CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2017; **00**:1–25
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Optimal Operator Deployment and Replication for Elastic Distributed Data Stream Processing

Valeria Cardellini[1]*, Francesco Lo Presti[1], Matteo Nardelli[1], and Gabriele Russo Russo[1]

[1]*Dept. of Civil Engineering and Computer Science Engineering, University of Rome Tor Vergata, Italy*

SUMMARY

Processing data in a timely manner, Data Stream Processing (DSP) applications are receiving an increasing interest for building new pervasive services. Due to the unpredictability of data sources, these applications often operate in dynamic environments, therefore they require the ability to elastically scale in response to workload variations. In this paper, we deal with a key problem for the effective runtime management of a DSP application in geo-distributed environments: we investigate the placement and replication decisions while considering the application and resource heterogeneity and the migration overhead, so to select the optimal adaptation strategy that can minimize migration costs while satisfying the application Quality of Service (QoS) requirements. We present Elastic DSP Replication and Placement (EDRP), a unified framework for the QoS aware initial deployment and runtime elasticity management of DSP applications. In EDRP, the deployment and runtime decisions are driven by the solution of a suitable integer linear programming problem, whose objective function captures the relative importance between QoS goals and reconfiguration costs. We also present the implementation of EDRP and the related mechanisms on Apache Storm. We conduct a thorough experimental evaluation, both numerical and prototype-based, that shows the benefits achieved by EDRP on the application performance. Copyright © 2017 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Data stream processing, elasticity, deployment, replication, stateful migration, Apache Storm

1. INTRODUCTION

The advent of the Big Data era and the diffusion of the Cloud computing paradigm have renewed the interest in Data Stream Processing (DSP) applications. Exploiting on-the-fly computation, i.e., without storing data, DSP applications are widely used to process unbounded streams of data and extract valuable information in a near real-time fashion. As such, they enable the development of new intelligent and pervasive services that can improve our everyday life in several domains, e.g., health-care, energy management, logistic, and transportation.

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. An operator can be either stateless (e.g., filtering, projection) or stateful (e.g., aggregation), the latter meaning that it stores a state built upon past processing steps.

In the Big Data era, DSP applications should be capable of seamlessly processing huge amount of data, which requires to dynamically scale their execution on multiple computing nodes. To deal with the fact that some operators in the application can be overloaded and thus become a

*Correspondence to: V. Cardellini, Dept. of Civil Engineering and Computer Science Engineering, University of Rome Tor Vergata, via del Politecnico 1, 00133 Rome, Italy. E-mail: cardellini@ing.uniroma2.it bottleneck, a commonly adopted stream processing optimization is *data parallelism* [1], which consists of scaling-out or scaling-in the number of parallel instances (i.e., *replicas*) for the operators, so that each replica can process a subset of the incoming data flow in parallel (e.g., [2, 3]). Due to the unpredictable rate at which the sources may produce the data streams, a static or manual configuration of the operator *replication degree* (which in literature is also referred to as parallelization degree) does not allow to effectively manage changes that can either occur in the execution environment (e.g., workload fluctuations) or in the Quality of Service (QoS) requirements of the DSP application (which is typically long-running). Therefore, a key design choice in a DSP framework is to enable it with elastic data parallelism, where the replication degree is self-configured at run-time. As expected, it is more complex to achieve elasticity for stateful operators.

Furthermore, since data sources can be geographically distributed (e.g., in Internet of Things scenarios), the deployment and execution of DSP applications can also take advantage of the ever increasing presence of distributed Cloud and Fog computing resources, which can reduce latency by moving the computation towards the network edges, closer to data sources. Nevertheless, such geo-distributed infrastructures pose new challenges to deal with, including the heterogeneity of computing and networking resources and the non-negligible network latencies [4].

This paper focuses on the initial deployment and run-time reconfiguration of DSP applications over geo-distributed computing nodes. We propose Elastic DSP Replication and Placement (**EDRP**), a unified general formulation of the elastic operator replication and placement problem. EDRP takes into account the heterogeneity of infrastructural resources and the QoS application requirements and determines the number of replicas for each operator and where to deploy them on the geo-distributed computing infrastructure. Moreover, EDRP takes into account the reconfiguration costs, which arise when an operator is migrated from one computing resource to another, as well as when a scaling-in/out decision changes the replication degree of an operator. These costs are significant in case of stateful operators, which require the relocation of the internal state on a different node in such a way that no state information is lost. As in most recent research, e.g., [2], we consider partitioned stateful operators, for which the internal state can be decomposed into partitions based on a partitioning key and each partition can be assigned to a distinct replica.

Differently from most works in literature [5, 6, 7, 8], EDRP can jointly determine the application placement and replication of its operators, while optimizing the QoS attributes of the DSP application. At run-time, by modeling the reconfiguration costs of migration and scaling operations, EDRP can be used to determine whether the application can be more conveniently redeployed.

The main research contributions of our paper are as follows.

- We model the EDRP problem as an Integer Linear Programming (ILP) problem (Sections 3, 4, and 5) which can be used to optimize different QoS metrics. Specifically, in this paper we minimize the response time, defined as the critical path delay to traverse the application DAG, and the monetary cost of all the computing and networking resources involved in the processing and transmission of the application data streams. We propose a general formulation of the reconfiguration costs caused by operator migration and scaling in terms of application downtime. The latter depends on the fact that most DSP frameworks do not support seamlessly reconfiguration, since the ongoing computation is interrupted while the migration and/or scaling operations are being carried out.
- Leveraging on our extension of Apache Storm [9], which supports partitioned stateful operator migrations, we develop a prototype scheduler named S-EDRP. S-EDRP can replicate and place the DSP application operators according to the EDRP solution (Section 6). To this end, we adapt the reconfiguration costs in order to consider the migration protocol used for relocating stateful operators.
- We extensively evaluate our proposal through numerical experiments as well as experiments based on our S-EDRP prototype (Section 7). The former aims to investigate the effectiveness of EDRP in taking the reconfiguration decisions under different scenarios of reconfiguration costs. The latter aims to evaluate the prototyped solution in a real setting; to this purpose, we use a DSP application that solves the DEBS 2015 Grand Challenge [10].

2. RELATED WORK

In this section, we review the research results and development efforts related to placing and deploying a DSP application on the computing infrastructure, including the approaches devoted to reconfiguring the deployment at runtime.

Placement and Replication Problem. The decisions about placing the operators of a DSP application and determining their replication degree can be taken at two different stages of the application lifecycle, that is at the *initial* deployment of the application on the distributed computing infrastructure and at *run-time*. Furthermore, the operator placement and operator replication can be either taken as independent and orthogonal decisions or can be jointly considered.

The DSP placement problem [11] has been widely investigated in literature under different modeling assumptions and optimization goals, e.g., [5, 8, 12]. In [12], we propose a general formulation of the optimal DSP placement which takes into account the heterogeneity of computing and networking resources and which encompasses different solutions proposed in the literature. Since the placement problem is NP-hard, several heuristics have been proposed, e.g. [13, 14, 15]. They aim at minimizing a diversity of utility functions, such as DSP application latency, inter-node traffic, and network usage. Although in this paper we focus on application latency and execution cost, our problem formulation can be easily extended to take into account different utility functions.

Most works in literature consider DSP operator placement and replication as independent and orthogonal decisions, where the placement is first carried out without determining the optimal number of replicas for each operator. Then, in response to some performance deterioration, the operators to be replicated and their new replication degree are identified. This two-stage approach requires to reschedule the DSP application in order to enact the new application configuration. Since most existing DSP frameworks require to restart the application to adapt its deployment (e.g., Apache Storm [16]), performing a reconfiguration may incur in a significant downtime. In [17] we proposed a *single-stage* approach to jointly define both the placement and the replication degree of the operators in a DSP application. With respect to the problem studied in [17], which only considered the initial DSP deployment, in this paper we deal with the elastic run-time adaptation.

Elasticity is a key feature for DSP systems that is attracting many research efforts. Most approaches that enable elasticity are intrinsically organized according to the MAPE (Monitor, Analyze, Plan and Execute) reference model for self-adaptive systems [18]. Following this model, the current placement and replication is adapted by scaling-in/out the amount of operator replicas and possibly rescheduling the application in response to changes, either observed or predicted, in some monitored performance metric. Some works, e.g., [9, 19, 20, 21], exploit best-effort threshold-based policies based on the utilization of either the system nodes or the operator replicas. Different approaches can be identified for the threshold definition. A single statically-defined threshold is used, e.g., in [20] to limit load unbalance among computing nodes. Multiple statically-defined thresholds are used, e.g., in [19] to customize the behavior of each individual node within the system. A dynamically set threshold improves the system adaptivity, e.g., in [21] where a reinforcement learning approach is proposed. Overall, when the utilization exceeds the threshold, the replication degree of the involved operators is modified accordingly.

Other works, e.g., [2, 6, 7, 22], use more complex policies to determine the scaling decisions. Lohrmann et al. [6] propose a strategy that enforces latency constraints by relying on a predictive latency model based on queueing theory; nevertheless, their solution manages only stateless DSP applications. Mencagli [7] presents a game-theoretic approach where the control logic is distributed on each operator. Stela [22] relies on throughput-based metric to identify those operators that need to be scaled; it has been integrated into Storm, but stateful operators do not appear to be managed.

The works most closely related to ours have been presented in [3, 23, 24]. Heinze et al. [3, 23] propose a model to estimate latency spikes caused by operator reallocations, and use it to define a placement heuristic. It results a solution that places only the newly added operators and minimizes the latency violations. Our approach differs in that we propose an optimal problem formulation which deals with the reconfiguration costs of the entire DSP application. Moreover, it represents a benchmark against which heuristics can be compared. Madsen et al. [24] formulate a MILP optimization problem aimed to control load balancing and horizontal scaling, which works in combination with a heuristic in charge of collocating operators on computing nodes. Similarly to our work, their solution considers operator collocation (in our case, it follows from minimizing the response time) and state-migration overheads. However, we further consider the network among computing nodes, which impacts on placement, replication, and also on migration costs.

Differently from the above cited works that present reactive scaling strategies, De Matteis and Mencagli [25] propose a proactive strategy for a DSP distributed environment that takes into account a limited future time horizon to choose the reconfigurations. Differently from our solution, their approach is not integrated in an existing DSP framework.

As regards the placement and deployment of DSP applications in geo-distributed environments, we have extended Storm with a self-adaptive QoS-aware distributed placement heuristic [14]. Similarly to our proposal, the recently presented SpanEdge approach [26] is implemented in Storm, but it does not support operator migrations. Saurez et al. [27] propose Foglets, a programming model specifically designed for the Fog computing environment, which supports the migration of application components. Our EDRP formulation could be integrated into Foglets, as it is designed to operate in a geo-distributed infrastructure.

Stateful Operators. While scaling-in/out stateless operators can be achieved by just turning off/on operator replicas, elasticity of stateful operators requires state migration and repartitioning among the replicas, because the system needs to preserve the consistency of the operations [2]. State management in DPS systems is nicely surveyed by To et al [28]; in the following we focus on the issues that are more closely related to our work.

Operator state migration is a challenging task, because it should be application-transparent and with a minimal footprint (i.e., amount of migrated state). The most common solutions are the *pause-and-resume* approach (that we adopt in this paper) and the *parallel track* approach [29]. In the pause-and-resume approach, the current state is extracted from the old operator instance, which is paused to ensure a semantically correct migration; then, the state is moved to the new instance and the buffered tuples are rerouted and replayed within the new instance. Its drawback is a peak in the application latency during the migration. To migrate in a more gradual fashion, in the parallel track approach, the old and the new operator instances run concurrently until the state of both is synchronized and the new instance can safely take over. While this approach does not entail a latency peak, it requires enhanced mechanisms [19], e.g., to avoid incorrect results, which can increase the cost of state migration. Furthermore, this approach is more suitable for window-based operators [30]. This latter work also models the time cost of the two approaches for partitioned stateful operators. To improve the efficiency of the pause-and-resume approach, some works [19, 31, 32] exploit the checkpoints that are already backed up for failure recovery. A checkpoint allocation problem is defined by Madsen et al. [31] with the goal to maximize the checkpoints reuse for migration.

The second challenge posed by stateful operators regards the stream partitioning and load balancing among the replicas. When a stateful operator scales out, its processing state must be split across the number of replicas and this restricts the exploitation of elasticity only to partitioned stateful operators. Fernandez et al. expose an API to let the user manually manage the state [19], while Gedik et al. [2] propose a stream partitioning method based on consistent hashing and a state management API to support transparent elastic replication. Nasir et al. [33] propose partial key grouping, an elegant solution relying on two hash functions where a key can be sent to two different replicas instead of one. In our approach, the minimum unit of migratable state is defined by the user through the Storm execution model and the load is distributed according to key partitioning.

DSP Frameworks. Aside the specific functionalities, the most popular open-source DSP frameworks (Storm, Spark Streaming, Flink, and Heron) use directed graphs to model DSP applications. They provide an abstraction layer where to execute DSP applications and allow their users to focus solely on the application logic, being the tasks related to the application placement, distribution, and execution managed by the frameworks themselves. As regards the operator elasticity, in most cases these frameworks require their users to manually tune the number of replica per operator. Since the user might over-/under-estimate the expected load, this approach can lead to a sub-optimal provisioning of resources. Furthermore, 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 6.1. Leveraging on Trident [34], Storm supports stateful applications. Differently from our extension, that manages an operator-related state, Trident can persist a state which is obtained by applying a sequence of Trident transformations on the input data. However, this approach requires to play the stream as a sequence of micro-batches, processed in a commit-like fashion, thus causing a constant latency overhead. In [9] we extend Storm to support the elastic run-time adaptation of DSP applications, by introducing new system components that allow the elasticity and stateful migration of the DSP operators. In this paper, we combine that extension with our Distributed Storm [35] to exploit the distributed monitoring capabilities of the latter. We present in Section 6 the resulting enhanced framework. An approach to support elastic scaling of DSP applications in Storm has been also presented at the same time in [36]; interestingly, their proposal reduces the interruption due to scaling operations by keeping the application running while scaling, instead of shutting down the application operators and restarting them. However, they considered a clustered architecture and their improved version of Storm has not been released publicly. Therefore, in our experiments we use the standard rebalancing command of Storm.

Developed by Twitter as the successor of Storm, Heron [37] preserves Storm's abstraction layer while introducing some improvements and a multi-layer architecture. Dhalion [38] is a newly presented framework on top of Heron that provides elastic capabilities to the underlying streaming system; it also addresses the overhead related to restarting the topology by allowing its updating. However, at the time of writing Dhalion has not yet been open-sourced and details on the elasticity strategy are not yet available. Spark Streaming [39] 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 [40], which provides a unified solution for batch and stream processing. Although Flink supports the manual scaling of stateless operators and checkpointing of stateful operators for fault tolerance, it does not yet provide any mechanism for autonomic operator scaling and for stateful operator migration. We also observe that, despite the recent efforts towards elasticity in some frameworks, all those cited are not designed to efficiently operate in a geo-distributed environment.

DSP systems are also offered as Cloud services. Google Cloud Dataflow[†] provides a unified programming model to process batch and streaming data. Amazon offers Kinesis Streams[‡] to process near real-time streams of data. Both of them abstract the underlying infrastructure and support dynamic scaling of the computing resources; however, it appears that they execute in a single data center, conversely to the geo-distributed environment we investigate in this paper.

Several research efforts use Storm to evaluate operator placement algorithms or some architectural improvements (e.g., [13, 15, 16, 36, 41] and, from our group, [9, 12, 35]). In this paper, we also implement the proposed EDRP strategy into Storm.

3. SYSTEM MODEL

In this section we present the system resource, deployment and reconfiguration models. For the sake of clarity, in Table I we summarize the notation used throughout the paper.

3.1. Resource Model

Computing and network resources can be represented as a labeled fully connected directed graph $G_{res} = (V_{res}, E_{res})$, where the set of nodes V_{res} represents the distributed computing resources,

[†]https://cloud.google.com/dataflow/
[‡]https://aws.amazon.com/kinesis/

Table I. Description of main symbols used in the model.

Symbol	Description	Symbol	Description
G_{dsp}	Graph representing the DSP application	Res_i	Required resources to execute $i \in V_{dsp}$
V_{dsp}	Set of vertices (operators) of G_{dsp}	$I_{S,i}$	Internal state size of $i \in V_{dsp}$
E_{dsp}	Set of edges (streams) of G_{dsp}	$I_{C,i}$	Operator code size of $i \in V_{dsp}$
C_i	Cost of deploying $i \in V_{dsp}$	$\lambda_{(i,j)}$	Avg. tuple rate on $(i,j) \in E_{dsp}$
R_i	Latency of $i \in V_{dsp}$	λ_i	Avg. incoming tuple rate on $i \in V_{dsp}$
G_{res}	Graph representing computing resources	$d_{(u,v)}$	Network delay on $(u,v) \in E_{res}$
V_{res}	Set of vertices (comp. nodes) of G_{res}	$r_{(u,v)}$	Transfer rate on $(u,v) \in E_{res}$
E_{res}	Set of edges (network links) of G_{res}	$C_{(u,v)}$	Cost for data unit on $(u,v) \in E_{res}$
Res_u	Resources available on $u \in V_{res}$	$t_{s,u}$	Time for instance spawning on $u \in V_{res}$
S_u	Processing speed-up of $u \in V_{res}$	t_{syn}	Synchronization time for reconfiguration
$V_{res}^i \subseteq V_{res}$	Subset of nodes where to place $i \in V_{dsp}$	$x_{i,\mathcal{U}}$	Placement of $i \in V_{dsp}$ on nodes in $\mathcal{U}$
$\mathcal{S} \sqsubset S$	Multiset of elements in set S	$\mathcal{U}_{0,i}$	Current deployment for $i \in V_{dsp}$
$\mathcal{P}(S;k)$	Set of all multisets with elements taken in S and cardinality no greater than k	$y_{(i,j),(\mathcal{U},\mathcal{V})}$	Placement of $(i,j) \in E_{dsp}$ on network paths between $\mathcal{U}$ and $\mathcal{V}$

and the set of links E_{res} represents the *logical connectivity* between nodes. Observe that, at this level, links represent the logical links across the networks corresponding to the network paths between nodes (as determined by the network operator routing strategies). Each node $u \in V_{res}$ is characterized by: Res_u, the amount of available resources; and S_u, the processing speed-up on a reference processor. Each link $(u, v) \in E_{res}$, with $u, v \in V_{res}$ is characterized by: $d_{(u,v)}$, the network delay between node u and v; $r_{(u,v)}$, the transfer rate between node u and v; and $C_{(u,v)}$, the cost per unit of data transmitted along the network path between u and v. This model considers also edges of type (u, u) (i.e., loops); they capture network connectivity between operators placed in the same node u, and are considered as perfect links, i.e., always active with no network delay. We assume that the considered QoS attributes can be obtained by means of either active/passive measurements or through some network support (e.g., SDN).

3.2. DSP Deployment Model

A DSP application can be represented at different levels of abstraction; we distinguish between a user-defined *abstract model* and an *execution model*, used to run the application.

The DSP *abstract model* defines the streams and their characteristics, along with the type, role, and granularity of the stream processing elements. At this level, the DSP application can be regarded as a network of operators connected by streams. An operator is a self-contained processing element that carries out a specific operation (e.g., filtering, aggregation, merging) or something more complex (e.g., POS-tagging), whereas a stream is an unbounded sequence of data (e.g., packet, tuple, file chunk). Moreover, we distinguish between stateless and stateful operator whether the operator computes the output data using only the incoming data or also some internal state information, respectively. A DSP abstract model can be represented as a labeled directed acyclic graph (DAG) $G_{dsp} = (V_{dsp}, E_{dsp})$, where the nodes in V_{dsp} represent the application operators as well as the data stream sources (i.e., nodes without incoming links) and sinks (i.e., nodes without outgoing links), and the links in E_{dsp} represent the streams, i.e., data flows, between nodes. Due to the difficulties in formalizing the non-functional attributes of an abstract operator, we characterize it with the non-functional attributes of a reference implementation on a reference architecture: Res_i, the amount of resources required for running the operator; R_i, the average operator instance latency (which accounts for the waiting time on the input queues as well as the execution time of a data unit); and C_i, the cost of deploying an operator instance. We characterize the stream exchanged from operator i to j, $(i, j) \in E_{dsp}$, with its average tuple rate $\lambda_{(i,j)}$. To model load-dependent latency, we assume that R_i is a function of the operator input tuple rate λ_i, that is, $R_i = R_i(\lambda_i)$, where $\lambda_i = \sum_{j \in V_{dsp}} \lambda_{(j,i)}$; without loss of generality, we also assume that R_i is an increasing function in λ_i. In this paper, we assume that Res_i is a scalar value, but our placement model can be easily extended to consider Res_i as a vector of required resources (e.g., CPU cores, memory).

At any give time, the DSP *execution model* is obtained from the abstract model by replacing each operator with the current number of operator replicas, that is operator instances each of which processes a subset of the incoming data flow. By partitioning the stream over multiple replicas, running on one or more computing nodes, the load of each replica is reduced, which, in turn, yields lower operator (and overall application) latency. Since the load can vary over time, the number of replicas in the execution model can change accordingly as to optimize some non functional requirements, e.g., response time.

3.3. Reconfiguration Model

When a DSP application is launched, an initial number of replicas and their placement is determined based on the current (expected) load and QoS requirements. Then, at run-time, changes of QoS attributes of the application and the execution environment (e.g., load, latency) can call for the application reconfiguration. The latter aims to preserve high application performance, facing the dynamism of the run-time environment. A DSP application can be reconfigured by combining migrations and scaling operations. A *migration* moves an operator replica to a different computing resource, so to optimize resource utilization and, in turn, application performance. A *scaling* operation changes the replication degree of an operator. Specifically, a scale-out decision increases the number of replicas when the operator needs more computing resources, whereas a scale-in decreases the number of replicas when the operator under-uses its resources.

To perform a reconfiguration while preserving the application integrity in terms of extracted information, we assume a simple pause-and-resume approach, as introduced in Section 2. Its drawback is the application downtime for the entire duration of the reconfiguration process, which negatively impacts on the application perceived QoS.

Replica Migration. Migrating an operator replica using the pause-and-resume approach involves the following operations. First, the DSP system terminates the replica running on the old location and stops the related upstream operators from emitting data towards the operator under reconfiguration. Then, the operator *code*[§] is copied from an external repository to the new location, if the latter does not hold it, i.e., if no other replica is currently deployed there. Moreover, if the replica is an instance of a stateful operator, the DSP system has to migrate the replica internal state from the old location to the new one. Finally, the DSP system starts the new replica and resumes the application execution. We consider that the DSP system performs the code and state migrations leveraging on a storage system, named *DataStore*. The DataStore acts as repository for the operators code and allows replicas to save and restore their state during reconfigurations.

Scaling Operation. The complexity of changing at run-time the replication degree of an operator depends on the presence of its internal state. If the operator is stateless, a scaling operation involves only adding or removing replicas. When a new replica is added, the DSP system also determines *where* the replica should be executed; as a consequence of this decision, the operator code may be transferred from the DataStore to the new location. If the operator is stateful, a scaling operation has also to redistribute the operator internal state among its replicas, so to preserve the application integrity. Similarly to most of the existing solutions (e.g., [2, 16, 20, 31, 32]), we model the operator internal state as a set of key-value pairs, where the key identifies the smallest and indivisible state entity. These keys are partitioned and assigned to the operator replicas. After a scaling operation, the keys are redistributed with the help of the DataStore.

4. ELASTIC OPERATOR REPLICATION AND PLACEMENT MODEL

In this section, we present the EDRP elastic replication and placement model and derive the expressions of the QoS metrics of interest.

[§]We use the term *code* to refer to the entity that encapsulates the execution logic of an operator. Depending on the specific system implementation, it can correspond, e.g., to Java classes, executables, or container/virtual machine images.

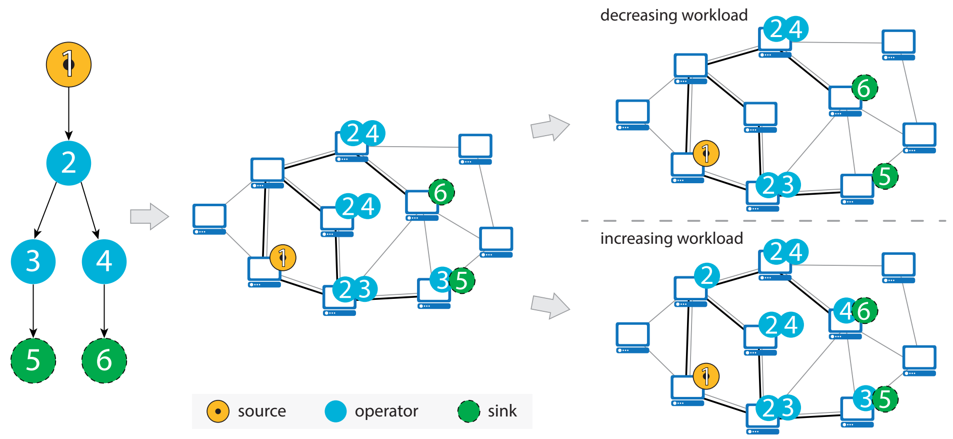


Figure 1. Elastic replication of the application operators and their placement. The right part of the figure shows: (top) scale-in of operators 2, 3 and 4 due to load decrease, (bottom) scale-out of operators 2 and 4 due to load increase. The figure does not differentiate between stateful or stateless operators.

4.1. *Operator Replication and Placement*

The EDRP problem consists in determining, for each operator $i \in V_{dsp}$, the number of replicas and where to deploy them on the computing nodes in V_{res}. EDRP can evaluate the application deployment either periodically or in response to changes on the execution environment (e.g., workload variations), so to execute application with high QoS even in dynamic running conditions. EDRP can identify the optimal reconfiguration strategy and evaluate its convenience in terms of application downtime. Figure 1 represents a simple instance of the problem. The operator replication and placement can be represented by a function map which maps an operator $i \in V_{dsp}$ to a multiset of computing nodes in V^i_{res}, where $V^i_{res} \subseteq V_{res}$ denotes the subset of nodes where an operator can be deployed[¶]. We resort to multisets because a deployment can place multiple replicas of the same operator on the same computing node. For instance, $map(i) = \{u, u, v\}$, $i \in V_{dsp}$, $u, v \in V^i_{res}$, indicates that operator i deployment consists of 3 replicas, two of which on node u and one on node v. A multiset $\mathcal{S}$ over a set S, which we denote as $\mathcal{S} \sqsubset S$, is defined as a mapping $\mathcal{S} : S \rightarrow \mathbb{N}$ where, for $s \in S$, $\mathcal{S}(s)$ denotes the multiplicity of s in $\mathcal{S}$. Hence, $s \in \mathcal{S}$ if and only if $\mathcal{S}(s) \geq 1$. The cardinality of a multiset $\mathcal{S}$, denoted $|\mathcal{S}|$, is defined by the number of elements in $\mathcal{S}$, that is $|\mathcal{S}| = \sum_{s\in\mathcal{S}} \mathcal{S}(s)$. Hereafter, without lack of generality, we assume that in a deployment each operator $i \in V_{dsp}$ can be replicated at most k_i times. We also define the power multiset $\mathcal{P}(S)$ of a set S as the set of all multisets with elements taken from S and the subset $\mathcal{P}(S; k) \subset \mathcal{P}(S)$ of the multiset over S with cardinality no greater than k, that is $\mathcal{P}(S; k) = \{\mathcal{S} \in \mathcal{P}(S) | \sum_{s\in\mathcal{S}} \mathcal{S}(s) \leq k\}$.

We model the EDRP problem with binary variables $x_{i,\mathcal{U}}$, $i \in V_{dsp}$ and $\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)$: $x_{i,\mathcal{U}} = 1$ if and only if $map(i) = \mathcal{U}$, that is, i is replicated in $|\mathcal{U}|$ instances with exactly $\mathcal{U}(u)$ copies deployed in u, with $u \in \mathcal{U}$. We also find convenient to consider binary variables associated to links, namely $y_{(i,j),(\mathcal{U},\mathcal{V})}$, with $(i, j) \in E_{dsp}$, $\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)$, and $\mathcal{V} \in \mathcal{P}(V^j_{res}; k_j)$, which denotes whether the data stream flowing from operator i to operator j traverses the network paths from nodes in $\mathcal{U}$ to nodes in $\mathcal{V}$. By definition, we have $y_{(i,j),(\mathcal{U},\mathcal{V})} = x_{i,\mathcal{U}} \wedge x_{j,\mathcal{V}}$. For short, in the following we denote by $\boldsymbol{x}$ and $\boldsymbol{y}$ the placement vectors for nodes and edges, respectively, where $\boldsymbol{x} = \langle x_{i,\mathcal{U}} \rangle$, $\forall i \in V_{dsp}$, $\forall \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)$, and $\boldsymbol{y} = \langle y_{(i,j),(\mathcal{U},\mathcal{V})} \rangle$, $\forall x_{i,\mathcal{U}}, x_{j,\mathcal{V}} \in \boldsymbol{x}$.

[¶] A DSP operator cannot be usually placed on every node in V_{res}, because of physical (i.e., pinned operator) or other motivations (e.g., security, privacy). Being application-dependent and without impact on the problem formulation, we do not explicitly formalize the function that, given $i \in V_{dsp}$ and V_{res}, identifies V^i_{res}. By default, we consider $V^i_{res} = V_{res}$ for unpinned operator i, and $V^j_{res} \subseteq V_{res}$, $|V^j_{res}| = 1$, for pinned operator j, where the node in V^j_{res} is defined a-priori.

4.2. *QoS Metrics*

In this paper, we consider as QoS metrics of interest the application response time and deployment cost. Since DSP applications are usually employed in latency-sensitive domains (e.g., [10]), a desired placement should minimize the response time resulting from the operators deployment. However, if on the one hand a higher replication degree leads to lower response time, on the other hand it incurs in higher resource wastage, which leads to a higher deployment cost. We first formulate these metrics for operators and streams in isolation, then derive the expressions for a DSP application. The analysis can be easily extended to include other QoS metrics, as application availability and network related metrics (e.g., see [17]).

4.2.1. Operator QoS Metrics. For each $i \in V_{dsp}$, the QoS of the operator depends on the hosting resources, that is its deployment $\mathcal{U}$. Let $R_{i,\mathcal{U}}$ and $C_{i,\mathcal{U}}$ denote the maximum latency [‖] and the cost experienced when i runs on $\mathcal{U}$, respectively. We readily have:

$$\begin{aligned} R_{i,\mathcal{U}} &= \max_{u \in \mathcal{U}} \frac{R_i\left(\frac{\lambda_i}{|\mathcal{U}|}\right)}{S_u} && (1) \\ C_{i,\mathcal{U}} &= \sum_{u \in \mathcal{U}} \mathcal{U}(u) C_i Res_i && (2) \end{aligned}$$

under the assumption that the traffic is equally split among the different operator replicas. Good load balancing can be achieved even for stateful operators relying on the stream partitioning schemes mentioned in Section 2, e.g., [2, 33]. It is worth pointing out that here we do not make any particular assumption on the actual expression of R_i. We can adopt either a queuing model closed-form expression, e.g., the response time of a M/M/1 queue (as in Section 7.2), or an experimental characterization of the operator response time as function of the load (as in Section 7.3).

4.2.2. Stream QoS Attributes. For a stream $(i, j) \in E_{dsp}$, the QoS depends on the upstream and downstream operators deployments $\mathcal{U}$ and $\mathcal{V}$. Let $d_{(i,j),(\mathcal{U},\mathcal{V})}$ and $C_{(i,j),(\mathcal{U},\mathcal{V})}$ denote the maximum latency[**] and the cost experienced when i runs on $\mathcal{U}$ and j on $\mathcal{V}$, respectively. We readily have:

$$\begin{aligned} d_{(i,j),(\mathcal{U},\mathcal{V})} &= d_{(\mathcal{U},\mathcal{V})} = \max_{u \in \mathcal{U}, v \in \mathcal{V}} d_{(u,v)} && (3) \\ C_{(i,j),(\mathcal{U},\mathcal{V})} &= \sum_{u \in \mathcal{U}, v \in \mathcal{V}} \lambda_{(i,j),(\mathcal{U},\mathcal{V})} C_{u,v} && (4) \end{aligned}$$

where $\lambda_{(i,j),(\mathcal{U},\mathcal{V})} = \frac{\lambda_{(i,j)}}{|\mathcal{U}||\mathcal{V}|} \quad u \in \mathcal{U}, v \in \mathcal{V}$ is the amount of stream (i, j) traffic exchanged between two replicas for the deployments $\mathcal{U}$ and $\mathcal{V}$.

4.2.3. DSP Application QoS Metrics. We derive the following expressions for a DSP application.
Response Time: For a DSP application, with data flowing from several sources to several destinations, there is no unique definition of response time. For any placement vector $\boldsymbol{x}$ (and resulting $\boldsymbol{y}$), we consider as response time $R(\boldsymbol{x}, \boldsymbol{y})$ the *critical path average delay*. We define the critical path of the DSP application as the set of nodes and edges, forming a path from a data source to a sink, for which the sum of the operator computational latency ($R_{i,\mathcal{U}}$) and network delays

[‖]This definition of the operator response time might appear counter intuitive. It is nevertheless consistent with our definition of the DSP application response time (see (5)-(7) below) which we define as the expected delay along the critical path of the DSP application. It is indeed easy to verify that with this definition of operator response time $R_{i,\mathcal{U}}$, since different replicas can experience different average response time, (5) corresponds to the average latency along the DSP critical path. We omit the formal proof for space limitation.
[**]Similarly to the response time definition (1), this definition might appear counter intuitive. Nevertheless, it is easy to verify that it is consistent with the DSP application response time definitions, (5)-(7). We omit the proof for space limitation.

($d_{(i,j),(\mathcal{U},\mathcal{V})}$) is maximal; hence, the critical path average delay is the expected traversal time of the critical path. Formally, we have:

$$R(\boldsymbol{x},\boldsymbol{y}) = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x},\boldsymbol{y}) \tag{5}$$

where $R_\pi(\boldsymbol{x},\boldsymbol{y})$ is the end-to-end delay along path π and Π_{dsp} the set of all source-sink paths in G_{dsp}. For any path $\pi = (i_1, i_2, \ldots, i_{n_\pi}) \in \Pi_{dsp}$, where i_p and n_π denote the p^{th} operator and the number of operators in the path π, respectively, we obtain:

$$R_\pi(\boldsymbol{x},\boldsymbol{y}) = \sum_{p=1}^{n_\pi} R_{i_p}(\boldsymbol{x}) + \sum_{p=1}^{n_\pi - 1} D_{(i_p, i_{p+1})}(\boldsymbol{y}) \tag{6}$$

where for any $i \in V_{dsp}$ and $(i,j) \in E_{dsp}$

$$R_i(\boldsymbol{x}) = \sum_{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)} R_{i,\mathcal{U}} x_{i,\mathcal{U}} \quad \text{and} \quad D_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \\ \mathcal{V} \in \mathcal{P}(V^j_{res}; k_j)}} d_{(\mathcal{U},\mathcal{V})} y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{7}$$

denote respectively the execution time of operator i when deployed on $\mathcal{U}$ and the network delay for transferring data from i to j when the two operators are mapped on $\mathcal{U}$ and $\mathcal{V}$, respectively.
Cost: We define the cost C of the DSP application as the monetary cost of all the resources and links involved in the processing and transmission of the application data streams. We have:

$$C(\boldsymbol{x},\boldsymbol{y}) = \sum_{i \in V_{dsp}} C_i(\boldsymbol{x}) + \sum_{(i,j) \in E_{dsp}} C_{(i,j)}(\boldsymbol{y}) \tag{8}$$

where

$$C_i(\boldsymbol{x}) = \sum_{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)} C_{i,\mathcal{U}} x_{i,\mathcal{U}} \quad \text{and} \quad C_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \\ \mathcal{V} \in \mathcal{P}(V^j_{res}; k_j)}} C_{(i,j),(\mathcal{U},\mathcal{V})} y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{9}$$

denote respectively the cost of deploying i on the multiset $\mathcal{U}$ and the cost of transferring data from i to j when the two operators are mapped on $\mathcal{U}$ and $\mathcal{V}$, respectively.

4.2.4. Reconfiguration-related Metric. The management operations that change the application deployment temporarily degrade the application performance introducing a downtime.
Application downtime: We express the cost of reconfigurations in terms of application downtime, so to capture the trade-offs between benefits and costs of changing the deployment. Under the assumption that different operators can be reconfigured in parallel, the overall application downtime $T_D(\boldsymbol{x})$ corresponds to the longest operator downtime:

$$T_D(\boldsymbol{x}) = \max_{i \in V_{dsp}} T_{D,i}(\boldsymbol{x}) \tag{10}$$

where $T_{D,i}(\boldsymbol{x})$ is the downtime duration for operator $i \in V_{dsp}$, which depends on its new deployment configuration and can be expressed as:

$$T_{D,i}(\boldsymbol{x}) = \sum_{\mathcal{V} \in \mathcal{P}(V^i_{res}; k_i)} t_D\left(i, \mathcal{U}_{0,i}, \mathcal{V}\right) x_{i,\mathcal{V}} \tag{11}$$

where $t_D(i, \mathcal{U}, \mathcal{V})$ represents the time needed to reconfigure the operator i from the deployment $\mathcal{U}$ to $\mathcal{V}$, and $\mathcal{U}_{0,i}$ represents the current deployment of i. The expression for $t_D(i, \mathcal{U}, \mathcal{V})$ depends on the actual type of operator reconfiguration (i.e., no reconfiguration, operator migration, and operator scaling). For the sake of readability, the derivation of their expressions is postponed to Appendix A.

5. EDRP OPTIMIZATION PROBLEM

In this section we present the EDRP optimization problem. We assume that, at each reconfiguration, the system goal is to optimize a suitable objective function which, depending on the scenario, could be aimed at optimizing different, possibly conflicting, QoS attributes. We further assume that each QoS metric must not exceed an application dependent worst case behavior. We use the Simple Additive Weighting (SAW) technique [42] to define the objective cost function $F(\boldsymbol{x}, \boldsymbol{y})$ as a weighted sum of the normalized QoS attributes of the application, as follows:

$$F(\boldsymbol{x}, \boldsymbol{y}) = w_r \frac{R(\boldsymbol{x}, \boldsymbol{y})}{R_{max}} + w_c \frac{C(\boldsymbol{x}, \boldsymbol{y})}{C_{max}} + w_d \frac{T_D(\boldsymbol{x})}{T_{D,max}} \tag{12}$$

where $w_r,\ w_c,\ w_d \geq 0$, $w_r + w_c + w_d = 1$, are weights associated to the different QoS attributes. $R_{\max}$, $C_{\max}$, and $T_{D,\max}$ are user-defined parameters which denote, respectively, the worst case bound on the expected response time, cost, and reconfiguration downtime. After normalization, each metric ranges in the interval $[0, 1]$, where the value 1 corresponds to the worst possible value.

We formulate the EDRP problem as an ILP cost minimization problem as follows:

$$\min_{\boldsymbol{x}, \boldsymbol{y}, r, t_D} F'(\boldsymbol{x}, \boldsymbol{y}, r, t_D)$$

subject to:

$$r \geq R_\pi(\boldsymbol{x}, \boldsymbol{y}) \qquad \forall \pi \in \Pi_{dsp} \tag{13}$$

$$t_D \geq T_{D,i}(\boldsymbol{x}) \qquad \forall i \in V_{dsp} \tag{14}$$

$$R(\boldsymbol{x}, \boldsymbol{y}) \leq R_{max} \tag{15}$$

$$C(\boldsymbol{x}, \boldsymbol{y}) \leq C_{max} \tag{16}$$

$$T_D(\boldsymbol{x}) \leq T_{D,max} \tag{17}$$

$$Res_u \geq \sum_{\substack{i \in V_{dsp} \\ \mathcal{U} \in \mathcal{P}(V^i_{res})}} \mathcal{U}(u) Res_i x_{i,\mathcal{U}} \qquad \forall u \in V_{res} \tag{18}$$

$$1 = \sum_{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)} x_{i,\mathcal{U}} \qquad \forall i \in V_{dsp} \tag{19}$$

$$x_{i,\mathcal{U}} = \sum_{\mathcal{V} \in \mathcal{P}(V^j_{res}; k_j)} y_{(i,j),(\mathcal{U},\mathcal{V})} \qquad \begin{array}{c} \forall (i,j) \in E_{dsp}, \\ \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \end{array} \tag{20}$$

$$x_{j,\mathcal{V}} = \sum_{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_u)} y_{(i,j),(\mathcal{U},\mathcal{V})} \qquad \begin{array}{c} \forall (i,j) \in E_{dsp}, \\ \mathcal{V} \in \mathcal{P}(V^j_{res}; k_j) \end{array} \tag{21}$$

$$x_{i,\mathcal{U}} \in \{0, 1\} \qquad \begin{array}{c} \forall i \in V_{dsp}, \\ \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \end{array} \tag{22}$$

$$y_{(i,j),(\mathcal{U},\mathcal{V})} \in \{0, 1\} \qquad \begin{array}{c} \forall (i,j) \in E_{dsp}, \\ \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \\ \mathcal{V} \in \mathcal{P}(V^j_{res}; k_j) \end{array} \tag{23}$$

In the problem formulation we use the linear objective function $F'(\boldsymbol{x}, \boldsymbol{y}, r, t_D)$, obtained from $F(\boldsymbol{x}, \boldsymbol{y})$ by replacing $R(\boldsymbol{x}, \boldsymbol{y})$ and $T_D(\boldsymbol{x})$ with the auxiliary variables r and t_D, respectively. Observe that, indeed, while F is nonlinear in $\boldsymbol{x}$ and $\boldsymbol{y}$, since $R(\boldsymbol{x}, \boldsymbol{y}) = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y})$ and $T_D(\boldsymbol{x}) = \max_{i \in V_{dsp}} T_{D,i}(\boldsymbol{x})$ are nonlinear terms, F' is linear in r and t_D as well as in $\boldsymbol{x}$ and $\boldsymbol{y}$. Equation (13) follows from (5)–(6). Since r must be larger or equal than the response time of any path and, at the optimum, r is minimized, $r = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y}) = R(\boldsymbol{x}, \boldsymbol{y})$. Similarly, Equation (14) defines t_D that, at the optimum, is $t_D = \max_{i \in V_{dsp}} T_{D,i}(\boldsymbol{x})$. Constraints (15)–(17) are the worst case bounds on the QoS metrics. The constraint (18) limits the placement of operators on a node $u \in V_{res}$ according to its available resources. Equation (19) guarantees that each operator $i \in V_{dsp}$ is placed on one and only one node $u \in V^i_{res}$. Finally, constraints (20)–(21) model the logical AND between the placement variables, that is, $y_{(i,j),(u,v)} = x_{i,u} \wedge x_{j,v}$.

Theorem 1
The EDRP problem is NP-hard.

Proof
It suffices to observe that the EDSP Replication and Placement problem is a generalization of the DSP Placement problem which has been shown to be NP-hard [12]. □

6. STORM INTEGRATION

To enable the usage of EDRP in a real DSP framework, we have developed a prototype scheduler for Apache Storm, named S-EDRP. We first briefly provide an overview of Storm; then, we present the prototype design in detail and how EDRP is adjusted to consider the prototype features.

6.1. Apache Storm

Storm is an open source, real-time, and scalable DSP system maintained by the Apache Software Foundation. It provides an abstraction layer where DSP applications can be executed 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), whereas the overlay network comprises the logical links among these nodes. In Storm, we can distinguish between an abstract application model and an execution application model. In the abstract model, an application is represented by its *topology* (see Figure 4), 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. Therefore, if the operator has some internal state (i.e., it is a stateful operator), a task handles a partition of it. In Storm, the number of tasks for an operator is statically defined. Each task processes incoming data with an at-least-once semantic, which can be turned into an exactly-once semantic relying on the Trident API (not used in this work because of its processing overhead). For the execution, one or more tasks of the *same* operator are grouped into executors. An *executor* is the smallest schedulable unit; Storm can process large data volumes in parallel by launching multiple executors for each operator. The number of executors of an operator must always be less than or equal to the number of tasks of the same operator. From an operational perspective, Storm implements the executors as threads and also introduces the *worker process*, that is a Java process, to run a subset of executors of the *same* topology. Therefore, there is an evident hierarchy among the Storm entities: a group of tasks runs sequentially in the executor, which is a thread within the worker process, that serves as container on the worker node.

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*[*], 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*.

[*]http://zookeeper.apache.org/

6.2. Elasticity in Storm

To elastically adapt the applications deployment in Storm, we resort on Distributed Storm[†], which is our extension of Storm with QoS-aware scheduling capabilities [35], and on S-EDRP, which is a new custom scheduler for Storm we propose in this work. Distributed Storm enhances the official Storm architecture by introducing key components that support the execution of the MAPE control loop. In our case, the MAPE loop controls the run-time reconfiguration of the application deployment in response to changes in the workload conditions. As shown in Figure 2, we have designed a partially decentralized control loop, that follows the master-worker pattern presented in [43]. Specifically, we have centralized the Analyze and Plan phases on Nimbus, and decentralized the Monitor and Execute phases on Nimbus and on the worker nodes, which contribute with local components. The Analyze component collects the monitored data from the decentralized Monitors on the worker nodes and periodically triggers the Plan component. The latter solves the EDRP model, thus determining if needed the reconfiguration actions (i.e., scale-in, scale-out, migrate) that improve application performance. In such a case, the decentralized Execute components perform the corresponding adaptation actions.

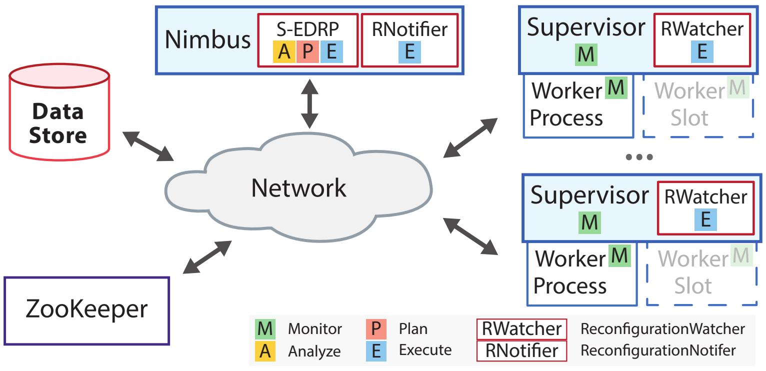


Figure 2. Extended Storm architecture.

Monitor. The first phase of the control loop enriches S-EDRP with infrastructure and application QoS-awareness. Specifically, the monitoring components of Distributed Storm running on each worker node provide to Nimbus inter-node information, such as network delay and exchanged data rate. Network latencies are estimated relying on the Vivaldi algorithm [44], a decentralized and scalable algorithm that creates a network coordinate system. The decentralized nodes send monitored data to the centralized components through ZooKeeper. Further details on our extension can be found in [35].

Analyze and Plan. In S-EDRP, the Analyze phase collects the monitored data and periodically instantiates the EDRP model. The latter has been properly adjusted, as described in Section 6.4, to represent the DSP and resource models used by Storm. The Analyze phase instantiates the EDRP model using the collected monitored data to parametrize nodes and edges in G_{dsp} and G_{res} and give value to the average data rate exchanged between communicating operator replicas, $\lambda_{(i,j)}$, and the network latencies, $d_{(u,v)}$. When the Plan phase is activated, it resolves the EDRP model relying on CPLEX©[‡], the state-of-the-art solver for ILP problems. Observe that, to smoothly integrate our solution with the existing Storm architecture, the Analyze phase uses a time-based triggering strategy to activate the Plan phase. However, more efficient strategies can be designed to recompute the deployment only when specific events occur, e.g., when QoS bounds are violated.

Execute. When EDRP devises a new application deployment, S-EDRP needs to accordingly reconfigure the replication and placement of the application operators. Storm provides an API to change the operators replication at run-time (i.e., `rebalance`); nevertheless, this operation restarts

[†]Source code available at `http://bit.ly/extstorm`
[‡]`http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`

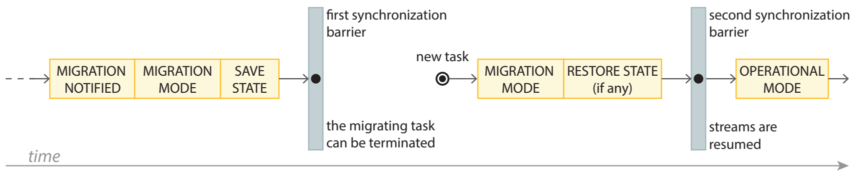


Figure 3. Sequence of operations involving a task during a reconfiguration.

the application topology, compromising the integrity of stateful operators (which lose their internal state). To overcome this issue, we further extend the Storm architecture to include mechanisms that support the migration and replication of stateful operators. In Section 6.3, we describe the newly introduced components and the adopted stateful migration protocol.

Besides the run-time adaptation, Nimbus uses S-EDRP to determine the deployment of new applications as well. In such a case, since information on the exchanged data rate is not yet available, S-EDRP defines an early assignment and monitors the application execution to harvest the needed information. Then, at the first MAPE execution, S-EDRP can reassign the application by solving the updated EDRP model and neglecting reconfiguration costs (i.e., $w_d = 0$).

6.3. *Stateful Migrations in Storm*

In Storm, a task represents the smallest entity that handles a partition of the operator state. Therefore, the key idea is to introduce, within the Storm architecture, new components that perform reconfigurations by following the pause-and-resume approach. To this end, a task is enhanced with the ability to be paused and resumed, and to export and import the managed operator state. The newly introduced components, represented in red in Figure 2, are DataStore, ReconfigurationNotifier, and ReconfigurationWatcher. The *DataStore* provides a staging area for the operator state during reconfigurations. It is a key-value storage component implemented using Redis; we could not rely on ZooKeeper, because it is not designed to hold large data values. The *ReconfigurationNotifier* is a centralized component that we added on Nimbus in order to coordinate the reconfiguration actions, in such a way to apply them while preserving the operators integrity (i.e., saving and restoring their internal state) during adaptations. This component notifies tasks when a reconfiguration is going to be performed, so that they can export their state, wait the reconfiguration, and finally import again their state. On each worker node, the *ReconfigurationWatcher* handles these notifications; it is a watchdog component that pauses and resumes the execution of the application tasks and executes the migration protocol.

To interact with the newly introduced components, we also extend the Storm APIs that define code of spouts and bolts, introducing the *StatefulSpout* and *StatefulBolt* classes. These new entities should be used when a Storm topology requires elasticity. The new definition of spouts and bolts let them work either in an operational mode or in a migration mode. The *operational mode* is associated to the normal functionality of the task, whereas the *migration mode* activates the stateful migration protocol for tasks involved in a reconfiguration.

Stateful Migration Protocol. When the number of executors or the placement for an operator is changed, the ReconfigurationNotifier publishes a reconfiguration message on ZooKeeper to instruct the involved worked nodes. As a consequence, on each worker node the ReconfigurationWatcher activates the migration mode for the application tasks, which, in turn, run the stateful migration protocol. As represented in Figure 3, the migration protocol performs the following steps. First, it pauses the outgoing streams, before emitting a special *end-of-stream* tuple to the downstream operators tasks. The latter start buffering all the incoming data until the receipt of the *end-of-stream* tuple from all the upstream tasks. Second, it exports the state managed by the tasks under reconfiguration and the buffered data, saving them on the centralized DataStore. To reduce the amount of data sent over the network, the tasks relocated on the same node use a swapping area on the local file system. At this point, the tasks reach a synchronization barrier implemented on ZooKeeper. When all the tasks reach the barrier, the ReconfigurationNotifier gives the control back to S-EDRP, which can finally complete the application re-deployment using the `rebalance` API provided by Storm. As soon as the new deployment is up and running, the tasks immediately enter migration mode, retrieve the state and the buffered tuples from the DataStore (or from the swapping area), and reach a second synchronization barrier. When all the tasks arrive to the barrier, they come back to operational mode and resume processing, starting from the tuples buffered before reconfiguration, so to preserve in-order processing and guarantee at-least-once processing. Due to space limitations, we omit further details on the stateful migration, which can be found in [9].

6.4. *S-EDRP: EDRP in Storm*

To conclude this section, we present how the EDRP model has been adjusted to represent the abstractions of Storm. We first describe the representation of applications and resources and then the modeling of the stateful migration protocol developed in Storm.

DSP and Resource Model of Storm. We have to model the fact that Storm runs multiple executors to replicate an operator, and that a Storm scheduler deploys these executors on the available worker slots, considering that at most $EPS_{\max}$ executors can be co-located on the same slot. Hence, S-EDRP defines $G_{dsp} = (V_{dsp}, E_{dsp})$, with V_{dsp} as the set of operators and E_{dsp} as the set of streams exchanged among them. Since in Storm an operator is considered as a black box element, we conveniently assume that its attributes are unitary, i.e., $C_i = 1$ and $Res_i = 1, \forall i \in V_{dsp}$. By solving the replication and placement model, S-EDRP determines the number of executors for each operator $i \in V_{dsp}$, leveraging the cardinality of $\mathcal{U}$ when $x_{i,\mathcal{U}} = 1$, with $\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)$ and k_i equals to the number of tasks of i. The resource model $G_{res} = (V_{res}, E_{res})$ must take into account that a worker node $u \in V_{res}$ offers some worker slots $WS(u)$, and each worker slot can host at most $EPS_{\max}$ executors. For simplicity, S-EDRP considers the amount of available resources C_u on a worker node u to be equal to the maximum number of executors it can host, i.e., $C_u = WS(u) \times EPS_{\max}$. To enable the parallel execution of executors, C_u should be equal (or proportional) to the number of CPU cores available on u.

Modeling the Stateful Migration Protocol. Differently from the generic formulation described in Section 4.2.4, to model the downtime induced by our the stateful migration protocol we need to account that, in Storm, the entire application is restarted when a configuration is carried out. Therefore, to preserve the application integrity, even the replicas of the operators not directly involved in a reconfiguration need to save and restore the internal state on their local swapping area. This significantly impacts the application downtime during reconfiguration. For the sake of readability, we postpone the new downtime expression derivation to Appendix A.2.

7. EXPERIMENTAL RESULTS

In this section, we evaluate EDRP through two different sets of experiments. First, we analyze the elasticity capabilities of the EDRP model through numerical investigations. Then, we evaluate the elasticity mechanisms introduced in Storm by our extension and the QoS achieved by the S-EDRP scheduler. In both cases, we compare the results achieved using EDRP with a baseline solution that does not consider reconfiguration costs (it suffices to set $w_d = 0$ to ignore the downtime effects).

7.1. *Reference DSP Application*

As a test-case application, we consider a prototype application that solves the first query of *DEBS 2015 Grand Challenge* [10]. By processing data streams originated from the New York City taxis, the goal of the query is to find the top-10 most frequent routes during the last 30 minutes. The application topology is represented in Figure 4. *Data source* reads the dataset from DataStore (Redis in our prototype), and pushes the data towards *parser*, which parses them and filters out irrelevant and invalid data. Afterwards, *filterByCoordinates* forwards only the events related to a specific observation area, whose extension is about 22 500 Km^2. The operator *computeRouteID* is in charge of identifying the route covered by taxis, computing the IDs of the starting and destination cell (500 m $\times$ 500 m squares on the map). *CountByWindow* counts the route frequency in the last

Table II. Parameters of the experimental setup.

Service rate expressed in tuples per second (tps) and internal state size in bytes per operator

Operator	μ_i (tps)	State (B)	Operator	μ_i (tps)	State (B)
data source	284	82	metronome	300	328
parser	233	-	countByWindow	335	1376
filterByCoordinates	253	-	partialRank	2371	1536
computeRouteID	253	-	globalRank	3000	480

Default values used for metrics computation

Parameter	Value	Parameter	Value
C_i	1 \$/h	$r_{(u,DS)}, r_{(DS,u)}$	100 Mbps
$C_{(u,v)}$	0.02 \$/tps	$I_{C,i}$	300 KB
$t_{s,u}$	500 ms	t_{syn}	250 ms

30 minutes; the notion of time is managed by a coordinator component, called *metronome*, which pulses when the time related to the dataset events advances. The next operators, *partialRank* and *globalRank*, compute the top-10 most frequent routes using a two-step approach that computes the ranking in a distributed and parallel manner. Finally, *globalRank* publishes the top-10 updates on a message queue, implemented with RabbitMQ.

We assume that *data source* and *globalRank* are pinned operators. In the experiments, we define that each operator can be replicated at most three times (i.e., $k_i = 3$, $\forall i$), except for the pinned ones (i.e., *data source* and *globalRank*) and the *metronome*, which cannot be easily parallelized. If not otherwise specified, we use the configuration parameters reported in Table II.

7.2. Evaluation of EDRP Model

This set of experiments evaluates the behavior of the EDRP optimization model for the reference DSP application through numerical investigation. We have implemented EDRP in CPLEX© (version 12.6.3) and we have executed the experiments on an Amazon EC2 virtual machine (c4.xlarge with 4 vCPU and 7.5 GB RAM). In these experiments, G_{res} models a portion of ANSNET, a geographically distributed network where 15 computing nodes are interconnected with non-negligible network delays (whose average is 32 ms). Within this network, we assume that a logical link $(u, v) \in E_{res}$ between any two computing resources $u, v \in V_{res}$ always exists; each logic link results by the underlying physical network paths and a shortest-path routing strategy. We assume that each computing node u has $Res_u = 2$ available resources and provides a unitary $S_u = 1$ processing speed-up. Additionally, in this scenario, we assume that each operator can be replicated at most twice (i.e., $k_i = 2, \forall i \in V_{dsp}$), except for the pinned ones (i.e., *data source* and *globalRank*) and the *metronome*, which cannot be easily parallelized. EDRP estimates the response time R_i for operator i subject to the incoming load $\lambda_i/|\mathcal{U}|$ by modeling the underlying computing node as an M/M/1 queue, i.e., $R_i(\lambda_i/|\mathcal{U}|) = (\mu_i - \lambda_i/|\mathcal{U}|)^{-1}$, where μ_i is the service rate of i measured on a reference processor. The operators service rate has been obtained through preliminary experiments and it is shown in Table II. We assume that the network latency to and from DataStore is constant and set to 5 ms, and the data transfer rate between all the pairs of nodes is 100 Mbps. We consider a 1 hour experiment during which the *data source* emission rate linearly increases from 70 to 220 tuples per second, during the first half of the experiment, and then decreases down to 70 tuples per

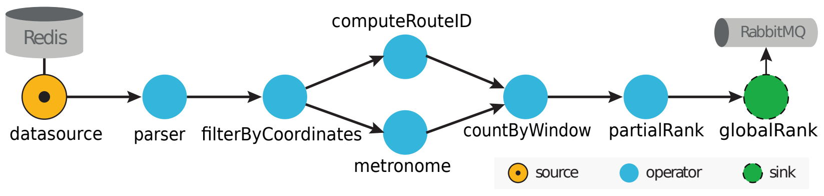


Figure 4. Reference DSP application.

second, during the second half. We solve EDRP periodically, every minute, possibly reconfiguring the application deployment. The initial placement is computed by EDRP as well, ignoring the downtime-related metrics, with a balanced objective function having $w_r = w_c = 0.5$. Considering the execution environment, we impose the following bounds on the QoS metrics: $R_{max} = 175$ ms, $C_{max} = 30$ \$/h, and $T_{D,max} = 7.5$ s.

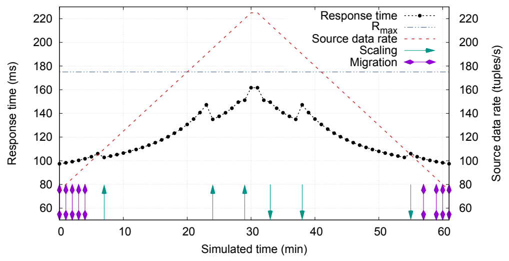
We consider different objective function configurations by changing the weights w_X, $X \in \{r, c, d\}$, which translate to different QoS optimization goals into the model. We first consider the case whereby we do not account for reconfiguration costs, i.e., $w_d = 0$, and assign equal weights to response time and cost, i.e., $w_r = w_c = 0.5$. Then, we consider a balanced scenario with $w_d = 0.4$ and $w_r = w_c = 0.3$. Finally, we consider configurations which focus on either response time or deployment cost: $w_r = 0.6$, $w_d = 0.4$, and $w_c = 0.6$, $w_d = 0.4$.

Results. We first study the DSP application behavior when we do not consider reconfiguration cost, with $w_r = w_c = 0.5$ and $w_d = 0$. In the initial deployment at $t = 0$ minutes of simulated time (min), each component gets exactly one instance. As shown in Figure 5a, as load starts growing, EDRP repeatedly migrates components to exploit better operator placement configurations. At $t = 7$ min, a second instance of *partialRank* is eventually launched. Then, at $t = 24$ min, the model scales-out at once other components: *parser*, *filterByCoordinates*, and *computeRouteID*. A third scale-out is performed when the input rate reaches the peak, and *countByWindow* is replicated as well. In the second half of the experiment, EDRP progressively comes back to the initial configuration. Table III summarizes the experiment results, reporting the mean response time and cost, and total downtime. Observe that, although the application achieves good performance, it suffers appreciably longer downtime than other scenarios (from 240% to 500% higher values).

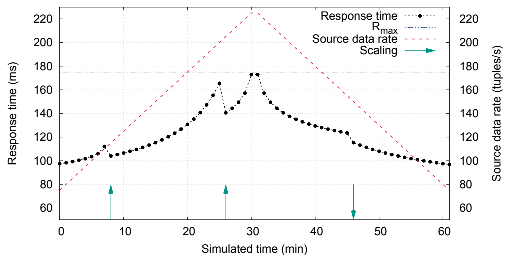
We now turn our attention to the scenarios where EDRP accounts for the reconfiguration costs. In the following experiments we set $w_d = 0.4$ and vary the other weights. Figure 5b illustrates the results for the balanced scenario with $w_r = w_c = 0.3$ and $w_d = 0.4$. Compared to the previous case, here we have only three reconfigurations overall. First, as the input rate grows, after 8 min EDRP launches a second replica for *partialRank*. This new instance is placed on the same node where *partialRank* is already running along with *countByWindow*, which minimizes the amount of state moved across the network. The latter operator is concurrently migrated to a different node. Then, at $t = 27$ min, since response time is approaching the user-defined threshold, EDRP scales-out the *parser*, *filterByCoordinates*, and *computeRouteID* operators, while also migrating others. some components. When the load decreases, EDRP is quite conservative and does not immediately scale-in operators. After 46 min, the model terminates all the replicas launched at $t = 27$ min. So all the operators are left with just one replica, except for *partialRank* because, according to the model, terminating this extra replica is not worth more downtime. We observe that the scale-in operation also results in a reduction of the application latency. This can be explained by observing that, along with the scale-in, EDRP takes advantage from co-locating communicating operators on the same node, which reduces network latency and, in turn, overall application latency.

If we focus on response time optimization, setting $w_r = 0.6$ and $w_d = 0.4$, we get results which are similar to the previous balanced objective scenario with better latency but higher cost. As shown in Figure 5c, EDRP scales-out operators in two phases in the first part, and performs a single scale-in after 50 min. Finally, we consider the deployment cost minimization, setting $w_c = 0.6$ and $w_d = 0.4$. As expected, we experience lower cost and higher response time with respect to the previous cases. As shown in Figure 5d, EDRP scales-out *partialRank* at $t = 8$ min, similarly to other settings. However, it delays other scaling operations until it is forced to use more replicas to satisfy the latency bound; EDRP progressively launches new instances from $t = 26$ min to $t = 30$ min. Nevertheless, consistently with the cost minimization objective, EDRP immediately terminates those replicas as input rate decreases. Finally, at $t = 53$ min, EDRP scales-in *partialRank*, leaving each operator with a single running instance.

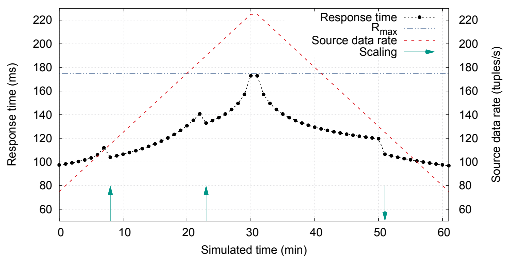
Scalability. For any given topology and number of computing nodes, the EDRP optimization problem complexity depends on the maximum degree of operator replication k_i, which impacts the number of operator configuration alternatives, that is the cardinality of the sets $\mathcal{P}(V^i_{res}; k_i)$ and thus the number of operator and link variables, $x_{i,\mathcal{U}}$ and $y_{(i,j),(\mathcal{U},\mathcal{V})}$, respectively. We computed the



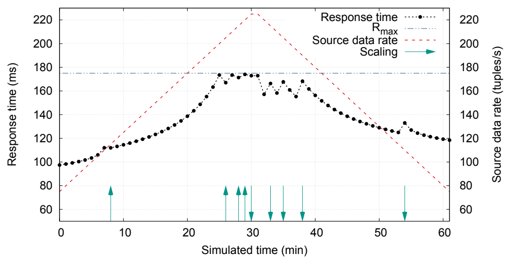
(a) Baseline ($w_r = 0.5, w_c = 0.5, w_d = 0$): minimization of response time and cost, with no reconfiguration costs.



(b) Balanced ($w_r = 0.3, w_c = 0.3, w_d = 0.4$): minimization of response time and cost, with reconfiguration costs.



(c) Response time ($w_r = 0.6, \quad w_d = 0.4, \quad w_c = 0$): minimization of response time, with reconfiguration costs.



(d) Cost ($w_c = 0.6, w_d = 0.4, w_r = 0$): minimization of deployment cost, with reconfiguration costs.

Figure 5. EDRP model: application response time under different optimization configurations.

Table III. Mean response time and cost, and total downtime with different EDRP objective configurations.

Objective	Resp. time	Cost	Downtime
Baseline ($w_d = 0, w_c = w_r = 0.5$)	120.8 ms	9.6 \$/h	12.0 s
Balanced ($w_d = 0.4, w_c = w_r = 0.3$)	121.7 ms	9.9 \$/h	2.4 s
Response time ($w_d = 0.4, w_r = 0.6$)	121.5 ms	10.3 \$/h	2.4 s
Cost ($w_d = 0.4, w_c = 0.6$)	136.0 ms	9.2 \$/h	4.9 s

number of variables and measured the EDRP resolution time for our reference DSP application for an increasing number of computing nodes and maximum replication degree k. The results are shown in Table IV (at the left of the slash sign /). We can observe that the number of variables and resolution time rapidly grow even for $k = 2$ and small number of nodes and it is not possible even to generate the entire instance for $k = 3$ but for the smallest network size. This is due to the huge number of link variables which is $O(|E_{dsp}||V_{res}|^{2k})$ and thus exponentially grows with k.

These results clearly show that EDRP can be directly applied only for small instances and limited replication degree. Therefore, in order to support runtime operation computationally efficient heuristics are required. In this respect, EDRP nevertheless represents a benchmark against which the performance of heuristics solutions can be assessed. To address this limitation, we can take advantage of the fact that, as confirmed by our experiments, most of the times an operator reconfiguration entails a single scale-out, scale-in operation or a migration. As a consequence, in practice, for each operator i we do not need to consider all possible configurations $\mathcal{V} \in \mathcal{P}(V_{res}^i; k_i)$, but only a (significantly smaller) subset. For example, given the current operator i deployment $\mathcal{U}$, we can consider, as a heuristic, the set of configurations $\mathcal{P}(V_{res}^i; k_i; \mathcal{U}_{i,0}) \subset \mathcal{P}(V_{res}^i; k_i)$ which only comprises the deployments $\mathcal{V}$ which differ from the current deployment $\mathcal{U}_{i,0}$ by only one element (one more node $u \in V_{res}^i$ for a scale-out operation, one less node $u \in \mathcal{U}_{i,0}$, $\mathcal{U}(u) > 0$ for a scale-in operation, and a node replacement $u \in \mathcal{U}_{i,0}, \mathcal{U}_{i,0}(u) > 0$ replaced by a node $v \in V_{res}^i$ for a migration). More formally, $\mathcal{P}(V_{res}^i; k_i; \mathcal{U}_{i,0}) = \{\mathcal{V} \in \mathcal{P}(V_{res}^i; k_i) \mid \wedge_{v \in V_{res}^i} (|\mathcal{U}_{i,0}(v) - \mathcal{V}(v)| \leq 1) \wedge \sum_{v \in V_{res}^i} |\mathcal{U}_{i,0}(v) - \mathcal{V}(v)| \leq 2 \wedge \sum_{v \in V_{res}^i} (\mathcal{U}_{i,o}(v) - \mathcal{V}(v)) \cdot \sum_{v \in V_{res}^i} (\mathcal{V}(v) - \mathcal{U}_{i,0}(v)) \leq 0\}$.

Table IV. Number of variables and execution time under EDRP and *r*EDRP (expressed as EDRP / *r*EDRP).

$\|V_{res}\|$	Max Replication Degree = 2		Max Replication Degree = 3		Max Replication Degree = 4	
	Variables ($\times 10^3$)	Resolution Time (s)	Variables ($\times 10^3$)	Resolution Time (s)	Variables ($\times 10^3$)	Resolution Time (s)
8	9.4 / 1.6	3.7 / 0.8	113.7 / 1.8	298.0 / 0.7	994.5 / 1.8	- / 1.0
12	37.3 / 2.8	30.5 / 0.6	848.7 / 3.3	- / 0.9	- / 3.4	- / 0.8
16	103.2 / 5.5	122.6 / 1.1	- / 6.1	- / 1.2	- / 6.3	- / 1.2
20	231.6 / 8.8	415.2 / 1.6	- / 10.0	- / 1.9	- / 9.7	- / 2.5
24	453.3 / 12.7	593.7 / 3.1	- / 13.8	- / 3.9	- / 13.9	- / 4.2

We call this approach *restricted* EDRP problem, *r*EDRP for short. As shown in Table IV, *r*EDRP (at the right of / sign) is characterized by a much smaller set of variables (it is easy to verify that the number of variables is $O(|V_{res}|^2|V_{dsp}||E_{dsp}|)$, which is only quadratic on the number of nodes) and order of magnitudes faster execution times. The results with *r*EDRP (not shown for space limitation) are very close to the optimal behavior but require a fraction of the computational costs.

7.3. Evaluation of S-EDRP Scheduler

We perform the experiments using Apache Storm 0.9.3 on a cluster of 5 worker nodes, each with 2 worker slots, and a further node to host Nimbus and ZooKeeper. Each node is a machine with a dual CPU Intel Xeon E5504 (8 cores at 2 GHz) and 16 GB of RAM. To better exploit the presence of independent CPU cores, we configure the system so that a worker slot can host at most 4 executors, i.e., $EPS_{\max} = 4$; therefore, a worker node can host at most 8 operator replicas, one for each available CPU core. In S-EDRP, the Analyze phase triggers every 60 s the Plan phase, which solves the ILP problem using CPLEX© (version 12.6.3).

As regards the model parameters, we rely on the same configuration used in the numerical investigation (see Table II) upon which we introduce the following adjustments. First, in these experiments we do not rely on a closed formula expression, e.g., the response time of an M/M/1 queue, to compute the response time as function of the input load. Instead, we conducted a preliminary set of experiments to measure the response time of each operator for different values of the input load[§]. Then, we increase the synchronization overhead t_{sync} from 250 ms to 6 s, accounting for the execution time of Storm `rebalance` command.

In these experiments we use a portion of the real dataset provided by DEBS for the 2015 Grand Challenge. We replay the taxi dataset 60 times faster than the original dataset, thus obtaining that 1 minute of the original dataset time is equal to 1 second of time in our experiments. In particular, we feed the application with data collected during 4 days, characterized by different load levels during the various parts of the day: the tuple emission rate ranges from about 20 to 300 tuples per second. We impose the following bounds on the QoS metrics: $R_{max} = 150$ ms, $C_{max} = 15$ \$/h, $T_{D,max} = 60$ s. We compare different configurations for the objective function. Similarly to the numerical investigation, we first consider the baseline case that neglects reconfiguration costs, then we consider a balanced scenario with $w_d = 0.4$, and finally we consider the configurations that focus on either response time or deployment cost. Table V provides a summary of the obtained results in the different settings, showing mean response time (computed excluding tuples buffered during reconfigurations) and monetary cost, and total application downtime during each experiment.

Figure 6a shows the results for the baseline scenario, where S-EDRP reconfigures the application at almost every scheduling round. It adjusts the parallelism of *partialRank*, which appears to be a bottleneck in our application, following the variations of the input rate. The total number of replicas ranges from 8 (i.e., one per operator) to 10. Moreover, S-EDRP frequently migrates operators when the load is high, in order to exploit more promising deployments. As consequences of this behavior, we note that (i) the application downtime lasts for 10.18% of the experiment duration, and (ii) the

[§]Our earlier experiments revealed that a M/M/1 queue approximation provide a poor approximation of the operator response time. For a more realistic response time modeling, for each operator i we measured the average response time for a set of possible inputs and then derived closed expressions for $R_i(\lambda)$ through Least Squares polynomial interpolation.

Table V. Mean response time and cost, and total downtime with different S-EDRP objective configuration.

Objective	Resp. time	Cost	Downtime
Baseline ($w_d = 0$, $w_c = w_r = 0.5$)	103.6 ms	9.3 \$/h	10.18%
Balanced ($w_d = 0.4$, $w_c = w_r = 0.3$)	90.0 ms	9.8 \$/h	2.36%
Response time ($w_d = 0.4$, $w_r = 0.6$)	88.8 ms	9.9 \$/h	1.27%
Cost ($w_d = 0.4$, $w_c = 0.6$)	116.6 ms	8.9 \$/h	4.00%

latency and the source data rate present frequent spikes, which are caused by the buffering of tuples during the reconfigurations at both the source and operators.

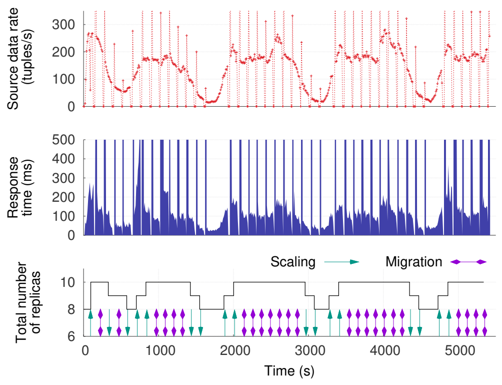
Figure 6b shows the results obtained with a balanced objective function, which equally weights response time and cost, and limits downtime (i.e., $w_r = w_c = 0.3$, $w_d = 0.4$). In this setting, the number of performed reconfigurations is significantly smaller. The balanced objective configuration allows to achieve a 75% reduction of the total application downtime. The mean response time, 90 ms, is lower with respect to the baseline scenario, whereas the deployment cost is slightly higher, amounting to 9.8 \$/h. We observe that, after 2000 s, S-EDRP finds more convenient to run with a higher number of replica, so to avoid the downtime introduced by a possible scale-in operation.

When S-EDRP focuses on response time minimization (i.e., $w_r = 0.6$, $w_d = 0.4$), the scheduler does not need to keep the number of replicas as low as possible. Nevertheless, it does not blindly scale-out operators, because a larger number of replicas would require using more nodes, thus involving a higher network delay in exchanging data streams. Indeed, S-EDRP scales-out only the bottleneck operator, *partialRank*. As shown in Figure 6c, S-EDRP achieves a lower mean latency, 88.8 ms, launching two extra replicas as input rate begins to grow. Then, other reconfigurations are driven by slight changes in the measured network latency, which make S-EDRP enact a few migrations to exploit better placement configurations. Running with a total of 10 replicas for most the time, the monetary cost of executing the topology is the highest with respect to the other scenarios (its mean value is equal to 9.9 \$/h). The cumulative application downtime is the 1.27% of the experiment duration.

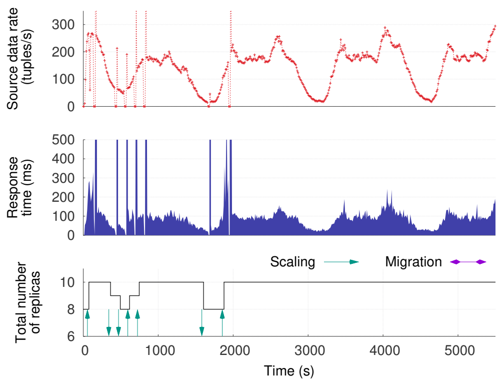
Figure 6d reports the source rate and the application latency when S-EDRP minimizes the deployment cost and downtime (i.e., $w_c = 0.6$, $w_d = 0.4$). In this setting, S-EDRP keeps the operators replication as low as possible. The number of replicas goes up to 10 when the load is high (with *partialRank* running up to 3 replicas), but, as the load significantly decreases, the scheduler scales-in the operator, restoring the initial count of 8 replicas. The mean deployment cost is indeed lower, amounting to 8.9 \$/h. The application latency is not minimized throughout the experiment, and ranges from 40 ms to 300 ms (except for the spikes soon after the reconfigurations). The cumulative application downtime is 4% of the experiment duration. Observe that, in this experiment, S-EDRP performs scale-out operations only when the bounds are violated: the violation of $R_{\max}$ and the minimization of cost lead to scale-out and subsequent scale-in operations, respectively.

8. CONCLUSIONS

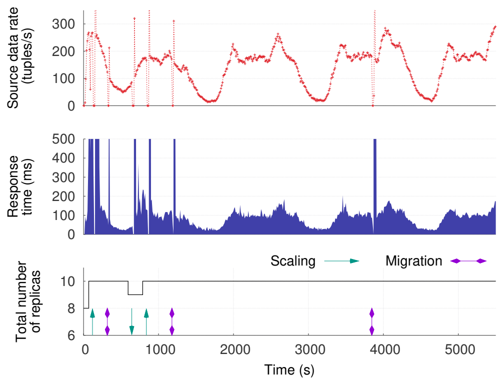
In this paper, we have presented and evaluated EDRP, an ILP formulation that jointly optimizes the replication and placement of DSP applications running in geo-distributed environments. At run-time, by monitoring the application deployment, EDRP can identify the optimal reconfiguration strategy and evaluate its convenience in terms of application downtime. Our formulation of EDRP is general and flexible, thus it can be easily extended or customized for considering different needs. In this paper, we have shown how it can be adapted and integrated in Apache Storm, one of the most used open-source DSP frameworks. Then, relying on an application that processes real-time data generated by taxis moving in a urban environment, we have conducted a thorough experimental evaluation. The latter has shown the benefits on the application performance that steam from a joint optimization of operators replication and placement and the detailed modeling of the reconfiguration costs. In particular, our results show the importance of taking into account reconfiguration cost to the overall application performance. In the considered scenario, which can be regarded as a best case scenario as far as reconfiguration is concerned, being executed on a local cluster, the



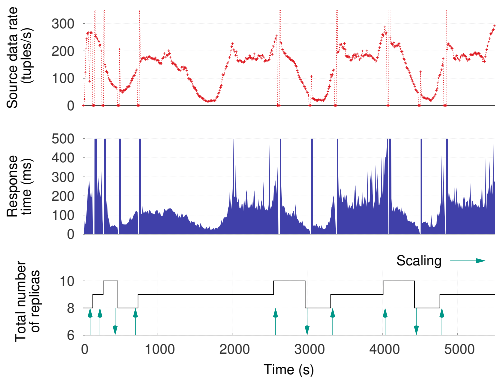
(a) Baseline ($w_r = 0.5$, $w_c = 0.5$, $w_d = 0$): minimization of response time and cost, with no reconfiguration costs.



(b) Balanced ($w_r = 0.3$, $w_c = 0.3$, $w_d = 0.4$): minimization of response time and cost, with reconfiguration costs.



(c) Response Time ($w_r = 0.6$, $w_d = 0.4$, $w_c = 0$): minimization of response time, with reconfiguration costs.



(d) Cost ($w_c = 0.6$, $w_d = 0.4$, $w_r = 0$): minimization of deployment cost, with reconfiguration costs.

Figure 6. S-EDRP and different optimization configurations: evolution of the application response time and operators replication under a varying source data rate.

reconfiguration downtime were as large as 10% of the overall experiment duration, and with EDRP we were able to achieve up to a tenfold reduction of the system downtime. Moreover, by adjusting the metric weights, we were able to trade-off the different performance metrics stressing latency and/or resource utilization. Since reconfiguration costs are expected to be significantly higher in a distributed scenario due to non negligible network delays and limited bandwidth, these results clearly show that importance of taking into account reconfiguration costs in the operation of self-adaptive DSP platforms.

As future work, we plan to develop solutions that address the critical points of runtime deployment strategies: the scalability and the identification of efficient reconfigurations. The investigated EDRP problem is NP-hard, therefore we need efficient heuristics to deal with large problem instances. By identifying the optimal deployment and run-time reconfiguration, EDRP can provide useful information for designing new heuristics that, not only reduce the resolution time, but also identify solutions as close as possible to the optimal one. To further increase the scalability of the system components in charge of computing the deployment, we also plan to investigate how the application placement and replication problem can be efficiently decentralized. This need rises from the emerging edge computing environment, where a plethora of IoT devices usually operate within a limited geographic area. Regarding the identification of efficient reconfigurations, we observe that the approach proposed in this paper determines reconfigurations relying on a single snapshot of the system, i.e., the system status when the optimization model is executed. Such an approach inherently computes greedy reconfiguration strategies which might be sub-optimal if the overall performance over a period of time is of interest. To this end, we will consider optimization approaches which explicitly take into account intervals of time. This will include model based optimization, *e.g.* Markov Decision Processes and Model Predictive Control if the system dynamic is known (or estimated) and model free optimization when the model is not available, *e.g.*, Reinforcement Learning. Another direction along which we plan to continue our work regards how to deal with percentiles of QoS attributes, *e.g.*, generalizing the approach we adopted in the field of composite services [45] to Data Stream Processing.

REFERENCES

1. Hirzel M, Soulé R, Schneider S, Gedik B, Grimm R. A catalog of stream processing optimizations. *ACM Comput. Surv.* Mar 2014; **46**(4).
2. Gedik B, Schneider S, Hirzel M, Wu KL. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.* 2014; **25**(6):1447–1463.
3. Heinze T, Roediger L, Meister A, Ji Y, *et al.*. Online parameter optimization for elastic data stream processing. *Proc. ACM SoCC '15*, 2015; 276–287.
4. Shi W, Cao J, Zhang Q, Li Y, Xu L. Edge computing: Vision and challenges. *IEEE Internet of Things J.* 2016; **3**(5):637–646.
5. Eidenbenz R, Locher T. Task allocation for distributed stream processing. *Proc. IEEE INFOCOM '16*, 2016.
6. Lohrmann B, Janacik P, Kao O. Elastic stream processing with latency guarantees. *Proc. IEEE ICDCS '15*, 2015; 399–410.
7. Mencagli G. A game-theoretic approach for elastic distributed data stream processing. *ACM Trans. on Autonomous and Adaptive Systems* 2016; **11**(2).
8. Thoma C, Labrinidis A, Lee A. Automated operator placement in distributed data stream management systems subject to user constraints. *Proc. IEEE ICDEW '14*, 2014; 310–316.
9. Cardellini V, Nardelli M, Luzi D. Elastic stateful stream processing in Storm. *Proc. HPCS '16*, 2016; 583–590.
10. Jerzak Z, Ziekow H. The DEBS 2015 grand challenge. *Proc. ACM DEBS '15*, ACM, 2015; 266–268.
11. Lakshmanan GT, Li Y, Strom R. Placement strategies for Internet-scale data stream systems. *IEEE Internet Computing* 2008; **12**(6):50–60.
12. Cardellini V, Grassi V, Lo Presti F, Nardelli M. Optimal operator placement for distributed stream processing applications. *Proc. ACM DEBS '16*, 2016; 69–80.
13. Aniello L, Baldoni R, Querzoni L. Adaptive online scheduling in Storm. *Proc. ACM DEBS '13*, 2013; 207–218.
14. Pietzuch P, Ledlie J, Shneidman J, Roussopoulos M, *et al.*. Network-aware operator placement for stream-processing systems. *Proc. IEEE ICDE '06*, 2006.
15. Xu J, Chen Z, Tang J, Su S. T-Storm: traffic-aware online scheduling in Storm. *Proc. IEEE ICDCS '14*, 2014; 535–544.
16. Toshniwal A, Taneja S, Shukla A, Ramasamy K, *et al.*. Storm@Twitter. *Proc. ACM SIGMOD '14*, 2014; 147–156.
17. Cardellini V, Grassi V, Lo Presti F, Nardelli M. Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Perform. Eval. Rev.* May 2017; **44**(4):11–22.
18. Kephart J, Chess D. The vision of autonomic computing. *IEEE Computer* Jan 2003; **36**(1):41–50.
19. Fernandez RC, Migliavacca M, Kalyvianaki E, Pietzuch P. Integrating scale out and fault tolerance in stream processing using operator state management. *Proc. ACM SIGMOD '13*, 2013.
20. Gulisano V, Jiménez-Peris R, Patino-Martínez M, Soriente C, Valduriez P. StreamCloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.* 2012; **23**(12):2351–2365.
21. Heinze T, Pappalardo V, Jerzak Z, Fetzer C. Auto-scaling techniques for elastic data stream processing. *Proc. IEEE ICDEW '14*, 2014; 296–302.
22. Xu L, Peng B, Gupta I. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. *Proc. IEEE IC2E '16*, 2016; 22–31.
23. Heinze T, Jerzak Z, Hackenbroich G, Fetzer C. Latency-aware elastic scaling for distributed data stream processing systems. *Proc. ACM DEBS '14*, 2014; 13–22.
24. Madsen KGS, Zhou Y, Cao J. Integrative dynamic reconfiguration in a parallel stream processing engine. *Proc. IEEE ICDE '17*, 2017; 227–230.
25. De Matteis T, Mencagli G. Elastic scaling for distributed latency-sensitive data stream operators. *Proc. PDP '17*, 2017; 61–68.
26. Sajjad HP, Danniswara K, Al-Shishtawy A, Vlassov V. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. *Proc. 2016 IEEE/ACM Symp. on Edge Computing*, 2016; 168–178.
27. Saurez E, Hong K, Lillethun D, Ramachandran U, *et al.*. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. *Proc. ACM DEBS '16*, 2016; 258–269.
28. To Q, Soto J, Markl V. A survey of state management in big data processing systems. *CoRR* 2017; **abs/1702.01596**. URL `http://arxiv.org/abs/1702.01596`.
29. Heinze T, Aniello L, Querzoni L, Jerzak Z. Cloud-based data stream processing. *Proc. ACM DEBS '14*, 2014; 238–245.
30. Madsen KGS, Zhou Y, Su L. Enorm: Efficient window-based computation in large-scale distributed stream processing systems. *Proc. ACM DEBS '16*, 2016; 37–48.
31. Madsen KGS, Zhou Y. Dynamic resource management in a massively parallel stream processing engine. *Proc. ACM CIKM '15*, 2015; 13–22.

32. Wu Y, Tan KL. ChronoStream: Elastic stateful stream computation in the cloud. *Proc. IEEE ICDE '15*, 2015; 723–734.
33. Nasir MAU, Morales GDF, Garca-Soriano D, Kourtellis N, Serafini M. The power of both choices: Practical load balancing for distributed stream processing engines. *Proc. IEEE ICDE '15*, 2015; 137–148.
34. Trident API overview 2015. URL `http://storm.apache.org/releases/1.1.0/Trident-API-Overview.html`.
35. Cardellini V, Grassi V, Lo Presti F, Nardelli M. Distributed QoS-aware scheduling in Storm. *Proc. ACM DEBS '15*, 2015; 344–347.
36. Li J, Pu C, Chen Y, Gmach D, Milojicic D. Enabling elastic stream processing in shared clusters. *Proc. IEEE CLOUD '16*, 2016; 108–115.
37. Kulkarni S, Bhagat N, Fu M, Kedigehalli V, *et al.*. Twitter Heron: Stream processing at scale. *Proc. ACM SIGMOD '15*, 2015; 239–250.
38. Floratou A, Agrawal A, Graham B, Rao S, Ramasamy K. Dhalion: self-regulating stream processing in Heron. *Proc. VLDB '17*, 2017.
39. Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized streams: Fault-tolerant streaming computation at scale. *Proc. ACM SOSP'13*, 2013; 423–438.
40. Carbone P, Katsifodimos A, Ewen S, Markl V, *et al.*. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 2015; **36**(4).
41. Peng B, Hosseini M, Hong Z, Farivar R, Campbell R. R-Storm: Resource-aware scheduling in Storm. *Proc. ACM/IFIP/USENIX Middleware '15*, 2015; 149–161.
42. Yoon KP, Hwang CL. *Multiple Attribute Decision Making: an Introduction*. Sage Pubs, 1995.
43. Weyns D, Schmerl B, Grassi V, Malek S, *et al.*. On patterns for decentralized control in self-adaptive systems. *Software Engineering for Self-Adaptive Systems II*, *LNCS*, vol. 7475. Springer, 2013; 76–107.
44. Dabek F, Cox R, Kaashoek F, Morris R. Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.* 2004; **34**(4):15–26.
45. Cardellini V, Casalicchio E, Grassi V, Lo Presti F. Adaptive management of composite services under percentile-based service level agreements. *Proc. ICSOC '10*, *LNCS*, vol. 6470, Springer, 2010; 381–395.

A. RECONFIGURATION METRICS

In this appendix we present the detailed expressions of the reconfiguration cost metrics. In Section A.1, we derive the expressions for the reconfiguration downtime metric under the *pause-and-resume* approach. Then, in Section A.2, we provide the detailed downtime expressions for our implementation of the stateful migration protocol in Storm.

A.1. *Reconfiguration Downtime*

As shown in Section 4.2.4, the reconfiguration downtime of the operator $i \in V_{dsp}$ is defined as:

$$T_{D,i}(\boldsymbol{x}) = \sum_{\mathcal{V} \in \mathcal{P}(V^i_{res}; k_i)} t_D\left(i, \mathcal{U}_{0,i}, \mathcal{V}\right) x_{i,\mathcal{V}} \tag{24}$$

where $t_D(i, \mathcal{U}, \mathcal{V})$ represents the time needed to reconfigure the operator i from the deployment $\mathcal{U}$ to $\mathcal{V}$, and $\mathcal{U}_{0,i}$ represents the current deployment of operator i. The expression for $t_D(i, \mathcal{U}, \mathcal{V})$ depends on the actual type of reconfiguration to be enacted when the deployment of i changes from $\mathcal{U}$ to $\mathcal{V}$:

$$t_D(i, \mathcal{U}, \mathcal{V}) = \begin{cases} 0 & \text{if } \mathcal{U} = \mathcal{V} \\ t_{syn} + t_{D,MI}(i, \mathcal{U}, \mathcal{V}) & \text{if } |\mathcal{U}| = |\mathcal{V}|, \mathcal{U} \neq \mathcal{V} \\ t_{syn} + t_{D,SO}(i, \mathcal{U}, \mathcal{V}) & \text{if } |\mathcal{U}| > |\mathcal{V}| \\ t_{syn} + t_{D,SI}(i, \mathcal{U}, \mathcal{V}) & \text{if } |\mathcal{U}| < |\mathcal{V}| \end{cases} \tag{25}$$

where t_{syn} is a constant synchronization overhead, $t_{D,X}(i, \mathcal{U}, \mathcal{V})$, with $X \in \{MI, SO, SI\}$, represents the time needed to perform a migration, a scale-out, and a scale-in operation on i, respectively. Since each type of reconfiguration requires data transfers between multiple pairs of nodes, which can be concurrently executed, the time to complete the reconfiguration corresponds to the longest of any such operation, that is:

$$t_{D,X}(i, \mathcal{U}, \mathcal{V}) = \max_{\substack{u \in \mathcal{U}, v \in \mathcal{V} \\ u \neq v}} \left\{\tau_{D,X}(i, u, v, \mathcal{U}, \mathcal{V})\right\} \tag{26}$$

where $\tau_{D,X}(i, u, v, \mathcal{U}, \mathcal{V})$, $X \in \{MI, SO, SI\}$ represents the time required to exchange data between node u to node v when the operator i is migrated, scaled out, or scaled in, respectively, during the reconfiguration from deployment $\mathcal{U}$ to deployment $\mathcal{V}$. $\tau_{D,X}\left(i,u,v,\mathcal{U},\mathcal{V}\right)$ has the following general expression:

$$\tau_{D,X}\left(i,u,v,\mathcal{U},\mathcal{V}\right) = \max\left\{\tau_{C}^{dwl}(i,v)\cdot \mathbb{1}_{\{\mathcal{U}(v)=0\}}, \tau_{S,X}^{upl}(i,u,v,\mathcal{U},\mathcal{V})\right\} + \\ + \tau_{S,X}^{dwn}(i,u,v,\mathcal{U},\mathcal{V}) + t_{s,v}\cdot \mathbb{1}_{\{\mathcal{V}(v)>\mathcal{U}(v)\}} \quad (27)$$

where $\mathbb{1}_{\{.\}}$ is the indicator function. The expression of $\tau_{D,X}\left(i,u,v,\mathcal{U},\mathcal{V}\right)$ accounts for: 1) $\tau_{C}^{dwl}(i,v)$, the time to download time the operator code to node v, if there was not a replica of operator i in v in the (current) deployment $\mathcal{U}$, i.e., if $\mathcal{U}(v)=0$; 2) $\tau_{S,X}^{upl}(i,u,v,\mathcal{U},\mathcal{V})$, the time to upload the internal state of the replicas present in u to the DataStore; 3) $\tau_{S,X}^{dwn}(i,u,v,\mathcal{U},\mathcal{V})$, the time to download the internal state of replicas from the DataStore to node v; and 4) $t_{s,v}$, the time to (concurrently) initialize new replicas in v, if any, i.e., if $\mathcal{V}(v)>\mathcal{U}(v)$.

The time needed to transfer the operator code, i.e., $t_{C}^{dwl}(i,v)$, and the time needed to redistribute the replicas state, i.e., $t_{S,X}^{upl}(i,u,v,\mathcal{U},\mathcal{V})$ and $t_{S,X}^{dwn}(i,u,v,\mathcal{U},\mathcal{V})$, are function of: 1) the round-trip network delay between the DataStore and the related computing resource, $d_{x,DS}+d_{DS,x}$, with $x\in\{u,v\}$; 2) the amount of data to transfer data from or to the DataStore, that is, the code $I_{C,i}$ to be downloaded to node v, the state $I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{upl}$ to be uploaded from u to the DataStore and the state $I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{dwn}$ to be then downloaded to v, respectively; and 3) the data rate $r_{(x,DS)}$ and $r_{(DS,x)}$, with $x\in\{u,v\}$, to and from the DataStore. We readily get the following expressions:

$$\tau_{C}^{dwn}(i,v) = d_{(v,DS)} + \frac{I_{C,i}}{r_{(DS,v)}} + d_{(DS,v)} \quad (28)$$

$$\tau_{S,X}^{upl}(i,u,v,\mathcal{U},\mathcal{V}) = d_{(u,DS)} + \frac{I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{upl}}{r_{(u,DS)}} + d_{(DS,u)} \quad (29)$$

$$\tau_{S,X}^{dwn}(i,u,v,\mathcal{U},\mathcal{V}) = d_{(v,DS)} + \frac{I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{dwn}}{r_{(DS,v)}} + d_{(DS,v)} \quad (30)$$

We conclude by deriving the expressions for the amount of state that has to be redistributed, which depends on the specific reconfiguration action. When a migration is performed, i.e., $X = MI$, a replica first stores and then retrieves its portion of the state on the DataStore during the reconfiguration. A scaling operation redistributes the operator state among its replicas. Specifically, when a scale-out is performed, i.e., $X = SO$, each replica provides a uniform fraction of its internal state to the new ones; similarly, when a scale-in is performed, i.e., $X = SI$, the internal state of the terminated replicas is redistributed among the other operator replicas. Let $I_{S,i}$ denote operator i state size. Under the assumption that the state is uniformly redistributed among replicas, we have the following expressions:

$$I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{upl} = \begin{cases} I_{S,i}\left[\frac{\delta_u^-(\mathcal{U},\mathcal{V})}{|\mathcal{U}|} + \mathcal{U}(u)\frac{|\mathcal{V}|-|\mathcal{U}|}{|\mathcal{U}||\mathcal{V}|}\right] & \text{if X = SO} \\ I_{S,i}\frac{\delta_u^-(\mathcal{U},\mathcal{V})}{|\mathcal{U}|} & \text{otherwise} \end{cases} \quad (31)$$

$$I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{dwn} = \begin{cases} I_{S,i}\left[\frac{\delta_v^+(\mathcal{U},\mathcal{V})}{|\mathcal{V}|} + \min\left\{\mathcal{U}(v),\mathcal{V}(v)\right\}\frac{|\mathcal{U}|-|\mathcal{V}|}{|\mathcal{U}||\mathcal{V}|}\right] & \text{if X = SI} \\ I_{S,i}\frac{\delta_v^+(\mathcal{U},\mathcal{V})}{|\mathcal{V}|} & \text{otherwise} \end{cases} \quad (32)$$

where $\delta_u^+\left(\mathcal{U},\mathcal{V}\right)$ (and $\delta_u^-\left(\mathcal{U},\mathcal{V}\right)$) denotes the number of replicas to be added (and removed) in node $u\in V_{res}$ when the configuration changes from $\mathcal{U}$ to the deployment $\mathcal{V}$, that is: $\delta_u^+\left(\mathcal{U},\mathcal{V}\right) = \max\{\mathcal{V}\left(u\right)-\mathcal{U}\left(u\right),0\}$ and $\delta_u^-\left(\mathcal{U},\mathcal{V}\right) = \max\{\mathcal{U}\left(u\right)-\mathcal{V}\left(u\right),0\}$, respectively. In (31)-(32), $\frac{\delta_u^-(\mathcal{U},\mathcal{V})}{|\mathcal{U}|}$ is the percentage of state uploaded from the node u to the DataStore during a migration or scale in operation, which is function of the number of terminated replicas $\delta_u^-\left(\mathcal{U},\mathcal{V}\right)$. Similarly, $\frac{\delta_v^+(\mathcal{U},\mathcal{V})}{|\mathcal{V}|}$ is the percentage of new state transfered to v from the DataStore, going from deployment $\mathcal{U}$ to $\mathcal{V}$. The terms proportional to $\frac{|\mathcal{U}|-|\mathcal{V}|}{|\mathcal{U}||\mathcal{V}|}$ represent the amount of state that is redistributed among the other operator replicas after a scaling operation.

A.2. *Stateful Migration Downtime in Storm*

As mentioned in Section 6.4, in Storm we need to take into account that, for any reconfiguration, the entire application must be restarted. In this case we can rewrite (24) as:

$$T_{D,i}(\boldsymbol{x}) = \max \left\{ \sum_{\mathcal{V} \in \mathcal{P}(V^i_{res};k_i)} t_D\left(i, \mathcal{U}_{0,i}, \mathcal{V}\right) x_{i,\mathcal{V}} \, , t_{D,R}(i, \mathcal{U}_{0,i}) \cdot \mathbb{1}_{\{\sum_i \sum_{\mathcal{V} \neq \mathcal{U}_{0,i}} x_{i,\mathcal{V}} > 0\}} \right\} \tag{33}$$

where the first term is the time needed to reconfigure the operator i from the deployment $\mathcal{U}$ to $\mathcal{V}$, $\mathcal{U}_{0,i}$ is the current deployment of i, whereas $t_{D,R}(i, \mathcal{U})$ is the time needed to restart i, should any configuration occurs in the application (condition expressed by $\sum_i \sum_{\mathcal{V} \neq \mathcal{U}_{0,i}} x_{i,\mathcal{V}} > 0$).

Since our Storm migration protocol does not distinguish the different reconfiguration actions (i.e., migration or scaling), we have a single expression for $t_D(i, \mathcal{U}, \mathcal{V})$, i.e., the time to reconfigure the operator i from $\mathcal{U}$ to $\mathcal{V}$. It needs to account for a constant synchronization overhead, t_{syn}, and the longest downtime over any possible replica reconfiguration. Thus, we have:

$$t_D(i, \mathcal{U}, \mathcal{V}) = t_{syn} + \max_{\substack{u \in \mathcal{U}, v \in \mathcal{V} \\ u \neq v}} \{\tau_D(i, u, v, \mathcal{U}, \mathcal{V})\} \quad \mathcal{U} \neq \mathcal{V} \tag{34}$$

where

$$\tau_D(i, u, v, \mathcal{U}, \mathcal{V}) = \tau_C^{dwl}(i, v) \mathbb{1}_{\{v \notin \bigcup_i \mathcal{U}_{0,i}\}} + \tau_S^{upl}(i, u, v, \mathcal{U}, \mathcal{V}) + \tau_S^{dwn}(i, u, v, \mathcal{U}, \mathcal{V}) + \mathcal{V}(v) t_{s,v}, \tag{35}$$

and $t_D(i, \mathcal{U}, \mathcal{V}) = 0$ if $\mathcal{U} = \mathcal{V}$. Comparing the expression above to (27), we note that: (i) we need to consider the sum of the different terms in place of the maximum, because the pause-and-resume approach serializes the different phases; (ii) the time to download the operator code on v, $\tau_C^{dwl}(i, v)$, needs to be considered only if no replica of *any* operator is currently deployed on that node, because Storm packages the code of the whole application in a single archive, which is transferred to each worker node the first time it has to execute a component of the topology; (iii) the whole application is restarted after a reconfiguration, thus all the $\mathcal{V}(v)$ replicas deployed on v have to be launched at the end of the migration process, as expressed by the last term. We also obtain slightly different expressions for the time spent moving state and code as well:

$$\tau_C^{dwn}(i, v) = d_{(v,DS)} + \frac{\sum_{i \in V_{dsp}} I_{C,i}}{r_{(DS,v)}} + d_{(DS,v)} \tag{36}$$

$$\tau_S^{upl}(i, u, v, \mathcal{U}, \mathcal{V}) = d_{(u,DS)} + \frac{I_{S,i} \frac{\mathcal{U}(u)}{|\mathcal{U}|}}{r_{(u,DS)}} + d_{(DS,u)} \tag{37}$$

$$\tau_S^{dwn}(i, u, v, \mathcal{U}, \mathcal{V}) = d_{(v,DS)} + \frac{I_{S,i} \frac{\mathcal{V}(v)}{|\mathcal{V}|}}{r_{(DS,v)}} + d_{(DS,v)} \tag{38}$$

where we model that, differently from the general formulation (28)–(30), in Storm: (i) a node has to download the whole topology code; and (ii) each node has to save (and restore) the internal state for all the hosted replicas, because the entire application is restarted during a reconfiguration.

Finally, we consider $t_{D,R}(i, \mathcal{U})$, the time needed to restart i on the same location. This term accounts for a constant synchronization overhead, t_{syn}, and the restart time of any replica. We have:

$$t_{D,R}(i, \mathcal{U}) = t_{syn} + \max_{u \in \mathcal{U}} \{\tau_{D,R}(i, u, \mathcal{U})\} \tag{39}$$

where

$$\tau_{D,R}(i, u, \mathcal{U}) = \frac{I_{S,i} \frac{\mathcal{U}(u)}{|\mathcal{U}|}}{r_{(u,LS)}} + \frac{I_{S,i} \frac{\mathcal{U}(u)}{|\mathcal{U}|}}{r_{(LS,u)}} + \mathcal{U}(u) t_{s,v} \tag{40}$$

models the time to store and retrieve the state from the swapping area and restart the replicas of i.