Future Generation Computer Systems 87 (2018) 171–185

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Decentralized self-adaptation for elastic Data Stream Processing

Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli *, Gabriele Russo Russo

Department of Civil Engineering and Computer Science Engineering, University of Rome Tor Vergata, Italy

HIGHLIGHTS

- Hierarchical control architecture for elastic DSP applications can improve performance and scalability without compromising stability.
- Lightweight global control policies can overcome the lack of coordination of fully decentralized solutions.
- Model-based Reinforcement Learning algorithms outperform classical model-free algorithms like Q-learning.
- Reinforcement Learning policies allow the users to focus on what they aim to obtain, instead of how it should be obtained.

ARTICLE INFO

Article history:
Received 17 November 2017
Received in revised form 7 April 2018
Accepted 11 May 2018
Available online 15 May 2018

Keywords:
Data stream processing
Self adaptive
Hierarchical control
MAPE
Reinforcement learning

ABSTRACT

Data Stream Processing (DSP) applications are widely used to develop new pervasive services, which require to seamlessly process huge amounts of data in a near real-time fashion. To keep up with the high volume of daily produced data, these applications need to dynamically scale their execution on multiple computing nodes, so to process the incoming data flow in parallel. In this paper, we present a hierarchical distributed architecture for the autonomous control of elastic DSP applications. It consists of a two-layered hierarchical solution, where a centralized per-application component coordinates the run-time adaptation of subordinated distributed components, which, in turn, locally control the adaptation of single DSP operators. Thanks to its features, the proposed solution can efficiently run in large-scale Fog computing environments. Exploiting this framework, we design several distributed self-adaptation policies, including a popular threshold-based approach and two reinforcement learning solutions. We integrate the hierarchical architecture and the devised self-adaptation policies in Apache Storm, a popular open-source DSP framework. Relying on the DEBS 2015 Grand Challenge as a benchmark application, we show the benefits of the presented self-adaptation policies, and discuss the strengths of reinforcement learning based approaches, which autonomously learn from experience how to optimize the application performance.

1. Introduction

The ubiquitous presence of sensing devices, which continuously produce streams of data, creates a fertile ground for the development of new and pervasive services. An efficient use of those data can improve the quality of everyday life in many cross-concern domains, including health-care, energy management, logistic, and transportation. Data Stream Processing (DSP) represents a prominent approach to elaborate data as soon as they are generated, thus enabling the design of near real-time applications.

A DSP application is represented as a directed acyclic graph (DAG), with data sources, operators, and final consumers as vertices, and streams as edges. Each *operator* can be seen as a black-box processing element that continuously receives incoming streams, applies a transformation, and generates new outgoing streams. In modern scenarios, DSP applications are characterized by 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 enables to process data in parallel on multiple computing nodes (given that a single machine cannot provide enough processing power). Data parallelism consists in scaling-out or scaling-in the number of parallel instances for the operators, so that each instance processes a subset of the incoming data flow (e.g., [1]). As the application workloads are typically highly variable, the application parallelism should *elastically* self-adapt at run-time to match the workload and prevent resource wastage.

Moreover, since data sources are in general geographically distributed (e.g., in IoT scenarios), recently we also have witnessed

* Corresponding author.
E-mail addresses: cardellini@ing.uniroma2.it (V. Cardellini), lopresti@info.uniroma2.it (F. Lo Presti), nardelli@ing.uniroma2.it (M. Nardelli), russo.russo@ing.uniroma2.it (G. Russo Russo).

https://doi.org/10.1016/j.future.2018.05.025
0167-739X/© 2018 Elsevier B.V. All rights reserved.

a paradigm shift with DSP applications being deployed over distributed Cloud and Fog computing resources. This computing environment *de facto* brings applications closer to the data, rather than the other way around, to reduce application latency and make better use of the ever increasing amount of resources at the network edges. Nevertheless, this very idea makes it challenging to control DSP application performance. Most of the approaches proposed in literature for the DSP application deployment and adaptation have been designed for cluster environments with a centralized control component, that can benefit from a global system view. These solutions typically do not scale well in a highly distributed environment, given the spatial distribution, heterogeneity, and sheer size of the infrastructure itself. In fact, modern DSP systems should be able to seamlessly deal with a large number of interconnected small and medium size devices (e.g., IoT devices), which continuously emit and consume data (e.g., in a smart city). To improve scalability, several decentralized management solutions have been proposed, e.g., [2–4]. Devising a decentralized policy that reconfigures the DSP application deployment exploiting only a local system view is, in general, not trivial. Indeed, the inherent lack of coordination of decentralized solutions might result in frequent reconfigurations that negatively affect the application performance (e.g., [5]).

Aiming to exploit the strengths of centralized and decentralized solutions, in our previous work [6], we proposed a hierarchical distributed architecture for controlling the elasticity of DSP applications. The control is organized according to the Monitor, Analyze, Plan and Execute (MAPE) architectural pattern for self-adaptive systems [7], which has been often adopted in Cloud auto-scaling systems [8]. Specifically, the proposed architecture relies on a two-layered hierarchical solution, where a high-level centralized MAPE-based *Application Manager* coordinates the run-time adaptation of subordinated MAPE-based *Operators Managers*, which, in turn, locally control the adaptation of single DSP operators. We originally proposed a simple yet effective hierarchical policy that combines a threshold-based policy, for locally controlling the adaptation of DSP operators, and a token-bucket policy, for coordinating the overall application adaptation (see Section 4 for details).

In this paper, encouraged by the positive preliminary results, we further explore hierarchical approaches for self-adapting DSP applications at run-time. Differently from the popular threshold-based approaches used to drive the operator elasticity (e.g., [1,9–12]), we seek to design an adaptive approach, able to customize the adaptation policy for each DSP operator, without the need of manually tuning various configuration knobs. To this end, we resort on machine learning techniques and, in particular, on Reinforcement Learning (RL). RL refers to a collection of trial-and-error methods by which an agent can learn to make good decisions through a sequence of interactions with a system or environment [13]. As such, RL allows to express *what* the user aims to obtain, instead of *how* it should be obtained (as required by threshold-based policies). The adaptive nature of RL makes it very appealing to devise auto-scaling policies [8,14]. Nevertheless, to the best of our knowledge, only the work by Heinze et al. [12] has so far exploited RL techniques in DSP systems. In the context of elasticity, an RL algorithm should learn when and how to change the number of operator replicas, so to efficiently handle the incoming load variations while avoiding resource wastage. One of the main issues with RL policies is the possibly long learning phase, which is especially experienced when the algorithm assumes that nothing about the system dynamics is known *a priori* (*model-free* learning). An approach to boost the learning process is to provide the learner with basic knowledge about its environment (*model-based* learning). In a preliminary work [15], we started to investigate RL-based elasticity policies while considering a single DSP operator in isolation. Motivated by the positive numerical results, in this paper we propose two RL based algorithms, that rely on different levels of system knowledge for controlling elasticity of DSP applications with many interconnected operators. First, we design a model-free learning algorithm for controlling elasticity, resorting to the well-known Q-learning algorithm. Then, we present a model-based approach that exploits what is known or can be estimated about the system dynamics, thus making the learner's task easier.

The main contributions of this paper are as follows.

- We discuss several design patterns for realizing decentralized control architectures for DSP systems, and identify the most suitable approach for Fog-based environments. Then, we describe in detail *Elastic and Distributed DSP Framework* (EDF), our hierarchical distributed architecture for the autonomous control of elastic DSP applications (Section 3).
- We design new hierarchical solutions for controlling the adaptation of DSP operators (local policies) and for coordinating the overall number of reconfigurations (global policies). As local policies, we propose a threshold-based policy and two RL-based policies, which, respectively, leverage a model-free and a model-based learning approach. As global policy, we propose a centralized token bucket based solution that solves conflicts and limits the number of reconfigurations (Sections from 4 to 7).
- We present a prototype implementation of EDF in Apache Storm, a well-known open source DSP framework, where the deployment of DSP applications is adapted at run-time using the designed control policies (Section 8).
- We evaluate our solution through experiments based on our EDF prototype (Section 9). To this purpose, we use a DSP application that solves the DEBS 2015 Grand Challenge [16] and works with real data streams originated from the New York City taxis.

2. Related work

DSP applications are usually long running and expose computational requirements that are usually unknown and, most importantly, can change continuously at runtime. Therefore, in the last years, research and industrial efforts have investigated the runtime adaptation of DSP applications achieved through elastic data parallelism.

System architecture. Most approaches that enable elasticity are often implicitly architected as self-adaptive software systems based on the MAPE loop, a well known pattern to design self-adaptive systems [7]. Following this model, the current application deployment is adapted by scaling-in/out the number of operator replicas and/or rescheduling the application in response to changes, either observed or predicted, in some monitored performance metrics.

Most of the existing system architectures rely on a centralized management solution, where a single coordination entity uses its knowledge about the entire system state so to plan the proper adaptation actions (e.g., [9,17–22]). Although this approach can potentially achieve a global optimum adaptation strategy, it may be not suitable for a geo-distributed environment, because a central manager represents a bottleneck in large-scale systems due to monitoring and planning overheads. Conversely, other solutions rely on decentralized adaptation planners that exploit a limited local view of the system, thus overcoming the scalability issues. In this case, the vast majority of the proposed approaches rely on a fully decentralized architecture (e.g., [2–4,23]). Although these decentralized solutions appear to be appealing for geo-distributed environments (like Fog computing), developing fully decentralized adaptation policies is a non trivial task. Some works recur to the exploitation of mathematical properties of specific deployment goals [4]. Nevertheless, in general, the lack of coordination among the decentralized agents may cause frequent reconfiguration decisions, which can cause instability that negatively affects the application performance (as shown in [5]).

Differently from the above approaches, we propose a hierarchical distributed architecture. We believe that the latter can take the best of centralized and fully decentralized architectures, thus improving performance and scalability without compromising stability. In this paper, we build on our previous work [6], where we firstly presented the hierarchical decentralized control architecture. Motivated by the positive preliminary results, we here slightly refactor the proposed architecture and, above all, develop and evaluate new elasticity control policies.

Policies for run-time adaptation. Several elasticity policies have been proposed so far in literature. Some works, e.g., [1,9–12], adopt simple decentralized threshold-based policies that use the utilization level of either the system nodes or the operator instances. The basic idea is that when the node or operator utilization exceeds the threshold, the replication degree of the involved operators is modified accordingly. Different approaches can be identified to define the threshold. A single statically-defined threshold is used, e.g., in [9] to limit load unbalance among computing nodes. Multiple statically-defined thresholds can also be used so to customize the behavior of each individual node within the system. A dynamically configured threshold improves the system adaptivity, e.g., in [12,24].

Other works, e.g., [17,25–28], use more complex centralized policies to determine the scaling decisions, exploiting optimization methods that rely on a global model, such as integer linear programming [17], control theory [25], queueing theory [26], and fuzzy logic [27]. In [17], we presented an integer linear programming problem for the runtime elasticity management of DSP applications that minimizes reconfiguration costs while satisfying the application QoS requirements. Lohrmann et al. [26] proposed a strategy that enforces latency constraints by relying on a predictive latency model based on queueing theory. Mencagli et al. [27] presented a two-level adaptation solution that handles workload variations at different time-scales: at a fast time-scale a control-theoretic approach is used to deal with load imbalance, while at a slower time-scale a global controller makes operator scaling decisions employing fuzzy logic. However, their solution is tailored to sliding-window preference queries executed on multicore architectures. While the previously mentioned approaches are reactive and cannot thus provision replicas in advance, De Matteis and Mencagli [25] proposed a proactive control-based strategy that takes into account a limited future time horizon to choose the reconfigurations. Centralized heuristic policies have been also proposed in [22,28,29]. Liu et al. [22] proposed a stepwise profiling framework that considers both application features and processing power of the computing resources and selectively evaluates the efficiency of several possible configurations of parallelism. Stela [28] relies on a throughput-based metric to estimate the impact of each operator towards the application throughput and identify those operators that need to be scaled. Kotto Kombi et al. [29] proposed an approach to preventively adapt the replication degree of operators according to stream rate fluctuations. Differently from the above centralized policies, Mencagli [23] presented a game-theoretic approach where the control logic is distributed on each operator; however, it is not integrated in a DSP system. In [6], we designed a hierarchical policy where a decentralized threshold-based approach works in combination with a centralized token-bucket solution. The former is in charge of adapting the operator replication degree in a fully decentralized manner, whereas the latter controls and limits the number of reconfiguration exploiting the global view on the application performance.

In this work, we design more sophisticated approaches based on RL. While the exploitation of RL for driving the elastic execution of DSP applications is considered worth of investigation [14], to the best of our knowledge, only the work by Heinze et al. [12] has so far proposed a RL-based elasticity policy. They rely on a simple model-free RL algorithm that learns when to acquire and release computing resources, so to efficiently process the incoming workload. Being model-free, the RL algorithm requires no knowledge of the system dynamics; nevertheless, this leads to slow convergence properties, because the learner also needs to experience ineffective actions (e.g., reducing the number of replicas when the system is overloaded). We started to investigate RL-based elasticity policies in [15]. Differently from [12], our preliminary solution is in charge of self-configuring the number of replicas of just a single DSP operator (rather than of a DSP application composed of many interconnected operators, as in this paper). Our numerical evaluation reveals that, by exploiting some knowledge about the system dynamics, we can overcome some popular drawbacks of RL-based solutions and achieve faster convergence rate. This result motivates us to further explore this research direction. Therefore, in this paper we propose two RL-based policies for elastic DSP applications that aim to limit the application response time and rely on different levels of system knowledge. We design a model-free learning algorithm, which uses the well-known Q-learning algorithm [13] and adopts a cost function driven by the operator's response time violation. Similarly to [12], the learning is model-free, but our algorithm controls the operator elasticity driven by the operator response time rather than scaling the underlying computing nodes driven by their utilization. Moreover, we present a full backup model-based approach that, by instilling knowledge on the estimated system dynamics, effectively addresses the slow convergence of the model-free learning algorithm.

DSP frameworks. Aside the specific functionalities, the most popular open-source DSP frameworks (Storm, Spark Streaming, Flink, and Heron) provide an abstraction layer to develop DSP applications focusing solely on the application logic. The tasks related to the application distribution, execution, and adaptation are managed by the frameworks themselves.

As regards the operator elasticity, in most cases these frameworks require their users to manually tune the number of replicas per operator. Since the user might over-/under-estimate the expected load, this approach can lead to sub-optimal application performance and operating costs. Therefore, manual tuning of configuration knobs is perceived as a difficult yet crucial task by users and developers of DSP applications [14]. Furthermore, most of these frameworks are equipped with elasticity mechanisms in an embryonic stage; indeed, they dynamically scale the application in a disruptive manner, because they enact reconfigurations by killing and restarting the whole application, thus introducing a significant downtime.

We provide an overview of Storm in Section 8.1, because we implement the proposed EDF architecture in it. Several research efforts have used Storm to evaluate scheduling algorithms or architectural improvements (e.g., [18,19,21,30,31] and, from our research group, [10,17,5]). In [10,17], we extended Storm to support the elastic run-time adaptation of DSP applications, by introducing new system components that allow the elasticity and stateful migration of DSP operators. An approach to support elastic scaling of DSP applications in Storm has been also presented in [31]. Interestingly, their proposal reduces the downtime due to the reconfiguration process by keeping the application running while scaling the application operators (instead of shutting down and restarting them). However, their improved version of Storm has not been released publicly.

Developed by Twitter as the successor of Storm, Heron [32] preserves Storm's abstraction layer while introducing some improvements and a multi-layer architecture. Dhalion [33] is a framework on top of Heron that provides elastic capabilities to the underlying streaming system; it also allows to reconfigure at run-time the application without introducing downtime. However, its current elasticity policy simply adjusts the replication degree of an operator so to satisfy its throughput; anyway, the investigation of reinforcement learning techniques in Dhalion is considered as an exciting area for future research [14].

Spark Streaming [34] is an extension on top of Apache Spark that enables data stream processing. It is throughput-oriented, whereas Storm can minimize the application latency and can thus be preferable in latency-sensitive scenarios. From version 2.0, Spark Streaming supports elastic scaling through the dynamic allocation feature, which uses a simple heuristic where the number of executors is scaled up when there are pending tasks and is scaled down when executors have been idle for a specified time. Another emerging framework is Apache Flink [35], which provides a unified solution for batch and stream processing. Although Flink supports the manual scaling of operators and state management, it does not yet provide any auto-scaling capability [36].

Despite the recent efforts towards elasticity in some frameworks, all those cited are not designed to efficiently operate in a geo-distributed environment. At this regards, we proposed Distributed Storm [5], an extension of Apache Storm that introduces self-adaptive and distributed scheduling capabilities. In [17], we integrated elasticity and stateful migration capabilities in Distributed Storm. SpanEdge [37] is implemented in Apache Storm, but it does not support operator migrations. Saurez et al. [38] proposed a new Fog-specific programming model supporting the migration of application components.

3. System architecture

3.1. Problem definition

A DSP application can be regarded as a 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 performed computation involves only the input data or also some internal state information, respectively.

DSP applications are usually employed in latency-sensitive domains [16,25,26], where reduced response time is required. Although multiple definitions of response time exist, the widely used one defines it as the overall processing and transmission latency from a data source to a final consumer on the application DAG. In this work, we will assume that the DSP application exposes requirements on its response time, in terms of target value R_{max} which should not be exceeded. To improve the performance, 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, and so is the processing latency. Since the workload usually varies over time, the number of replicas should accordingly change at run-time as to meet the performance target while avoiding resource wastage.

For the execution, a DSP application needs to be deployed on computing resources, which will host and execute the operators. In particular, we consider computing resources that are scattered in a geo-distributed environment as the Fog computing. 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 and avoid costly over-provisioning of system resources, the deployment of DSP applications must be appropriately reconfigured at run-time, through migration and scaling operations. A *migration* moves an operator replica to another computing resource, so to balance resource utilization or allow to relinquish scarcely used resources. 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 to deal with load spikes, 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.

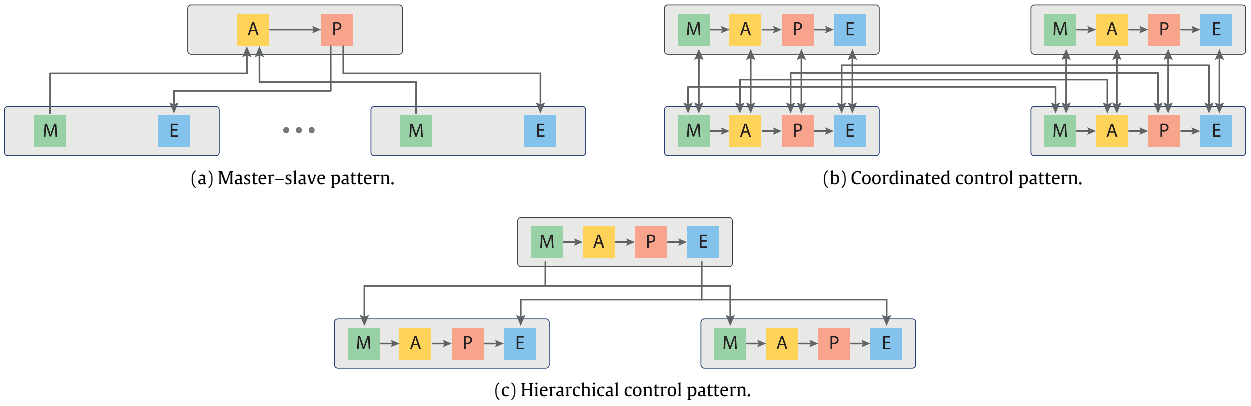
3.2. Architectural options for decentralized control

The MAPE loop represents a prominent and well-know architectural pattern 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 [7]. The *Monitor* component collects data about the controlled entity (e.g., application, operator) and about the execution environment. Then, the *Analyze* component processes the harvested data, so to determine whether adapting the application placement can improve performance (or can reduce the execution costs with acceptable performance penalties). To determine whether a reconfiguration is beneficial, the Analyze component should take into account the adaptation costs; indeed, performing a reconfiguration degrades the application performance in the short term, e.g., by causing a downtime. If the adaptation is needed, the *Plan* component determines which specific reconfiguration action is beneficial, and how it should be performed. Finally, the *Execute* component enacts the adaptation actions, thus updating the application deployment.

When the controlled or managed system is geo-distributed as in Fog computing, a fully centralized MAPE loop introduces a single point of failure and a bottleneck for scalability. Indeed, a centralized managing system may be able to efficiently control the adaptation of only a limited number of entities, and its efficacy may be negatively affected by the presence of network latencies among the managed system components. As described by Weyns et al. in [39], different patterns to design multiple MAPE loops have been used in practice by decentralizing the self-adaptation functions. We here describe some key configurations, aiming to identify the most suitable approach to control DSP applications in the geo-distributed execution environment under investigation.

Master–slave pattern. In a *master–slave pattern*, the system includes a single master component, which runs the Analyze and Plan phases, and multiple independent worker components, which run the Monitor and Execute phases in a decentralized manner. We represent this pattern in Fig. 1a. Differently from a fully centralized approach, this design pattern decentralizes the execution of the Monitor and Execute components, relieving the burden from the centralized control node. The latter is in charge of determining when and how a reconfiguration should be performed. Having single and centralized Analyze and Plan components, this pattern can be equipped with self-adaptation policies that can be more easily designed and, moreover, can more easily determine globally optimal reconfiguration strategies, e.g., [17]. Nevertheless, the centralized components of the MAPE loop can still represent a bottleneck, especially when they have to control a multitude of entities scattered in a large-scale geo-distributed system. Moreover, collecting monitoring data on the master component and dispatching the subsequent scaling actions to the decentralized executors may introduce significant communication overhead.

Coordinated control pattern. Sometimes controlling the elasticity of a system using a single centralized component is unfeasible, e.g., because of scale, administrative, or privacy issues. Anyway, we still need to efficiently control the application elasticity



(a) Master–slave pattern.

(b) Coordinated control pattern.

(c) Hierarchical control pattern.

Fig. 1. Different options for decentralizing the MAPE loop.

so to meet certain QoS metrics. As represented in Fig. 1b, the *coordinated control pattern* employs multiple decentralized MAPE loops, where each control loop oversees one specific part of the system. The control loops may also need to coordinate with one another, as peers, in order to reach joint adaptation decisions. With respect to the degree of cooperation, a great variety of inter-node behaviors can be devised, ranging from a fully uncoordinated to a tightly coordinated one. Each degree of coordination exhibits pros and cons. As observed in [5], in the context of distributed scheduling of DSP applications, the lack of coordination between the distributed agents may introduce too frequent and uncoordinated decisions that can be detrimental for the application performance. Conversely, a tightly coupled coordination reduces the system ability to quickly react to changes. Although this pattern allows to obtain highly scalable solutions, designing efficient control policies is, in general, not easy, because of the difficulty of guaranteeing convergence properties in a decentralized manner.

Hierarchical control pattern. The *hierarchical control pattern* revolves around the idea of a layered architecture, where each layer works at a different level of abstraction. In this pattern, multiple MAPE control loops work with time scales and concerns separation. Lower levels operate on a shorter time scale and are in charge of performing local adaptation. Exploiting a broader view on the system, higher levels steer the overall adaptation by providing guidelines to the lower levels. As represented in Fig. 1c, each layer usually includes a full MAPE loop with all the four components.

We believe that this approach is well suited for controlling DSP applications in a Fog environment: it promises to exploit the benefits of both centralized and decentralized architectures, thus improving performance and scalability without compromising stability. By working at different levels of abstraction, the system can more efficiently deal with a great number of near-edge and Cloud computing resources, which can also expose very different features. Near-edge resources are usually characterized by lower computing capacity, are interconnected by not negligible network latency, and can possibly have limited energy capacity. Conversely, Cloud resources expose (practically) infinite computing capacity and are interconnected with almost negligible network latency. A hierarchical control allows to rule the complexity by decentralizing as much as possible the low-level adaptation, while, at the same time, exploiting the benefit of lightweight higher-level coordination elements, which take advantage of a broader view of the system.

3.3. Hierarchical architecture

To efficiently control the execution of elastic DSP applications in a Fog environment, we propose *Elastic and Distributed DSP Framework* (EDF). It is organized according to the hierarchical pattern

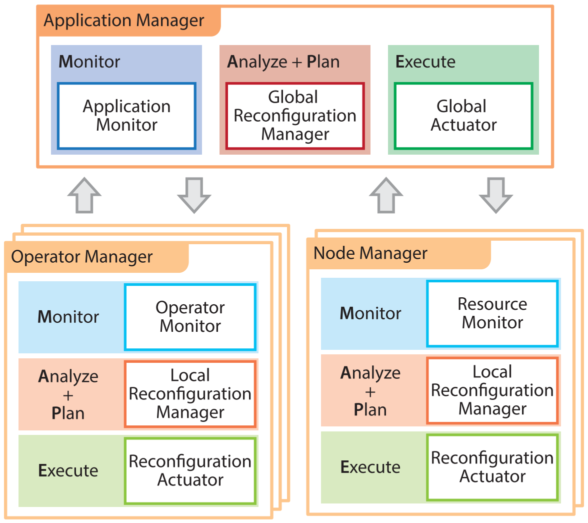


Fig. 2. The EDF conceptual architecture: hierarchical MAPE loops.

for decentralized control, 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. Fig. 2 illustrates the conceptual architecture of EDF, highlighting the hierarchy of the multiple MAPE loops and the system components in charge of the different MAPE phases.

At the lower level and at a faster time scale, EDF executes the Operator Manager and the Node Manager. The *Operator Manager* is a per-operator distributed entity in charge of controlling the adaptation of a single DSP application operator using a local MAPE loop. Through the *Operator Monitor*, it monitors the performance and the resources usage of the operator. Then, through the *Local Reconfiguration Manager*, it analyzes the monitoring information and determines if any local reconfiguration action is needed. The available actions are scale-in and scale-out, which reduce and increase the number of replicas per operator, respectively. When the Operator Manager determines that some adaptation should occur, it issues an operator adaptation request to the higher layer.

The *Node Manager* is a per-node distributed entity that oversees the working conditions of a computing resource using a local MAPE loop. Its goal is to avoid the over-utilization of the computing resource by migrating some of the hosted operator replicas to neighbor resources when needed. Specifically, it uses a *Resource*

Monitor to harvest data regarding the utilization of the node resources by the application operators (e.g., CPU utilization, incoming network traffic). Using the *Local Reconfiguration Manager*, it analyzes the monitored data and determines whether an operator replica should be more conveniently migrated to another computing resource. When the Node Manager determines that a migration should be performed, it issues an adaptation request to the higher layer.

At the higher level and at a slower time scale, EDF executes the *Application Manager*, which 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, using the *Global Reconfiguration Manager*, it analyzes the monitored data and the reconfiguration requests received by the multiple Operator Managers and Node Managers. The Application Manager decides which reconfigurations should be granted. These decisions are then communicated by the *Global Actuator* to each Operator Manager and Node Manager, which can, finally, execute the operator adaptation actions by means of their local *Reconfiguration Actuator*.

The sequence of operations carried out by the Reconfiguration Actuator depends on the reconfiguration protocol in use. In this work, we assume a *pause-and-resume* approach [40] and rely on the stateful migration protocol we presented in [17]. The goal of the protocol is to preserve the integrity of the streams, so that no data is lost, and of the operators internal state (if any), so that the computation semantics is not altered. To this end, in the pause-and-resume approach, the operators involved in the reconfiguration are *paused*, with their internal state being saved to a persistent memory and all the incoming data buffered; after the deployment is changed, the state is restored and the operators execution is *resumed*. Therefore, this process causes application downtime. Moreover, as all the incoming data are buffered during the downtime period, and must be processed as soon as the application is resumed, reconfigurations are expensive in terms of application performance as well.

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 thereof.

4. Hierarchical elasticity policy

The hierarchical architecture presented in Section 3 for self-adaptive DSP elasticity control identifies the different system macro-components (i.e., Application Manager, Node Manager and Operator Manager) 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 QoS requirements, infrastructures with different computing resources, and different user preferences.

The Operator Manager works at the granularity of a single DSP operator and implements what we call a *local policy*. Exploiting its limited local view of the system (e.g., the utilization level and the input data rate of its associated operator), and regardless of the reconfiguration needs determined by the other managers, the local policy can plan a scaling action for the operator. In this paper we explore two classes of Operator Manager policies, the first based on load thresholds (Section 5.1), the latter on Reinforcement Learning (Section 5.2).

The Node Manager works at a more coarse-grained level, controlling the group of operator replicas that are hosted on the same computing node. It exploits a node-level view (i.e., the utilization level of the node and its network distance from nodes in the neighborhood), and relying on a sender-initiated migration policy, described in Section 6, can plan the migration of some of the hosted operators to another destination node.

Both the Operator and Node Managers send their reconfiguration request to the Application Manager, which runs periodically and decides, according to its so called *global policy*, which reconfigurations should be enacted. The global policy works at the granularity of the whole application. By exploiting a global view on the application performance and reconfiguration needs identified by the local managers, it can resolve resource acquisition conflicts and possibly limit the number of reconfigurations, as described in Section 7.

Each reconfiguration request from an Operator or Node Manager specifies the requested *action* and its associated *reconfiguration score*. We consider two types of reconfiguration actions: operator 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 node, respectively. The *reconfiguration score* captures the prospective benefit of the adaptation action according to its proposer. It can express, for instance, the reduction of the operator’s utilization, the reduction of monetary cost for running the operator, or the improvement of some utility function.

5. Operator manager policy

The Operator Manager local policy implements the Analyze and Plan phases of the decentralized MAPE loop, which oversees the execution of a single DSP operator. The Operator Manager relies on the monitoring information collected by the Operator Monitor, which provides several metrics about the operator’s replicas (e.g., CPU utilization, input data rate, processing latency). After analyzing this information, the Operator Manager can possibly plan a reconfiguration of the operator deployment to adjust its parallelism degree.

Whenever a scale-out decision is taken, the Operator Manager has also to select the node that will host the new replica. This task is accomplished in two steps. With the help of the Node Manager, which has a node-level view, a list of the known neighbor nodes with available resources (possibly including the current node) is built, sorted according to their distance in terms of network delay. Then, the Operator Manager selects the new replica location using a randomized approach: the closer the node, the higher the probability of being selected. Similarly, in the case of a scale-in decision, the Operator Manager has to determine which replica will be terminated: again with the help of the Node Manager, it selects the replica hosted on the node with highest utilization.

The proposed reconfiguration request is then communicated to the centralized Application Manager which, based on all the reconfiguration requests it has received and the global policy, determines which requests can be accepted and which not. If the request is finally accepted, the reconfiguration protocol is activated. 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 [1]. So, a scale-out operation redistributes equally the partitions among replicas, whereas a scale-in operation aggregates the partitions from the merged replicas.

We consider two types of scaling policies for the Operator Manager. The first is a simple threshold-based policy whereby scaling decisions are based on the replicas CPU utilization compared to predefined threshold values. We investigate the threshold-based approach since most of the existing auto-scaling solutions rely on this kind of policy. We observe that identifying effective thresholds may require a complete characterization of the operator behavior (i.e., the relationship between the operator utilization and its response time), which can be cumbersome to obtain, as also shown in Section 9. Therefore, the choice of these threshold is usually based on empirical experience. The second approach is based on reinforcement leaning whereby the scaling policy is learned over time by direct interaction with the system. In this case, we exploit a relation between the global application performance and the local operator response time.

5.1. Threshold-based scaling policy

In the threshold-based scaling policy, the Operator Manager monitors the CPU utilization of the operator replicas. Let us denote by U_r the utilization of replica r, which measures the fraction of CPU time used by r. When the replica utilization exceeds the target utilization level $U_{\text{s-out}} \in [0, 1]$, the Operator Manager proposes to add a new replica. Conversely, the Operator Manager proposes a scale-in operation, which removes one of the n running replicas, when the average utilization of the remaining replicas would not exceed a fraction of the target utilization, i.e., when $\sum_{r=1}^{n} U_r/(n-1) < cU_{\text{s-out}}$, $c \in (0, 1)$. This avoids system oscillations with the Operator Manager executing a scale-out operation just after a scale-in.

A possible disadvantage of this policy is that it requires the user to set appropriate values for the threshold $U_{\text{s-out}}$ and for the coefficient c, which drive the scaling decisions. In particular, the task of determining a (near) optimal value for the scale-out threshold may be challenging, especially if the operator performance is not well characterized by its CPU utilization (e.g., it is I/O-bound, or it is subject to a bursty workload).

Reconfiguration score. For the threshold-based scaling policy, the reconfiguration score set by the Operator Manager indicates the severity of the replicas overload/underload when making a reconfiguration request. The score is then used by the Application Manager global policy to rank the different requests. For scale-out requests, we set the reconfiguration score equal to the excess utilization level and normalize it between the lowest value 0 and the highest value 1, that is $score_{s-\text{out}} = (U_r - U_{\text{s-out}})/(1 - U_{\text{s-out}})$. Similarly, in case of scale-in request, we set the reconfiguration score equal to the degree of underloading and normalize it, that is $score_{s-\text{in}} = \left[cU_{\text{s-out}} - \sum_{r=1}^{n} U_r/(n-1)\right]/cU_{\text{s-out}}$.

5.2. Reinforcement learning scaling policy

Reinforcement learning approaches aim to learn the optimal strategy – in our scenario the Operator Manager scaling strategy – through experience and direct interaction with the system [13]. A RL task basically considers an *agent* who aims to minimize a long-term *cost*. Considering a sequence of discrete time steps, which models the periodical activation of the local policy, at each step the agent performs an *action*, looking at the current *state* of its environment (i.e., the operator). The chosen action causes the payment of an immediate cost, and the transition to a new state. Both the paid cost and the next state transition usually depend on external unknown factors as well, hence are stochastic. In order to minimize the expected long-term (discounted) cost, the agent keeps estimates $Q(s, a)$, which represent the expected long-run cost that follows the execution of action a in state s. These estimates constitute the so-called Q-function, and are used by the Operator Manager to take scaling decisions. By observing the actual incurred costs, the Operator Manager updates these estimates over time, and by so doing, also improves its scaling policy.

We define the state of an operator at the beginning of the ith time interval as the pair $s_i = (k_i, \lambda_i)$, where k_i is the number of running replicas, and λ_i the measured average tuple arrival rate at the operator. Even though the tuple arrival rate is a real number, for the sake of analysis we discretize it by assuming that $\lambda_i \in \{0, \bar{\lambda}, \dots, L\bar{\lambda}\}$ where $\bar{\lambda}$ is a suitable quantum (measured in *tuple/s*). We also assume that $k_i \in \{1, \dots, K_{max}\}$. We will denote by $\mathcal{S}$ the set of all the possible operator states.

For each state $s \in \mathcal{S}$, we have a set of scaling decisions represented by a set of actions $\mathcal{A}(s) = \{+1, -1, 0\}$, where $a = +1$ denotes a scale-out decision, $a = -1$ a scale-in decision, and $a = 0$ is the *do nothing* decision. Obviously, not all of the above mentioned actions are available in those states with $k = 1$, where $\mathcal{A}(s) = \{+1, 0\}$ (at least one replica is always running), or with $k = K_{max}$, where $\mathcal{A}(s) = \{-1, 0\}$ (we cannot add replicas beyond the maximum allowed level).

To each triple (s, a, s') we also associate an immediate cost function $c(s, a, s')$, which captures the cost of carrying out action a when the system is in state s and transitions into s'. In our RL model we consider three different costs:

- the reconfiguration cost c_{rcf}. Whenever the system carries out scale-out or a scale-in operation, the operator suffers a downtime period during which no tuple is processed. For the purpose of learning the reconfiguration policy, it suffices to consider a simplified cost model that introduces a constant penalty for scaling actions[1] ;
- the performance penalty c_{perf}, paid whenever the operator response time exceeds a per-operator bound $R_{max,op}$;
- the resource cost c_{res}, that accounts for the cost of the computing resources used to run the operator replicas. For simplicity, we assume that we have a constant cost per replica.

We combine the different costs into a single cost function using the *Simple Additive Weighting* (SAW) technique [41]. According to SAW, we define the cost function $c(s, a, s')$ as the weighted sum of the costs (normalized in the interval [0, 1]):

$$\begin{aligned} c(s,a,s') &= w_{\text{rcf}} \frac{\mathbb{1}_{\{a\neq 0\}} c_{rcf}}{c_{rcf}} + w_{\text{perf}} \frac{\mathbb{1}_{\{R(k+a,\lambda')>R_{max,op}\}} c_{perf}}{c_{perf}} + \\ &\quad + w_{\text{res}} \frac{(k+a)c_{res}}{K_{max} c_{res}} \\ &= w_{\text{rcf}} \mathbb{1}_{\{a\neq 0\}} + w_{\text{perf}} \mathbb{1}_{\{R(k+a,\lambda')>R_{max,op}\}} + w_{\text{res}} \frac{k+a}{K_{max}} \end{aligned} \qquad (1)$$

where $\mathbb{1}_{\{\cdot\}}$ is the indicator function, w_{rcf}, w_{perf} and w_{res}, $w_{\text{rcf}}+w_{\text{perf}}+w_{\text{res}} = 1$, are non negative weights for the different costs, and $R(k, \lambda)$ the average response time when the operator has k replicas and an input tuple rate of λ tuple/s.

Intuitively, the cost function allows us to *instruct* the Operator Manager to discriminate between the *good* system configurations and actions and the *bad* configurations and actions (the larger the cost, the worse the configuration). As the Operator Manager aims at minimizing the incurred cost, it is encouraged by the cost function to (i) reduce the number of requested reconfigurations, (ii) keep the response time within the given bound, and (iii) limit the resource usage. The different weights allows us to express the relative importance of each cost term. Differently from the threshold-based solution, this policy directly optimizes the response time, without relying on system-dependent metrics, such as the CPU utilization. Indeed, as the application response time results from a combination of the single operators response time,

[1] We observe that a detailed model of the reconfiguration cost, that results by performing specific actions in specific system states (e.g., to migrate the operator with state size α from node u to node v, where u and v are interconnected with network delay β), would lead to a significantly larger state–action space, whose analysis could be too computationally demanding.

with an adequate choice of $R_{max,op}$ for each operator, the local cost function guides the agent towards meeting the global performance target in a fully decentralized way.

As regards the definition of the bounds $R_{max,op}$, we observe that they grant a share of the global bound R_{max} to each operator accordingly to their computational weight. They could be set either statically after preliminary profiling, or dynamically estimated and adapted at run-time by the Application Manager. In Section 9, we describe the simple criteria we followed for setting the bounds for our reference application during the experimental session.

Algorithm 1 illustrates the general RL scheme: the Q functions are first initialized (setting all to 0 will often suffices) (line 1); then, by direct interaction with the system, the Operator Manager at each step t chooses an action a_t (based on current estimates of Q) (line 3), observes the incurred cost c_t and the next state s_{t+1} (line 4), and then updates the Q function based on what it just experienced during step t, that is the tuple (s_i, a_i, c_i, s_{i+1}) (line 5). The different solutions differ for the actual learning algorithm adopted and on the assumptions about the system.

In this paper we consider two RL algorithms at the extremes of the spectrum. For its simplicity, we first consider the well-known *Q-learning* algorithm. Q-learning is a *model-free* learning algorithm which requires no knowledge of the system dynamics. Then, we present a *model-based* approach, which basically improves its estimates of the entire system dynamic over time and accordingly updates the Q function.

Algorithm 1 RL-based Operator Elastic Control Algorithm

```
1: Initialize the $Q$ functions
2: loop
3:     choose a scaling action $a_i$ (based on current estimates of $Q$)
4:     observe the next state $s_{i+1}$ and the incurred cost $c_i$
5:     update the $Q(s_i, a_i)$ functions based on the experience
6: end loop
```

5.2.1. Q-learning

Q-learning is an off-policy learning method that estimates the optimal action value function Q^* by its sample averages [13]. At any decision step (line 3 of Algorithm 1), Q-learning either: (1) *exploits* its knowledge about the system, that is, the current estimates Q, by *greedily* selecting the action that minimizes the *estimated* future costs, i.e., $a_i = \arg\min_{a' \in A(s_i)} Q(s_i, a)$; or (2) *explores* by selecting a random action to improve its knowledge of the system. Here we consider the simple ϵ*-greedy* action selection method, which chooses either a random action with probability ϵ or the greedy action with probability $1 - \epsilon$.

The algorithm performs simple one-step updates at the end of each time slot (line 5), as follows:

$$Q(s_i, a_i) \leftarrow (1-\alpha) Q(s_i, a_i) + \alpha \left[c_i + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} Q(s_{i+1}, a') \right] \quad (2)$$

where $\alpha \in [0, 1]$ is the *learning rate* parameter and $\gamma \in [0, 1)$ is the discount factor. Observe that (2) simply updates the old estimate for $Q(s, a)$ with the just observed cost c_i plus the discounted expected cost of following the greedy policy onward, that is $\min_{a' \in \mathcal{A}} Q(s_{i+1}, a')$.

5.2.2. Model-based reinforcement learning

At the other extreme of the RL strategies, we now consider the *full backup model-based* RL approach (see [13]). In the full backup approach, we rely on a possibly approximated system model, and directly use the Bellman equation [13,42] to compute the Q-functions:

$$Q(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a) \left[c(s, a, s') + \gamma \min_{a' \in \mathcal{A}} Q(s', a') \right] \quad \begin{array}{l} \forall s \in \mathcal{S}, \\ \forall a \in \mathcal{A}(s) \end{array}$$

We replace the unknown transition probabilities $p(s'|s, a) = P[s_{i+1} = s'|s_i = s, a_i = a]$, and the unknown cost function $c(s, a, s')$, $\forall s, s' \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ by their empirical estimates. In order to estimate the transition probabilities $p(s'|s, a)$, we observe that:

$$p(s'|s, a) = P[s_{i+1} = (k', \lambda')|s_i = (k, \lambda), a_i = a] = \\ = \begin{cases} P[\lambda_{i+1} = \lambda'|\lambda_i = \lambda] & k' = k + a \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

It follows that to estimate the state transition probabilities, it suffices to estimate the tuple arrival rate transition probabilities $P[\lambda_{i+1} = \lambda'|\lambda_i = \lambda]$. Hereafter, since λ takes value in a discrete set, we will write $P_{j,j'} = P[\lambda_{i+1} = j'\bar{\lambda}|\lambda_i = j\bar{\lambda}]$, $j, j' \in \{0, \ldots, L\}$ for short. Let $n_{i,jj'}$ the number of times the arrival rate changes from state $j\bar{\lambda}$ to $j'\bar{\lambda}$, in the interval $\{1, \ldots, i\}$, $j, j' \in \{1, \ldots, L\}$. At time i the transition probabilities estimates are then

$$\widehat{P_{j,j'}} = \frac{n_{i,jj'}}{\sum_{l=0}^{L} n_{i,jl}}$$

from which we derive the estimates $\hat{p}(s'|s, a)$ via (3).

In order to estimate the immediate cost $c(s, a, s')$, we first split it into two terms, respectively named the *known* and the *unknown* costs:

$$c(s, a, s') = c_k(s, a) + c_u(s') \quad (4)$$

Comparing the expression above with (1), we observe that the known cost $c_k(s, a)$ accounts for the reconfiguration and resources costs, and only depends on the current state and action. On the other hand, the unknown cost $c_u(s')$ accounts for the performance penalty, which depends on the replication level and input tuple rate in the next time interval, i.e., on the next state s'. As we assume that neither the input rate transition model nor the operator response time model are known, we have to estimate $c_u(s')$ online. To this end, the agent observes the incurred cost c_i at the end of each time interval i. Given this information collected from experience, it can determine $c_{u,i}$, the "unknown" cost paid in the ith time slot, as:

$$c_{u,i} = c_i - c_k(s, a)$$

by simply applying (4). Then, the unknown cost estimate is updated using a simple exponential weighted average:

$$\hat{c}_u(s') \leftarrow (1-\alpha)\hat{c}_u(s') + \alpha c_{u,i} \quad (5)$$

We must note that the cost estimation rule above does not exploit all the *a priori* knowledge about the system. Indeed, we can heuristically assume that the expected cost due to response time violation is not lower when the parallelism degree is reduced and/or the input rate increases, and it is not higher with more parallel instances and/or a lower data rate. Therefore, after applying (5) to $s = (k, \lambda)$, we always enforce the following constraints adjusting the estimates for states $s' = (k', \lambda')$:

$$\begin{array}{ll} \hat{c}_{u,i}(s) \le \hat{c}_{u,i}(s') & \forall k \ge k', \lambda \le \lambda' \\ \hat{c}_{u,i}(s) \ge \hat{c}_{u,i}(s') & \forall k \le k', \lambda \ge \lambda' \end{array}$$

The resulting Q function update step (line 5 of Algorithm 1) is summarized in Algorithm 2. Given the current estimates $Q(s, a)$, at any step the Operator Manager just chooses the greedy action, that is the action with the minimum long term estimated cost, i.e., $\arg\min_{a \in A(s_i)} Q(s_i, a)$. Therefore, differently from the model-free scenario, we do not need a mechanism for forcing exploration any more.

Algorithm 2 Full Backup Model-Based Learning Update

```
1: Update estimates $\widehat{P_{j,j'}}$ and $\hat{c}_{u,i}(s_i)$
2: for all $s \in \mathcal{S}$ do
3:     for all $a \in \mathcal{A}(s)$ do
4:         $Q(s,a) \leftarrow \sum_{s' \in \mathcal{S}} \hat{p}(s'|s,a) \left[ \hat{c}(s,a,s') + \gamma \min_{a' \in \mathcal{A}} Q(s',a') \right]$
5:     end for
6: end for
```

Reconfiguration score. For the RL scaling policy, we base the reconfiguration score on the expected gain in taking the proposed action. Since the *estimated* cost associated to not taking any action is $Q(s_i, 0)$, the normalized reconfiguration score for a scaling action is then

$$score(a) = \frac{Q(s_i, 0) - Q(s_i, a)}{\max_{\forall h=\{1,\dots,i\}; \forall a \in \mathcal{A}(s_h)} \{1, Q(s_h, 0) - Q(s_h, a)\}}.$$

It is worth observing that differently from the threshold-based policy, where the score takes into account the current utilization level, the reconfiguration score for the RL policy accounts for the expected long-term impact of taking the proposed action.

6. Node manager policy

The Node Manager policy implements the Analyze and Plan phases of the local MAPE loop (see Fig. 2). Its goal is avoiding to run the application components on an overloaded node. A computing node can host replicas of one or more operators. When the computing node is overloaded, the hosted replicas can experience a performance degradation. To overcome this issue, the Node Manager proposes to move some of the operator replicas away from the node using a sender-initiated policy.

The Node Manager exploits a partial knowledge of the system provided by the Resource Monitor: the utilization level of the node, and the network delays to a restricted suitable set of computing nodes (i.e., located in its neighborhood). Using such information, the Node Manager can plan a migration of some of the operator replicas deployed on its node to another computing node. As for the scaling actions proposed by the Operator Manager, such a migration request has to be communicated to the centralized Application Manager which, based on the other reconfiguration requests and the global policy, determines if the migration can be granted, and possibly activates the migration protocol.

We adopt a sender-initiated, reactive and threshold-based policy in order to decide when and how to perform the migration. We denote with U_h the overall CPU utilization of the node h. When U_h exceeds the target utilization value $U_{mig} \in [0, 1]$, the policy plans to migrate one operator replica, randomly selected, to a new location. The latter is identified in two steps using the same randomized policy we already presented in Section 5 to determine the target node in case of operator scale-out. The only difference is that the policy does not consider the overloaded resources, included the current one, in the list of possible target nodes.

Reconfiguration score. For the migration policy, the reconfiguration score set by the Node Manager indicates the severity of the node overload when making a reconfiguration request. Specifically, we set the migration score equal to the excess utilization level and normalize it between the lowest value 0 and the highest value 1, that is, $score_{mig} = (U_h - U_{mig})/(1 - U_{mig})$. Such a score is then used by the Application Manager global policy to rank the different requests.

7. Application manager policy

The Application Manager global policy implements the Analyze and Plan steps of the centralized MAPE loop. Its main goal is to achieve satisfying application performance, by coordinating the adaptation actions proposed by the decentralized managers (i.e., Operator Manager, Node Manager). Ideally, the global policy should encourage the local policies to perform reconfigurations, when the application is not performing well, and should promote a settling period, when the application is not subject to changing working conditions. To avoid over complicating the hierarchical policy design, we resort on a simple global policy that coordinates reconfigurations, prevents the enactment of conflicting reconfigurations (e.g., two operators requesting a new replica on the same computing resource), and eventually limit the overall number of granted reconfigurations. It proceeds in three steps: (1) reconfiguration request prioritization; (2) conflict prevention; (3) acceptance of requests.

In the *reconfiguration request prioritization* step, the global policy determines which are the most worthy reconfiguration requests that should be applied. To solve this task, it resorts on the reconfiguration score included by the decentralized managers in the reconfiguration request. For each type of reconfiguration action (i.e., migration, scale-in, scale-out), it sorts the requests by decreasing reconfiguration score, so that requests with higher score receive higher priority.

In the *conflict prevention* step, the global policy goes through the received reconfiguration requests sorted by priority, detect the conflicting requests and remove those having the lower score. We consider two reconfigurations as conflicting if they want to use the same computing resource to allocate a new replica or to migrate an existing one.

In the last step, *acceptance of requests*, the global policy decides which reconfiguration requests should be granted. Note that, at this point, these requests are not in conflict one another. The choice ranges from simply accepting all the proposed requests to limit somehow their number.

Token bucket based granting policy. In our previous work [6], we designed a granting policy based on a token bucket. Here, we propose a revised version, which overcomes few limitations of the previous solution. The token bucket aims to determine the number of reconfiguration requests, proposed by the decentralized Operator Managers and Node Managers, that should be enacted as to improve application performance or reduce deployment costs, while controlling the number of application reconfigurations (which cause application downtime).

We resort on a simple token bucket policy, where two types of tokens are available: the H-token, which grants either a scale-out operation or a migration, and the L-token, which grants a scale-in operation. These tokens are accumulated in a token bucket that has a finite capacity C_τ: when the bucket is full, new tokens pop out oldest ones from the bucket. Furthermore, we assume that the token bucket can host a single type of tokens (i.e., either H- or L-tokens) at any given time: e.g., the insertion of a H-token pops out all the L-tokens stored in the bucket. Since the number of granted reconfigurations is thus limited by the number of available tokens, a key role is played by the token generation rate. Ideally, when the application response time is within acceptable bounds, 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 degrade or be good enough to signal potential resource under-utilization, the system should be more prone to reconfigure itself. Periodically, every T_τ, if the application response time is above a high threshold τ_H, a new token H is generated and stored in the bucket; alternatively, if the response time is below a low threshold τ_L, a new token L is generated and stored. If the application response time is in between the two thresholds, no token is generated.

8. Storm integration

To manage DSP applications relying on the hierarchical control approach, we have integrated the designed EDF architecture in Apache Storm. In this section, we first briefly provide an overview of Storm; then, we present the prototype design.

8.1. Apache Storm

Storm is an open source, real-time, and scalable DSP system maintained by the Apache Software Foundation. It manages the execution of DSP applications over a set of worker nodes interconnected in an overlay network. A *worker node* is a generic computing resource (i.e., physical or virtual machine).

In Storm, we can distinguish between an abstract application model and an execution application model. In the abstract model, a DSP application is represented by its *topology*, which is a DAG with spouts and bolts as vertices and streams as edges. A *spout* is a data source that feeds data into the system through one or more streams. A *bolt* is either a processing element, which generates new outgoing streams, or a final information consumer. A *stream* is an unbounded sequence of *tuples*, which are key–value pairs. We refer to spouts and bolts as operators.

In the execution model, Storm transforms the topology by replacing each operator with its tasks. A *task* is an instance of an application operator (i.e., spout or bolt), and it is in charge of a share of the incoming operator stream. For the execution, one or more tasks of the *same* operator are grouped into executors, implemented as threads. An *executor* is the smallest schedulable unit; Storm can process large data volumes in parallel by launching multiple executors for each operator. The framework also introduces the *worker process*, that is basically a Java process acting as a container for a subset of the executors of the *same* topology. The maximum parallelism degree for an operator is achieved when each task is assigned its own executor; in other words, the number of executors of an operator must always be less than or equal to its number of tasks.

Besides the computing resources (i.e., the worker nodes), the architecture of Storm includes two additional components: Nimbus and ZooKeeper. *Nimbus* is a centralized component in charge of coordinating the topology execution; it uses its *scheduler* to define the placement of the application operators on the pool of available worker nodes. The assignment plan determined by the scheduler is communicated to the worker nodes through *ZooKeeper*,[2] that is a shared in-memory service for managing configuration information and enabling distributed coordination. Since each worker node can execute one or more worker processes, a *Supervisor* component, running on each node, starts or terminates worker processes according to the Nimbus assignments. A worker node can concurrently run a limited number of worker processes, based on the number of available *worker slots*.

8.2. Distributed Storm

To elastically adapt the applications deployment in Storm, we resort on Distributed Storm, which is our extension of Storm that was initially developed to enable the QoS awareness of the scheduler [5] and then further extended with mechanisms to support run-time stateful operator scaling and migration [17]. Furthermore, Distributed Storm introduces an infrastructure-level and application-level monitoring system. The monitoring system provides intra-node and inter-node information, including network latencies among nodes, CPU utilization (per node and per executor), and exchanged data rate among executors. Network latencies are estimated using a network coordinate system, which is built using Vivaldi [43], a decentralized algorithm having linear complexity with respect to the number of network locations. As regards the support for run-time adaptation, Distributed Storm provides mechanisms for pursuing elasticity while preserving the application integrity. To this end, our extension handles stateful operators by conveniently relocating and distributing their internal state, using the stateful migration protocol outlined in Section 3.3.

2 http://zookeeper.apache.org/.

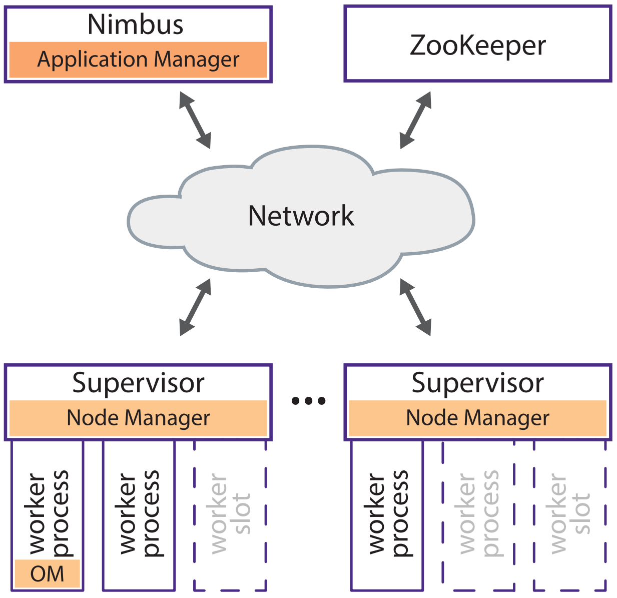


Fig. 3. Storm architecture with the new EDF components: Application Manager, Operator Manager (for short, OM), and Node Manager.

8.3. Integration of EDF in Storm

The implementation of EDF in Distributed Storm is straightforward. As represented in Fig. 3, we introduce the EDF components, described in Section 3, into the existing Storm architecture. More precisely, the newly introduced components are the Application Manager, the Operator Managers, and the Node Managers. These components use the shared in-memory service provided by ZooKeeper in order to exchange information regarding the proposed, accepted, and denied reconfiguration requests.

The *Application Manager* is created by Nimbus when a new application is submitted to Storm, and it runs for the whole application lifetime. As soon as the Application Manager is created, it determines the initial application placement on the set of worker nodes. Being interested in investigating the run-time adaptation policies, we determine the initial placement leveraging on the decentralized placement heuristic proposed in [2] and implemented in Distributed Storm [5]. The placement policy assigns the operators so to minimize the application response time, taking the network latency into consideration.

At run-time, Nimbus periodically executes the Application Manager every T_{AM}. This manager first analyzes the monitored application response time, acquired from Distributed Storm, and collects the reconfiguration requests coming from the decentralized managers. Then, the global policy is executed so to coordinate and grant the reconfiguration actions (see Section 7). After this planning phase, the Application Manager runs the Global Actuator to enact the deployment changes. To this end, the latter relies on the `rebalance` command of Storm, which accordingly updates the topology executors, and on the stateful migration mechanisms of

Distributed Storm, which allow to preserve the operators internal state.

When a new application is submitted to Storm, Nimbus also creates multiple *Operator Managers* (one per operator). They are assigned to the available worker nodes by the Storm scheduler, and run for the whole application lifetime. At run-time, each Operator Manager collects the amount of resources used by the managed operator, relying on the monitoring system provided by Distributed Storm. In particular, we measure the utilization for each replica (i.e., executor) retrieving the CPU usage information for the corresponding thread, the rate at which operators exchange tuples, and the average operator response time. The Operator Manager's local policy is periodically executed (every T_{OM}), so to identify beneficial reconfigurations and to propose them to the global Application Manager (see Section 5). Should a reconfiguration be performed, the Reconfiguration Actuator of the Operator Manager adapts the operator deployment (e.g., by changing its replication degree), while safely preserving the operator internal state. To this end, this component relies on the internal mechanisms of Storm and on the stateful migration mechanisms of Distributed Storm. After a reconfiguration is performed, the locale MAPE loop is suspended for a short period, in order to prevent the monitoring component to propagate measures heavily impacted by the reconfiguration process itself.

The *Node Manager* is implemented within the Supervisor and runs periodically (every T_{NM}) as long as the worker node is alive. It retrieves monitoring information about CPU utilization, exchanged inter-node traffic, and network latencies, from the monitoring components of Distributed Storm. Afterwards, it runs the local policy to determine whether migrating an operator replica can relieve the worker node overload, and possibly forwards the request to the Application Manager. Should a reconfiguration be performed, the Reconfiguration Actuator of the Node Manager adapts the operator deployment by exploiting mechanisms provided by Storm and, for the stateful migration, by Distributed Storm.

We observe that these components are not tied to a specific local or global policy, but support the execution of custom policies. In our experiments, we will use and evaluate different combinations of global and local policies.

9. Experimental results

We evaluate EDF and the proposed policies using our extended version of Apache Storm 1.1.0. We deploy EDF on a cluster made of 4 worker nodes and one further node to host Nimbus and ZooKeeper. Each worker node is configured to host up to 8 operator replicas (i.e., executors), and is equipped with a dual CPU Intel Xeon E5504 (8 cores at 2 GHz) and 16 GB of RAM.

For the experiments, we use the reference application that solves a query of the DEBS 2015 Grand Challenge [16], 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. Fig. 4 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 min, supported by *metronome* that defines the passing of time. Finally, *partialRank* and *globalRank* compute the top-10 most frequent routes.

We feed the application with a sample dataset provided by DEBS for the challenge, containing real data collected over one year. The taxi service utilization significantly changes during the day, thus the application input rate is variable as well. In order to obtain a more resource demanding workload, we accelerate and amplify the original dataset. In particular, we reduce the events

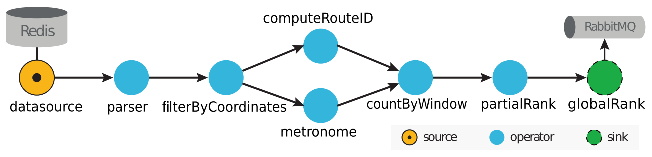


Fig. 4. Reference DSP application.

inter-arrival time by a factor of 10, so that in a 12-h experiment we process data collected during 5 days. Then, we amplify the DEBS dataset with new surrogate events, while preserving the original statistical properties in terms of taxi routes distribution. The resulting dataset is 5 times larger than the original one, with a tuple emission rate that ranges from about 20 to 500 tuples per second. The faster workload variations make the run-time adaptation even more challenging.

We run the experiments using all the proposed Operator Manager policies, namely the threshold-based policy and the two RL-based ones. As regards the threshold-based policy, we set the under-utilization parameter $c = 0.75$ and try different values for the threshold $U_{\text{s-out}}$. For the RL model-based local policy, we use two configurations of the cost weights: (i) $w_{\text{perf}} = w_{\text{res}} = 0.4$, $w_{\text{rcf}} = 0.2$, and (ii) $w_{\text{perf}} = w_{\text{rcf}} = 0.4$, $w_{\text{res}} = 0.2$. During preliminary simulations, these configurations turned out to well represent different trade-offs for QoS metrics optimization. For input rate discretization, we use $\bar{\lambda} = 50$ tuples/s.

We consider $R_{max} = 250$ ms as the target application response time. To set the target response time $R_{max,op}$ for each operator, we performed some preliminary experiments. First, we identified a configuration of the operators parallelism in which none of the operators was overloaded (i.e., their monitored utilization was below 70%). To this end, it was enough to run just one replica for all the operators except for *partialRank*, whose parallelism was set to 2. Then, keeping the parallelism fixed, we monitored the response time of the single operators and the whole application. Using this information, we observed how much time was spent at each operator when the application response time was close to R_{max}, and consequently set $R_{max,op}$ to those values. In particular, we set 5 ms for *parser*, *filterByCoordinates*, *computeRouteID*, and *countByWindow*, and 210 ms for *partialRank*, which represents a bottleneck.[3] As regards the maximum parallelism degree, we set $K_{max} = 3$ for each operator, except for *metronome* and *globalRank*, which cannot be replicated (i.e., $K_{max} = 1$), and for *partialRank*, which is a potential bottleneck and can run up to 6 replicas.

The planning phase of the Operator Managers and Application Manager is executed twice per minute (i.e., $T_{AM} = T_{OM} = 30s$). The global policy uses a token bucket that stores at most $C_\tau = 1$ token at any time. We also set $T_\tau = 60$ s, $\tau_L = 125$ ms (i.e., $\frac{1}{2}R_{max}$) and $\tau_H = 225$ ms (i.e., $\frac{9}{10}R_{max}$). To evaluate the impact of the token bucket, in the first experiments we use a baseline global policy that grants all the requested reconfigurations (except for the conflicting ones). The RL algorithms use the following parameters: discount factor $\gamma = 0.99$, learning rate $\alpha = 0.1$ and, for Q-learning, $\epsilon = 0.05$.

In the experiments, we focus on the scaling policies. To this end, we disable the migration policy in the Node Managers. Anyway, we verify that, with the above settings, none of the worker nodes is overloaded during the experiments.

In the rest of this section, we present the results of our evaluation with different combinations of local and global policies. As

[3] Observe that *metronome* and *globalRank* cannot be replicated; therefore, although they may influence the definition of $R_{max,op}$ for the other operators, we cannot enforce response time bounds for these components.

Table 1
Results of the experiments with different local policies (TH: threshold-based, QL: Q-learning, MB: Model-based RL), with and without the token bucket mechanism at the global policy level. P_{50} and P_{95} denote the median and 95th percentile of the application response time.

Local policy		Token bucket	Response time (ms)			Avg. replicas	Downtime (%)	R_{max} violation (%)
			Mean	P_{50}	P_{95}			
TH	$U_{\text{s-out}} = 0.7$		363.0	320.6	686.8	8.69	1.25	82.17
TH	$U_{\text{s-out}} = 0.6$		324.6	300.0	507.7	8.75	0.97	77.76
TH	$U_{\text{s-out}} = 0.5$		295.0	279.7	457.9	8.87	1.48	69.25
TH	$U_{\text{s-out}} = 0.4$		230.0	229.9	334.0	9.61	3.38	35.68
TH	$U_{\text{s-out}} = 0.4$	✓	211.4	209.8	302.3	9.50	1.23	25.19
TH	$U_{\text{s-out}} = 0.3$		161.6	152.3	274.6	12.40	6.12	7.48
TH	$U_{\text{s-out}} = 0.3$	✓	164.1	158.3	239.4	10.51	3.18	3.62
QL	$w_{\text{perf}} = w_{\text{res}} = 2w_{\text{rcf}}$		474.9	145.9	691.7	13.78	10.96	17.99
QL	$w_{\text{perf}} = w_{\text{res}} = 2w_{\text{rcf}}$	✓	144.0	137.8	216.9	11.46	1.56	1.31
QL	$w_{\text{perf}} = w_{\text{rcf}} = 2w_{\text{res}}$		492.5	138.5	789.8	13.89	10.84	14.97
QL	$w_{\text{perf}} = w_{\text{rcf}} = 2w_{\text{res}}$	✓	138.1	133.4	205.0	12.21	1.16	0.81
MB	$w_{\text{perf}} = w_{\text{res}} = 2w_{\text{rcf}}$		176.4	154.7	271.9	10.13	3.20	6.92
MB	$w_{\text{perf}} = w_{\text{res}} = 2w_{\text{rcf}}$	✓	168.4	161.9	240.4	9.87	1.40	3.06
MB	$w_{\text{perf}} = w_{\text{rcf}} = 2w_{\text{res}}$		94.0	89.3	138.9	12.00	0.00	0.01
MB	$w_{\text{perf}} = w_{\text{rcf}} = 2w_{\text{res}}$	✓	93.8	89.3	137.8	12.00	0.00	0.00

regards the local policy, we consider all the solutions described in Section 5, namely threshold-based, Q-learning, and model-based RL. We combine them with two different settings for the global policy, that is with and without the token bucket mechanism (which limits the number of reconfigurations by exploiting the global view of the application). When the token bucket is not used, the global policy only solves conflicting reconfiguration requests.

Table 1 summarizes the results of our experiments. For selected configurations, we also plot the dynamics of the relevant statistics over time in Fig. 5. These graphs highlight the negative effects of the reconfigurations in Storm, with evident peaks in the application response time and source data rate caused by the tuple buffering mechanisms used by the reconfiguration protocol to preserve the application integrity. Therefore, to neglect transient behaviors due to the initial system setup and the run-time reconfigurations, we exclude, for each metric monitored during the experiments, the initial 30 min and 2 min after each reconfiguration.

Threshold-based local policy. We first analyze the behavior of the system when the Operator Manager adopts the threshold-based local policy (TH in Table 1). To determine a good choice for the scale-out threshold $U_{\text{s-out}}$, we evaluated the policy using different threshold values. When we set $U_{\text{s-out}} = 0.7$, EDF uses only 8.7 replicas on average,[4] with the mean response time being 363 ms, and violating $R_{\max}$ for 82% of the time. We achieve similar results setting $U_{\text{s-out}}$ equal to 0.6 and 0.5. In order to achieve acceptable response times for the considered application, we need a CPU threshold not higher than 0.4[5] Therefore, for this application, the average utilization does not well describe the operator performance. Using the very low threshold $U_{\text{s-out}} = 0.3$, EDF runs more than 12 replicas on average, still violating the target response time 7% of the time. As shown in Fig. 5a, the number of replicas is frequently adjusted, keeping the application down for about 6% of the time.

In order to verify whether we could reduce the number of reconfigurations performed, we combine the threshold-based local policy with the global policy that relies on the token bucket mechanism. We focused on the configuration with $U_{\text{s-out}} \in \{0.3, 0.4\}$, since they are the only ones that achieve acceptable response time. Fig. 5b shows how the token bucket clearly limits the number of enacted reconfigurations when the scaling threshold is $U_{\text{s-out}} =$ 0.3. In this setting, the application performance is similar to the base case without the token bucket; nevertheless, EDF runs about 2 less replicas on average, and the application experiences half the downtime throughout the experiment. Although a scale-out threshold equal to 0.3 (or 0.4) is far from the load thresholds usually adopted, we identified these values by means of successive empirical experiments. Such low thresholds depend on the specific application logic, thus showing that, in some cases, determining the best scaling thresholds can be cumbersome.

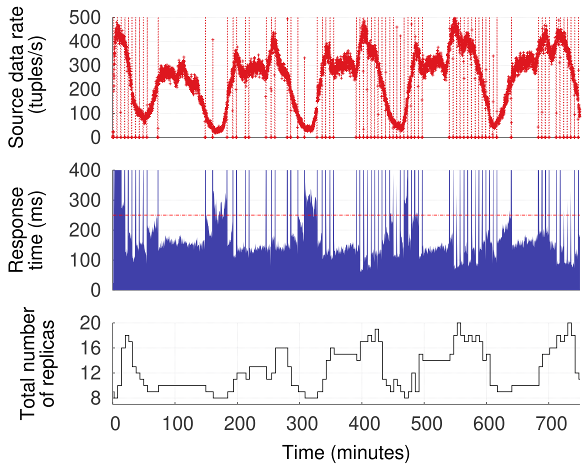
Q-learning local policy. RL based policies are very appealing, because they promise to obtain desirable behavior with limited knowledge about the system dynamics and, in particular, about how to optimize the deployment objectives. We first evaluate our simple solution based on Q-learning. When the Operator Manager uses this policy (QL in Table 1), the overall performance is poor, with the application being continuously reconfigured (see Fig. 5c). In fact, Q-learning is well known for its simplicity, provided at the cost of extremely slow convergence to the optimal strategy. Indeed, even varying the cost weights we cannot see significant changes in the policy behavior throughout our 12-h experiment, at the end of which Q-learning is still far from converging to good results.

Interestingly, when coupled with the token bucket, the overall results for the Q-learning policy dramatically improve. Although the Application Manager has not the power to turn sub-optimal decisions into smarter ones, it can at least avoid unnecessary reconfigurations. As shown in Fig. 5d, in this setting the token bucket mechanism avoids scaling actions whenever the application performance is acceptable, limiting both the $R_{\max}$ violations and the downtime length. By conveniently selecting the reconfigurations to be applied, the token bucket allows to meet the application requirements most of the time. However, to obtain this goal, we observe that the Application Manager only accepts less than 1% of the proposed reconfigurations, and runs more than 11 replicas on average. This behavior confirms that, in our case, Q-learning would require much more time to learn a suitable policy. During its learning process, it needs to gain knowledge on the system behavior at the price of performing exploratory counter-intuitive adaptation actions (e.g., scale-in when the operator exceeds its response time bound).

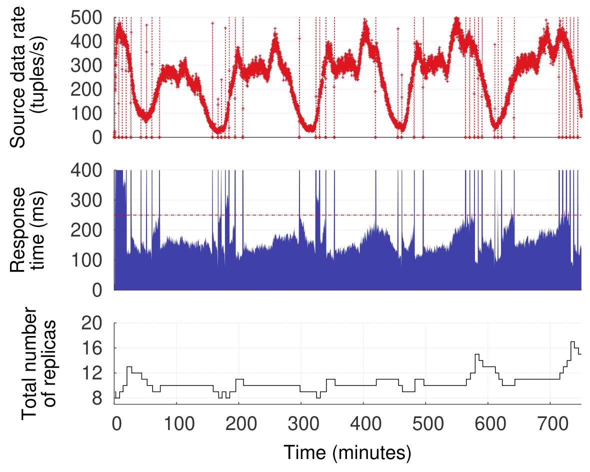
Model-based RL local policy. We now evaluate the model-based RL local policy (MB in Table 1), which exploits the (partially) available knowledge in order to speedup the learning process. We first consider a cost function configuration with $w_{\text{perf}} = w_{\text{res}} = 0.4$ and $w_{\text{rcf}} = 0.2$, without the token bucket at the global policy level. This fully decentralized solution manages to keep the application response time within $R_{\max}$, except for less than 7% of the time.

[4] For the reference application, the minimum number of running replicas is 8 (one per component).

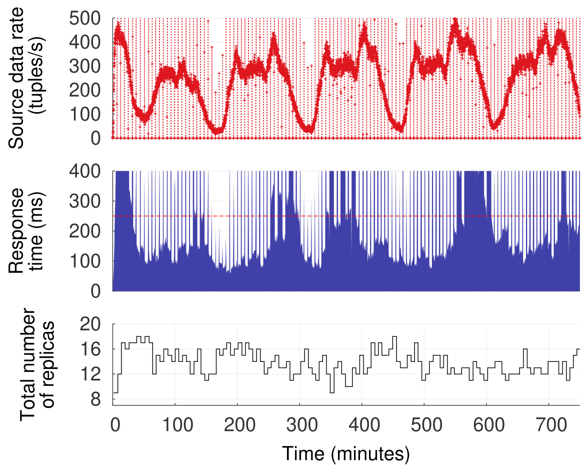
[5] Since the taxi route statistics are periodically refreshed, the ranking operators of the application are subject to a bursty workload. This periodic and frequent load bursts are not well captured by the average CPU utilization, which is computed by the Operator Manager every T_{OM} seconds.



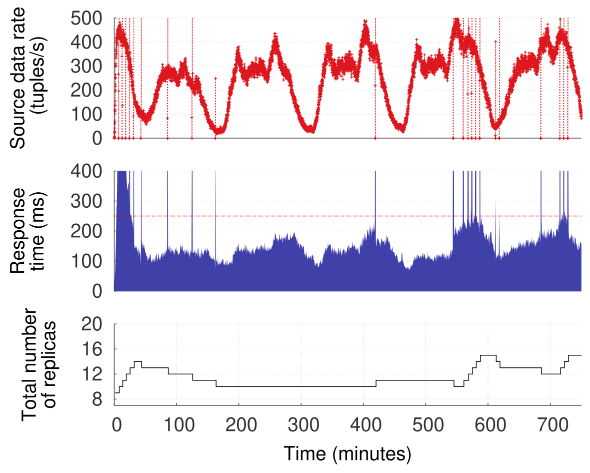
(a) Threshold-based, $U_{\text{s-out}} = 0.3$.



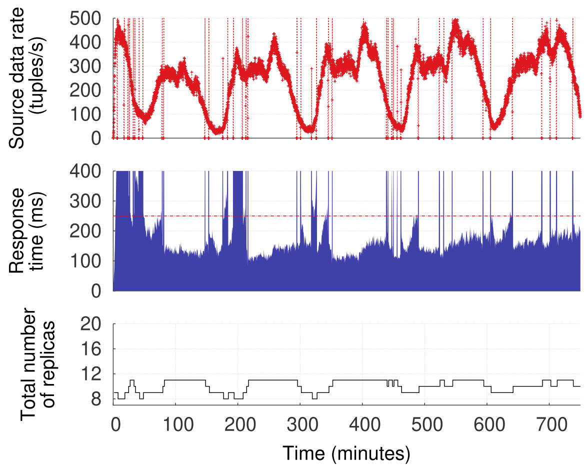
(b) Threshold-based, $U_{\text{s-out}} = 0.3$, with token bucket.



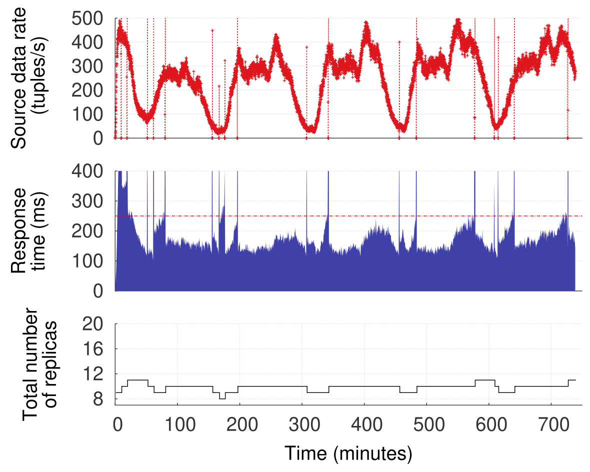
(c) Q-learning.



(d) Q-learning, with token bucket.



(e) Model-based RL.



(f) Model-based RL, with token bucket.

Fig. 5. Response time and number of replicas using different policies for both the OperatorManager and the ApplicationManager.

It runs about 10 replicas on average, and the overall application downtime is significantly reduced with respect to the previously described approaches that use no token bucket. From Fig. 5e, we can also observe that the model-based RL algorithm is very sensitive to fluctuations of the measured incoming data rate; in the current experiment, they generate occasional oscillations in the number of running replicas (e.g., a scale-out immediately followed by a scale-in for the same operator).

To further improve the behavior of this fully decentralized solution, we combine the model-based RL local policy with the token bucket. We plot the resulting behavior in Fig. 5f, where we note that the token bucket solves the occasional oscillations issue of the fully decentralized approach (shown in Fig. 5e). In this setting, the 95th percentile of the application response time is within R_{max}, running not more than 9.9 replicas on average. The downtime is limited to 1.4%.

We now show that the model-based RL policy can exploit different trade-offs between the considered QoS metrics. We set $w_{\text{perf}} = w_{\text{rcf}} = 0.4$ and $w_{\text{res}} = 0.2$, thus privileging the performance and reconfiguration related metrics over the resource usage cost. In this scenario, at the beginning of the experiment EDF determines a good parallelism degree that avoids response time violations, and then never changes the deployment. As we could expect, in this configuration the presence of the token bucket at the global layer does not practically have any impact, since no scaling actions are requested after the initial ones.

This experiment shows that the model-based RL policy overcomes the limitation of the Q-learning approach. Indeed, by efficiently exploiting the knowledge of the system dynamics, the model-based RL policy achieves faster convergence than Q-learning (as appears from Fig. 5e). Moreover, the model-based RL solution benefits from the strengths of RL approaches, which allow to express *what* the user aims to obtain, instead of *how* it should be obtained (as required by the threshold-based policy). It is true that, according to our model, the user needs also to define some policy parameters (i.e., the cost function weights). However, we believe that – for a user – it is way easier to express the relative importance of utility factors (e.g., avoid violations, reduce resource cost) rather than tune system-dependent metrics (e.g., scale-out threshold on the average CPU utilization). Last but not least, by comparing Fig. 5f against Fig. 5e, we readily note that our hierarchical control solution can be effectively adopted in distributed environments. Indeed, while the Operator Managers optimize a local cost function, the global view of the system exploited by the Application Manager allows to guide and adapt the overall system behavior in a lightweight manner.

10. Conclusions

In this paper, we have presented Elastic and Distributed DSP Framework (EDF), a hierarchical autonomous control for elastic DSP applications. Designed according to the decentralized hierarchical 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 use a local policy to control the adaptation of DSP operators by means of scaling and migration actions. At the higher level, a per-application centralized component oversees the overall DSP application performance. It uses a global policy to solve possible reconfiguration conflicts and conveniently limit the number of performed reconfigurations, aiming to reduce the application downtime.

Within this framework, we have considered different approaches for defining hierarchical control policies. As regards the local scaling policy, a first baseline solution relies on the simple threshold-based approach, which is widely used in literature. Aiming to design a more flexible self-adaptation strategy, we have investigated reinforcement learning based approaches, where distributed agents learn which are the most valuable reconfiguration actions to perform. Specifically, we have presented and evaluated a model-free and a model-based RL algorithm, which exploit different levels of available knowledge about the system dynamics. As regards the global policy, we have described a lightweight token bucket mechanism to control the application reconfigurations performed.

Relying on a reference application that processes real-time data generated by taxis in New York City, we conducted an experimental evaluation of the different policies. The results have shown that a simple threshold-based policy cannot guarantee good performance whenever the application behavior is not easily predictable. While the Q-learning based policy suffers from the slow convergence velocity of the underlying model-free learning algorithm, our model-based RL policy clearly outperforms the other solutions. Differently from the simple threshold based approach, the RL based solution can also account for other QoS metrics (e.g., the number of reconfigurations performed), providing the user with the flexibility for weighting the relative importance of each metric. Furthermore, the experiments have also demonstrated that a lightweight yet effective global policy can improve the overall performance of the system, with respect to a fully decentralized control solution.

As future work, we plan to further investigate RL approaches for elasticity. We will investigate more sophisticated techniques for improving the convergence speed of the learning process (e.g., by leveraging Bayesian Decision Trees, Function Approximation). Moreover, we plan to extend our model by explicitly considering network latencies within the self-adaptive policies. To this end, we plan to investigate the challenges of scaling DSP operators in a network-aware manner, where the presence of network latencies influences the scaling decisions. As regards the global policy, we plan to design more complex solutions that can pro-actively guide the local components behavior by providing more informative feedback. For example, the global policy could adjust the parameters of the local policies so to more efficiently operate under different working conditions (e.g., stable or critical application performance).

References

[1] B. Gedik, S. Schneider, M. Hirzel, K.-L. Wu, Elastic scaling for data stream processing, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1447–1463.
[2] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, et al., Network-aware operator placement for stream-processing systems, in: Proc. of IEEE ICDE '06, 2006.
[3] C. Hochreiner, M. Vögler, S. Schulte, S. Dustdar, Elastic stream processing for the Internet of Things, in: Proc. of IEEE Cloud 2016, 2016, pp. 100–107.
[4] S. Rizou, F. Durr, K. Rothermel, Solving the multi-operator placement problem in large-scale operator networks, in: Proc. of ICCCN '10, 2010, pp. 1–6.
[5] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Distributed QoS-aware scheduling in Storm, in: Proc. of ACM DEBS '15, 2015, pp. 344–347.
[6] V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo, Towards hierarchical autonomous control for elastic data stream processing in the fog, in: Proc. of Euro-Par 2017: Parallel Processing Workshops, in: LNCS, vol. 7475, 2018, pp. 106–117.
[7] J. Kephart, D. Chess, The vision of autonomic computing, IEEE Comput. 36 (1) (2003) 41–50.
[8] T. Chen, R. Bahsoon, X. Yao, A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems, ACM Comput. Surv. (2018).
[9] V. Gulisano, R. Jiménez-Peris, M. Patiño Martínez, C. Soriente, P. Valduriez, StreamCloud: An elastic and scalable data streaming system, IEEE Trans. Parallel Distrib. Syst. 23 (12) (2012) 2351–2365.
[10] V. Cardellini, M. Nardelli, D. Luzi, Elastic stateful stream processing in storm, in: Proc. of HPCS '16, IEEE, 2016, pp. 583–590.
[11] R.C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, Integrating scale out and fault tolerance in stream processing using operator state management, in: Proc. of ACM SIGMOD '13, 2013, pp. 725–736.
[12] T. Heinze, V. Pappalardo, Z. Jerzak, C. Fetzer, Auto-scaling techniques for elastic data stream processing, in: Proc. of IEEE ICDEW '14, 2014, pp. 296–302.
[13] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, Cambridge, 1998.
[14] A. Floratou, A. Agrawal, Self-regulating streaming systems: challenges and opportunities, in: Proc. of Int'l Workshop on Real-Time Business Intelligence and Analytics, BIRTE '17, ACM, 2017, pp. 1:1–1:5.
[15] V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo, Auto-scaling in data stream processing applications: A model based reinforcement learning approach, in: InfQ 2017 – New Frontiers in Quantitative Methods in Informatics, in: CCIS, vol. 825, 2018.
[16] Z. Jerzak, H. Ziekow, The DEBS 2015 grand challenge, in: Proc. of ACM DEBS '15, 2015, pp. 266–268.
[17] V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo, Optimal operator deployment and replication for elastic distributed data stream processing, Concurr. Comput.: Pract. Exper. 30 (9) (2018).
[18] L. Aniello, R. Baldoni, L. Querzoni, Adaptive online scheduling in Storm, in: Proc. of ACM DEBS '13, 2013, pp. 207–218.
[19] T.Z.J. Fu, J. Ding, R.T.B. Ma, M. Winslett, et al., DRS: Auto-scaling for real-time stream analytics, IEEE/ACM Trans. Netw. (2017) 1–15.

[20] K.G.S. Madsen, Y. Zhou, J. Cao, Integrative dynamic reconfiguration in a parallel stream processing engine, in: Proc. of IEEE ICDE '17, 2017, pp. 227–230.

[21] J. Xu, Z. Chen, J. Tang, S. Su, T-Storm: traffic-aware online scheduling in Storm, in: Proc. of IEEE ICDCS '14, 2014, pp. 535–544.

[22] X. Liu, A.V. Dastjerdi, R.N. Calheiros, C. Qu, R. Buyya, A stepwise auto-profiling method for performance optimization of streaming applications, ACM Trans. Auton. Adapt. Syst. 12 (4) (2018) 24:1–24:33.

[23] G. Mencagli, A game-theoretic approach for elastic distributed data stream processing, ACM Trans. Auton. Adapt. Syst. 11 (2) (2016) 13:1–13:34.

[24] F. Lombardi, L. Aniello, S. Bonomi, L. Querzoni, Elastic symbiotic scaling of operators and resources in stream processing systems, IEEE Trans. Parallel Distrib. Syst. 29 (3) (2018) 572–585.

[25] T. De Matteis, G. Mencagli, Elastic scaling for distributed latency-sensitive data stream operators, in: Proc. of PDP '17, 2017, pp. 61–68.

[26] B. Lohrmann, P. Janacik, O. Kao, Elastic stream processing with latency guarantees, in: Proc. of IEEE ICDCS '15, 2015, pp. 399–410.

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

[28] L. Xu, B. Peng, I. Gupta, Stela: Enabling stream processing systems to scale-in and scale-out on-demand, in: Proc. of IEEE IC2E '16, 2016, pp. 22–31.

[29] R.K. Kombi, N. Lumineau, P. Lamarre, A preventive auto-parallelization approach for elastic stream processing, in: Proc. of IEEE ICDCS '17, 2017, pp. 1532–1542.

[30] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, et al., Storm@Twitter, in: Proc. of ACM SIGMOD '14, 2014, pp. 147–156.

[31] J. Li, C. Pu, Y. Chen, D. Gmach, D. Milojicic, Enabling elastic stream processing in shared clusters, in: Proc. of IEEE CLOUD '16, 2016, pp. 108–115.

[32] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, et al., Twitter Heron: Stream processing at scale, in: Proc. of ACM SIGMOD '15, 2015, pp. 239–250.

[33] A. Floratou, A. Agrawal, B. Graham, S. Rao, K. Ramasamy, Dhalion: self-regulating stream processing in Heron, in: Proc. VLDB Endow. 10 (12) (2017) 1825–1836.

[34] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: Fault-tolerant streaming computation at scale, in: Proc. of ACM SOSP'13, 2013, pp. 423–438.

[35] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, et al., Apache flink: Stream and batch processing in a single engine, Bull. IEEE Comput. Soc. Tech. Committee Data Eng. 36 (4) (2015) 28–38.

[36] P. Carbone, S. Ewen, G. Fóra, S. Haridi, et al., State management in Apache Flink: Consistent stateful distributed stream processing, Proc. VLDB Endow. 10 (12) (2017) 1718–1729.

[37] H.P. Sajjad, K. Danniswara, A. Al-Shishtawy, V. Vlassov, SpanEdge: Towards unifying stream processing over central and near-the-edge data centers, in: Proc. of 2016 IEEE/ACM Symp. on Edge Computing, 2016, pp. 168–178.

[38] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, et al., Incremental deployment and migration of geo-distributed situation awareness applications in the fog, in: Proc. of ACM DEBS '16, 2016, pp. 258–269.

[39] D. Weyns, B. Schmerl, V. Grassi, S. Malek, et al., On patterns for decentralized control in self-adaptive systems, in: Software Engineering for Self-Adaptive Systems II, in: LNCS, vol. 7475, Springer, 2013, pp. 76–107.

[40] T. Heinze, L. Aniello, L. Querzoni, Z. Jerzak, Cloud-based data stream processing, in: Proc. of ACM DEBS '14, 2014, pp. 238–245.

[41] K.P. Yoon, C.-L. Hwang, Multiple Attribute Decision Making: An Introduction, vol. 104, Sage Publications, 1995.

[42] R. Bellman, Dynamic Programming, first ed., Princeton University Press, Princeton, NJ, USA, 1957.

[43] F. Dabek, R. Cox, F. Kaashoek, R. Morris, Vivaldi: A decentralized network coordinate system, SIGCOMM Comput. Commun. Rev. 34 (4) (2004) 15–26.

Valeria Cardellini is Associate Professor in the Department of Civil Engineering and Computer Science of the University of Rome Tor Vergata. She received the Doctorate degree in computer science from the University of Rome Tor Vergata in 2001. Her research interests are in the field of distributed computing systems, with a focus on Web and Cloud systems and services. She has more than 90 publications in international conferences and journals. She has served as TPC member of conferences on Web and performance and as frequent reviewer for well-known international journals.

Francesco Lo Presti is Associate Professor in the Department of Civil Engineering and Computer Science of the University of Rome Tor Vergata. He received the Doctorate degree in computer science from the University of Rome Tor Vergata in 1997. His research interests include measurements, modeling and performance evaluation of computer and communications networks. He has more than 70 publications in international conferences and journals. He has served as TPC member of conferences on networking and performance areas, and as reviewer for various international journals.

Matteo Nardelli is research associate at the University of Rome Tor Vergata. He received the Doctorate degree in computer science from the University of Rome Tor Vergata in 2018. His research interests are in the field of distributed systems and Cloud computing, with a special emphasis on the deployment of data stream processing applications in geo-distributed environments.

Gabriele Russo Russo graduated in Computer Engineering with "honors" in 2017 at the University of Rome Tor Vergata. He is currently a Ph.D. student at University of Rome Tor Vergata. His research interests span the area of Distributed Systems with emphasis on Performance Optimization.