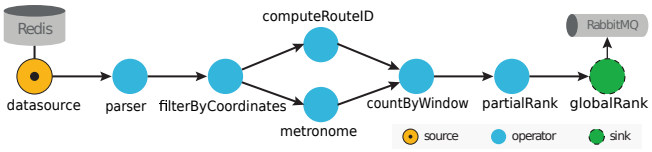


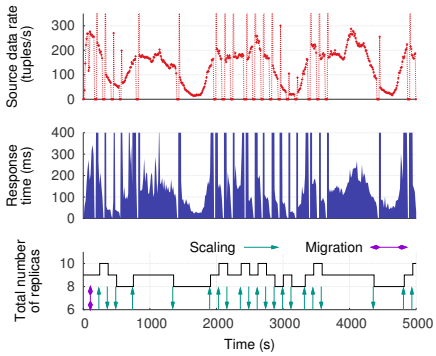
Figure 3: Reference DSP application

We feed the application with a sample dataset provided by DEBS, and process real data collected during 2 days. The taxi service utilization significantly changes during the day, thus the application input rate is variable as well. As regards the Operator Manager local policy, we set the scale-out and migration thresholds, $U_{\max}$ and $S_{\text{s-out}}$, to 0.7 and the scale-in parameter c to 0.75. Both OperatorManager and ApplicationManager run once every 30 s, respectively proposing and accepting/rejecting reconfigurations. We compare the baseline approach in which all reconfiguration requests are always accepted by the ApplicationManager to one in which the global policy in Section 3.2 is employed in order to determine which reconfigurations will be enacted. In particular, the token bucket stores at most 1 token at any time and the token generation rate is 1 per min only if the achieved application response time is above $\beta R_{\max}$, where $\beta \in [0, 1]$, otherwise no token is generated. In these experiments we set $R_{\max} = 200$ ms and vary β.

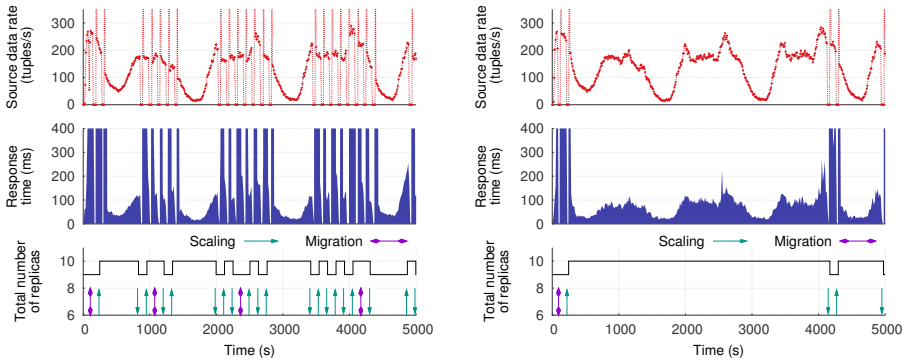
Figure 4a shows the application response time and number of replicas during the experiment when using the baseline approach. Since every reconfiguration proposed by any OperatorManager is accepted (like in a fully decentralized policy), the application is frequently reconfigured. As a consequence, the application is available only for 93.7% of the time. The measured response time shows many spikes, which are caused by tuples buffering during reconfiguration.

Figure 4b shows the application response time and number of operator replicas during the experiment using the full reconfiguration policy, with $\beta = 0.5$. As the response time frequently rises above $\beta R_{\max} = 100$ ms, the number of granted reconfigurations is not significantly reduced with respect to the baseline approach in Figure 4a (and so the application downtime). Nevertheless, we can observe that, by performing less reconfigurations, the total number of replicas is never reduced to 8, due to the lack of tokens and the low priority of the scale-in action.

Figure 4c shows the results when $\beta = 0.75$. As tokens are now generated in a more conservative manner (being $\beta R_{\max} = 150$ ms), the number of reconfigurations is significantly reduced. In the initial part of the experiment, the input rate grows up to 300 tuples/s, resulting in high response time; therefore, EDF generates tokens for performing a migration and for increasing the total number of replicas to 10. Then, the application is stable until a new input peak (at around 4000 s), when a scale-in followed by a scale-out of the bottleneck operator are accepted. The application downtime is limited (only 1.7%), which is beneficial for response time, but it might lead to higher cost, having more active replicas.



(a) All reconfigurations



(b) $\beta = 0.50$ (c) $\beta = 0.75$

Figure 4: Response time and number of replicas using different policies for ApplicationManager: in (a) accepting all the reconfiguration requests, in (b) and (c) generating a token only when response time is greater than $\beta R_{\max}$

5 Conclusions

In this paper, we presented Elastic and Distributed DSP Framework (EDF), a hierarchical autonomous control for elastic DSP applications. Designed according to the decentralized MAPE control pattern, our proposal revolves around a two layered approach with separation of concerns and time scale between layers. At the lower level, distributed components control the adaptation of DSP operators, so to improve their performance by means of scaling and migration actions. At the higher level, a per-application centralized component oversees the overall DSP application performance and coordinates its deployment by accepting or declining the proposed reconfiguration actions. Then, relying on an application that processes real-time data generated by taxis, we conducted an experimental evaluation. The results showed the effectiveness of our solution in achieving good trade-off in terms of application performance and number of application reconfigurations even adopting simple control policies. As future work,

we will further investigate the hierarchical approach for adapting DSP applications over geo-distributed infrastructures. We plan to extend some of the existing distributed policies to make them more robust to oscillations, and to design hierarchical multi-time scale policies relying on optimization frameworks such as Markov Decision Processes and reinforcement learning.

References

1. Cardellini, V., Lo Presti, F., Nardelli, M., Russo Russo, G.: Optimal operator deployment and replication for elastic distributed data stream processing. Concurr. Comput.: Pract. Exper. (2017), to appear
2. Cardellini, V., Grassi, V., Lo Presti, F., Nardelli, M.: Distributed QoS-aware scheduling in Storm. In: Proc. of ACM DEBS '15. pp. 344–347 (2015)
3. De Matteis, T., Mencagli, G.: Elastic scaling for distributed latency-sensitive data stream operators. In: Proc. of PDP '17. pp. 61–68 (2017)
4. Fernandez, R.C., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: Integrating scale out and fault tolerance in stream processing using operator state management. In: Proc. of ACM SIGMOD '13. pp. 725–736 (2013)
5. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. 25(6), 1447–1463 (2014)
6. Gulisano, V., Jiménez-Peris, R., Patiño Martínez, M., Soriente, C., Valduriez, P.: StreamCloud: An elastic and scalable data streaming system. IEEE Trans. Parallel Distrib. Syst. 23(12), 2351–2365 (2012)
7. Heinze, T., Pappalardo, V., Jerzak, Z., Fetzer, C.: Auto-scaling techniques for elastic data stream processing. In: Proc. of IEEE ICDEW '14. pp. 296–302 (2014)
8. Heinze, T., Roediger, L., Meister, A., Ji, Y., et al.: Online parameter optimization for elastic data stream processing. In: Proc. of ACM SoCC '15. pp. 276–287 (2015)
9. Jerzak, Z., Ziekow, H.: The DEBS 2015 grand challenge. In: Proc. of ACM DEBS '15. pp. 266–268 (2015)
10. Lohrmann, B., Janacik, P., Kao, O.: Elastic stream processing with latency guarantees. In: Proc. of IEEE ICDCS '15. pp. 399–410 (2015)
11. Mencagli, G.: A game-theoretic approach for elastic distributed data stream processing. ACM Trans. Auton. Adapt. Syst. 11(2) (2016)
12. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., et al.: Network-aware operator placement for stream-processing systems. In: Proc. IEEE ICDE '06 (2006)
13. Sajjad, H.P., Danniswara, K., Al-Shishtawy, A., Vlassov, V.: Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In: Proc. of 2016 IEEE/ACM Symp. on Edge Computing. pp. 168–178 (2016)
14. Saurez, E., Hong, K., Lillethun, D., Ramachandran, U., et al.: Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In: Proc. of ACM DEBS '16. pp. 258–269 (2016)
15. Weyns, D., Schmerl, B., Grassi, V., Malek, S., et al.: On patterns for decentralized control in self-adaptive systems. In: Software Engineering for Self-Adaptive Systems II, LNCS, vol. 7475, pp. 76–107. Springer (2013)
16. Xu, L., Peng, B., Gupta, I.: Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In: Proc. of IEEE IC2E '16. pp. 22–31 (2016)