Optimal Operator Replication and Placement for Distributed Stream Processing Systems

Valeria Cardellini
cardellini@ing.uniroma2.it

Vincenzo Grassi
vincenzo.grassi@uniroma2.it

Francesco Lo Presti
lopresti@info.uniroma2.it

Matteo Nardelli
nardelli@ing.uniroma2.it

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

ABSTRACT

Exploiting on-the-fly computation, Data Stream Processing (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. To keep up with the high volume of daily produced data, the operators that compose a DSP application can be replicated and placed on multiple, possibly distributed, computing nodes, so to process the incoming data flow in parallel. Moreover, to better exploit the abundance of diffused computational resources (e.g., Fog computing), recent trends investigate the possibility of decentralizing the DSP application placement.

In this paper, we present and evaluate a general formulation of the optimal DSP replication and placement (ODRP) as an integer linear programming problem, which takes into account the heterogeneity of application requirements and infrastructural resources. We integrate ODRP as prototype scheduler in the Apache Storm DSP framework. By leveraging on the DEBS 2015 Grand Challenge as benchmark application, we show the benefits of a joint optimization of operator replication and placement and how ODRP can optimize different QoS metrics, namely response time, inter-node traffic, cost, availability, and a combination thereof.

1. INTRODUCTION

The ever increasing diffusion of cheap sensors and the presence of an almost ubiquitous Internet connectivity is producing an exponential growth in the amount of daily produced data, which carry valuable information about our surrounding environment. Indeed, by extracting valuable information from noisy Big Data, it is possible to develop new intelligent and pervasive services that can improve our everyday life in several domains, e.g., healthcare, energy management, logistic, and transportation. The volume and velocity of daily produced data have fostered the development of new processing approaches that rely on the usage of multiple computing machines. Nowadays, two opposite approaches are commonly adopted: batch processing and stream processing. The former stores all the data, usually on a distributed file system, and then operates on them on the basis of different programming models, among which the well-known MapReduce [8]. The latter processes all the data on-the-fly, i.e., without storing them, so it can produce results in a near real-time fashion. Therefore, Data Stream Processing (DSP) applications are widely used to process unbounded streams of data and timely extract valuable information. We focus on this kind of applications.

A DSP application is represented as a directed acyclic graph (DAG), with data sources, operators, and final consumers as vertices, and streams as edges. Each *operator* can be seen as a black-box processing element that continuously receives incoming streams, applies a transformation, and generates new outgoing streams. In the Big Data era, DSP applications should be capable to seamlessly process huge amount of data, which require to scale their execution on multiple computing nodes, because a single machine cannot provide enough processing power. To this end, these applications usually exploit *data parallelism*, which consists in increasing or decreasing the number of parallel instances for the operators, so that each instance can process a subset of the incoming data flow in parallel (e.g., [12, 16]). Moreover, since data sources can be geographically distributed, the execution of DSP applications can also take advantage of the ever increasing presence of distributed Cloud and Fog computing resources, which can improve the system scalability and reduce latency by moving the computation towards the network edge, closer to data sources. Nevertheless, the use of such distributed infrastructure poses new challenges, that include network and system heterogeneity, geographic distribution as well as non-negligible network latencies [23].

This paper focuses on the deployment of DSP applications over distributed computing nodes. Specifically, we investigate and evaluate Optimal DSP Replication and Placement (**ODRP**) [3], a unified general formulation of the operator replication and placement problem. Differently from most works in literature [9, 20, 21, 24], ODRP can jointly determine the application placement and the replication of its operators, while optimizing the Quality of Service (QoS) attributes of the application. With respect to our previous work [3], in this paper we describe the integration of ODRP as prototype scheduler in Apache Storm, an open-source and widely used DSP framework. Moreover, we extensively evaluate the prototyped solution in a real setting with the aim of highlighting its strengths and drawbacks. To this end, we

have designed and implemented a benchmark DSP application that solves the DEBS 2015 Grand Challenge [17].

This paper is organized as follows. We review related work in Section 2; in Section 3 we describe the system model and the problem under investigation, before presenting ODRP and its integration, as prototype scheduler, in Apache Storm in Sections 4 and 5, respectively. Then, in Section 6, relying on the Storm-based prototype and the benchmark application that addresses the DEBS 2015 Grand Challenge, we show the benefits of a joint optimization of replication and placement and how ODRP can optimize different QoS metrics. Finally, we conclude in Section 7.

2. RELATED WORK

To deploy a DSP application, a DSP system needs to determine the replication degree of the application operators and their placement on the computing infrastructure. Even though these problems have been widely investigated separately, only few works study their joint optimization.

Placement and Replication Problem. Most works in literature consider DSP operator placement and operator replication as independent and orthogonal decisions, where the operator 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 take the new application configuration into account and may incur in a significant overhead. In this paper, we propose a *single-stage* approach to determine both the placement and the parallelism degree of the operators in a DSP application. The DSP placement problem has been widely investigated in literature under different modeling assumptions and optimization goals, e.g., [4, 9, 24]. In [4], we proposed a general formulation of the optimal DSP placement which takes into account the heterogeneity of computing and networking resources and which encompasses the different solutions proposed in the literature. In [3] we extended that formulation so as to determine the optimal number of replicas for each operator contextually to their placement on the underlying infrastructure; in this paper, we integrate such formulation in a Storm-based prototype and evaluate it using a real application.

Since the placement problem is NP-hard, several heuristics have been proposed, e.g. [1, 22, 26]. They aim at minimizing a diversity of utility functions, such as the DSP application end-to-end latency, the inter-node traffic, and the network usage. Our problem formulation can be adjusted to take into account these different utility functions.

Many research efforts have focused on scaling the amount of operator replicas in response to changes observed in some monitored performance metric. Some works, e.g., [5, 15], exploited threshold-based policies based on the utilization of either the system nodes or the operator instances. Other works, e.g., [12, 20, 21], used more complex policies to determine the scaling decisions. Lohrmann et al. [20] proposed a strategy that enforces latency constraints by relying on a predictive latency model based on queueing theory. Mencagli [21] presented a game-theoretic approach where the control logic is distributed on each operator.

An important issue related to operator replication regards the management of stateful operators, since their state must be migrated in order to preserve the application integrity [12]. The approach we propose in this paper jointly places and replicates operators, thus saving the overhead and latency penalty incurred by stateful operator migrations in case of disjointed replication and placement decisions.

The works most closely related to ours have been presented by Heinze et al. [14, 16]. They proposed a model to estimate the latency spike created by a set of operator movements and used it to define an operator placement algorithm based on a bin packing heuristic that minimizes the latency violations and focuses only on the placement of the newly added operators. We present an optimal problem formulation that targets the initial placement decision and can be used to benchmark existing heuristics.

While most works, including ours, focus on replicating the operators at the level of the application logic, thus changing the graph topology, Fu et al. [11] proposed a queueing theory approach to determine the number of computing resources on which each operator is placed; however, their approach does not consider network delays.

DSP Frameworks. A great variety of DSP frameworks has been proposed so far; being interested in integrating our ODRP model, we focus mainly on the open-source ones. Apache Storm [25] is a DSP framework that provides an abstraction layer to execute event-based applications. It allows the user to merely focus on the application logic, while the effort of placing, distributing, and executing the application is handled by the framework. Several research efforts have used Storm to either evaluate new operator placement algorithms in a real environment or to propose some architectural improvements (e.g., [1, 13, 19, 25, 26]). Developed as the successor of Storm, Heron [18] preserves its abstraction layer while introducing some architectural improvements. Apache Spark [28] is a general-purpose framework for large-scale processing, which provides a batch and micro-batch processing approach. This latter alternative is throughput-oriented, whereas Storm, which is a pure DSP system, can further minimize the application latency and thus can be preferred in latency sensitive scenarios. Another emerging framework is Apache Flink[1], which provides a unified solution for batch and stream processing.

Aside the specific functionalities, these open-source frameworks use directed graphs to model DSP applications; therefore, our ODRP formulation well represents their placement problem and can be integrated into their scheduler. As regards the operator replication, so far these frameworks leave completely to the user the definition of the number of replicas that have to be instantiated. Nevertheless, since the user might over-/under-estimate the incoming load, this approach can lead to a sub-optimal utilization of the available resources (i.e., over-/under-provisioning). Since several placement policies [1, 2, 6, 10, 22, 26] and seminal replication policies [5] have been already integrated into Storm, we have also chosen it to implement our ODRP scheduler.

3. SYSTEM MODEL AND PROBLEM STATEMENT

In this section we present the resource and DSP application model and define the operator replication and place-

[1] `https://flink.apache.org/`

Table 1: Main notation adopted in the paper

Symbol	Description
G_{dsp}	Graph representing the DSP application
V_{dsp}	Set of vertices (operators) of G_{dsp}
E_{dsp}	Set of edges (streams) of G_{dsp}
C_i	Cost of deploying operator $i \in V_{dsp}$
R_i	Latency of $i \in V_{dsp}$ on a reference processor
Res_i	Resources required to execute $i \in V_{dsp}$
k_i	Maximum replication degree of $i \in V_{dsp}$
$\lambda_{(i,j)}$	Average tuple rate exchanged on $(i,j) \in E_{dsp}$
$b_{(i,j)}$	Avg. number of byte per tuple on $(i,j) \in E_{dsp}$
G_{res}	Graph representing computing and network resources
V_{res}	Set of vertices (computing nodes) of G_{res}
E_{res}	Set of edges (logical links) of G_{res}
A_u	Availability of node $u \in V_{res}$
Res_u	Amount of resources available at $u \in V_{res}$
S_u	Processing speed-up of $u \in V_{res}$
$A_{(u,v)}$	Availability of $(u,v) \in E_{res}$
$C_{(u,v)}$	Transmission cost per data on $(u,v) \in E_{res}$
$d_{(u,v)}$	Network delay on $(u,v) \in E_{res}$
$V^i_{res} \subseteq V_{res}$	Subset of nodes where $i \in V_{dsp}$ can be placed
$\mathcal{X} \sqsubset X$	Multiset of elements in X
$x_{i,\mathcal{U}}$	Placement of $i \in V_{dsp}$ on nodes in $\mathcal{U} \sqsubset V^i_{res}$
$y_{(i,j),(\mathcal{U},\mathcal{V})}$	Placement of $(i,j) \in E_{dsp}$ on the network paths from nodes in $\mathcal{U} \sqsubset V^i_{res}$ to nodes in $\mathcal{V} \sqsubset V^j_{res}$
z_u	Activation variable for $u \in V_{res}$
$z_{(u,v)}$	Activation variable for $(u,v) \in E_{res}$

ment problem. For the sake of clarity, in Table 1 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, 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; S_u, the processing speed-up on a reference processor; and A_u, its availability, i.e., the probability that u is up and running. 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; $A_{(u,v)}$, the link availability, i.e., the probability that the link between u and v is active; 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 Model

A DSP application can be represented at different levels of abstraction, and we can distinguish between an *abstract model*, which is defined by the user, and an *execution model*, which is 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

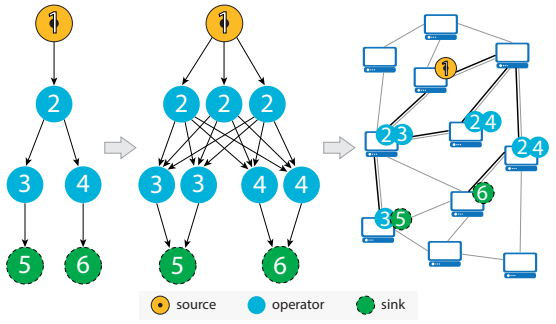


Figure 1: Replication of the application operators and their placement on the computing resources

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). 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 operator latency (which accounts for the waiting time on the input queues as well as the execution time of a unit of data); C_i, the cost of deploying an instance of the operator. We characterize the stream exchanged from operator i to j, $(i,j) \in E_{dsp}$, with its average tuple rate $\lambda_{(i,j)}$ and average number of bytes per tuple $b_{(i,j)}$. To model load-dependent latency, we assume that the latency is function of λ_i, the operator input tuple rate, $R_i = R_i(\lambda_i)$, 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.

The DSP *execution model* is obtained from the abstract model by replacing each operator with its replicas. Indeed, in order to improve performance, multiple instances (or replicas) of the same DSP operator can be instantiated over different computing nodes. The premise is that by partitioning the streams over multiple processing elements, we reduce the load of each processing element which in turn yields lower operator (and overall application) latency. Differently from most of the existing solutions, ODRP computes the execution model by optimizing, in a single stage, the number of operator instances and their placement.

3.3 Operator Replication and Placement

The DSP replication and placement 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}. Figure 1 represents a simple instance of the problem.

Observe that 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). This observation allows us to consider for each operator $i \in V_{dsp}$ a subset of candidate resources $V^i_{res} \subseteq V_{res}$ where it can be deployed. For example, if sources and sinks ($I \subset V_{dsp}$) are external applications, their placement is fixed, that is $\forall i \in I, |V^i_{res}| = 1$. The operator 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}. We recur 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 $\mathcal{X}$ over a set X, which we denote as $\mathcal{X} \sqsubset X$, is defined as a mapping $\mathcal{X} : X \to \mathbb{N}$ where, for $x \in X$, $\mathcal{X}(x)$ denotes the multiplicity of x in $\mathcal{X}$. $x \in \mathcal{X}$ if and only if $\mathcal{X}(x) \geq 1$. The cardinality of a multiset $\mathcal{X}$, denoted $|\mathcal{X}|$, is defined by the number of elements in $\mathcal{X}$, that is $|\mathcal{X}| = \sum_{x \in \mathcal{X}} \mathcal{X}(x)$. Hereafter, without lack of generality, we will assume that in a deployment each operator $i \in V_{dsp}$ can be replicated at most k_i times. Therefore, we also find convenient to define the power multiset $\mathcal{P}(X)$ of a set X as the set of all multisets with elements taken from X and the subset $\mathcal{P}(X; k) \subset \mathcal{P}(X)$ of the multiset over X with cardinality no greater of k, that is $\mathcal{P}(X; k) = \{\mathcal{X} \in \mathcal{P}(X) | \sum_{x \in \mathcal{X}} \mathcal{X}(x) \leq k\}$.

4. OPTIMAL REPLICATION AND PLACEMENT MODEL

In this section we present our model for the ODRP problem. As the optimal solution depends on non-functional attributes, we first derive the expression for the different QoS metrics of interest and then present the ODRP formulation.

4.1 ODRP Variables

We model the ODRP problem with binary variables $x_{i,\mathcal{U}}$, $i \in V_{dsp}$ and $\mathcal{U} \sqsubset V^i_{res}$: $x_{i,\mathcal{U}} = 1$ if and only if the operator $map(i) = \mathcal{U}$, 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 $y_{(i,j),(\mathcal{U},\mathcal{V})}$, $(i,j) \in E_{dsp}$, $\mathcal{U} \sqsubset V^i_{res}$, $\mathcal{V} \sqsubset V^j_{res}$, 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 $y_{(i,j),(\mathcal{U},\mathcal{V})} = x_{i,\mathcal{U}} \wedge x_{j,\mathcal{V}}$.

Finally, we also consider the variables z_u, $u \in V_{res}$ which denote whether at least one operator is deployed on node u and the variables $z_{(u,v)}$, $(u,v) \in E_{res}$, which denote whether a stream (or a portion of it) traverses the network path (u,v). By definition, we have $z_u = \vee_{i \in V_{dsp}, \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)} x_{i,\mathcal{U}}$ and $z_{(u,v)} = \vee_{(i,j) \in E_{dsp}, \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i), \mathcal{V} \in \mathcal{P}(V^j_{res}; k_j)} y_{(i,j),(\mathcal{U},\mathcal{V})}$. 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 $\boldsymbol{y} = \langle y_{(i,j),(\mathcal{U},\mathcal{V})} \rangle$, $\forall x_{i,\mathcal{U}}, x_{j,\mathcal{V}} \in \boldsymbol{x}$. Similarly, we denote by $\boldsymbol{z_V}$ and $\boldsymbol{z_E}$ the vectors $\boldsymbol{z_V} = \langle z_u \rangle$, $\forall u \in V_{res}$, and $\boldsymbol{z_E} = \langle z_{(u,v)} \rangle$, $\forall (u,v) \in E_{res}$.

4.2 Qos Metrics

4.2.1 Operator QoS Metrics

Let us first consider an operator in isolation. For each $i \in V_{dsp}$, the QoS of the operator deployment depends on the deployment $\mathcal{U}$. Let $R_{i,\mathcal{U}}$, $C_{i,\mathcal{U}}$, and $A_{i,\mathcal{U}}$ denote the maximum latency, the cost, and the availability of the deployment $\mathcal{U}$, respectively. We readily have:

$$R_{i,\mathcal{U}} = \max_{u \in \mathcal{U}} \frac{R_i(\frac{\lambda_i}{|\mathcal{U}|})}{S_u} \quad (1)$$

$$C_{i,\mathcal{U}} = \sum_{u \in \mathcal{U}} \mathcal{U}(u) C_i Res_i \quad (2)$$

$$A_{i,\mathcal{U}} = \prod_{u \in \mathcal{U}} A_u \quad (3)$$

under the assumption that the traffic is equally split among the different operator replicas.

4.2.2 Stream QoS Attributes

We now turn our attention to the QoS attributes related to a stream. For a stream (i,j), the QoS depends on the upstream and downstream operators' deployments $\mathcal{U}$ and $\mathcal{V}$. Let $d_{(i,j),(\mathcal{U},\mathcal{V})}$, $C_{(i,j),(\mathcal{U},\mathcal{V})}$, and $A_{(i,j),(\mathcal{U},\mathcal{V})}$ denote the maximum latency, the cost, and the availability of the deployments $\mathcal{U}$ and $\mathcal{V}$, respectively. We readily have:

$$d_{(i,j),(\mathcal{U},\mathcal{V})} = d_{(\mathcal{U},\mathcal{V})} = \max_{u \in \mathcal{U}, v \in \mathcal{V}} d_{(u,v)} \quad (4)$$

$$C_{(i,j),(\mathcal{U},\mathcal{V})} = \sum_{u \in \mathcal{U}, v \in \mathcal{V}} \lambda_{(i,j),(\mathcal{U},\mathcal{V})} C_{u,v} \quad (5)$$

$$A_{(i,j),(\mathcal{U},\mathcal{V})} = \prod_{u \in \mathcal{U}, v \in \mathcal{V}} A_{(u,v)} \quad (6)$$

where

$$\lambda_{(i,j),(\mathcal{U},\mathcal{V})} = \frac{\lambda_{(i,j)}}{|\mathcal{U}||\mathcal{V}|} \quad u \in \mathcal{U}, v \in \mathcal{V} \quad (7)$$

is the amount of stream (i,j) traffic exchanged between two operator replicas under the deployments $\mathcal{U}$ and $\mathcal{V}$.

4.2.3 DSP Application QoS Metrics

We consider both user-oriented and system-oriented QoS metrics, such as application response time, cost, and availability for the former, and network related metrics for the latter.

Response Time: For a DSP application, with data flowing from several sources to several destinations, there is no unique definition of response time. In the following, we consider as response time R the worst end-to-end delay from a source to a sink. Given this definition, we have that:

$$R = \max_{\pi \in \Pi_{dsp}} R_\pi \quad (8)$$

being R_π the delay along path π and Π_{dsp} the set of all source-sink paths in G_{dsp}. Given a placement vector $\boldsymbol{x}$ (and resulting $\boldsymbol{y}$) and a path $\pi = (i_1, i_2, \ldots, i_{n_\pi})$, we have $R = R(\boldsymbol{x}, \boldsymbol{y}) = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y})$ with $R_\pi(\boldsymbol{x}, \boldsymbol{y})$ defined as:

$$R_\pi(\boldsymbol{x}, \boldsymbol{y}) = \sum_{p=1}^{n_\pi} R_{i_p}(\boldsymbol{x}) + \sum_{p=1}^{n_\pi - 1} D_{(i_p, i_{p+1})}(\boldsymbol{y}) \quad (9)$$

where

$$R_i(\boldsymbol{x}) = \sum_{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)} R_{i,\mathcal{U}} x_{i,\mathcal{U}} \quad (10)$$

$$D_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \\ \mathcal{V} \in \mathcal{P}(V^j_{res}; k_j)}} d_{(\mathcal{U},\mathcal{V})} y_{(i,j),(\mathcal{U},\mathcal{V})} \quad (11)$$

denote respectively the execution time of operator i when deployed over the multiset $\mathcal{U}$ and the worst case network delay for transferring data from i to j when the two operators are mapped over $\mathcal{U}$ and $\mathcal{V}$, respectively.

Cost: We define the cost C of the DSP application as the monetary cost of all the computing resources and paths involved in the processing and transmission of the application data streams. We have:

$$C(\boldsymbol{x},\boldsymbol{y}) = \sum_{i\in V_{dsp}} C_i(\boldsymbol{x}) + \sum_{(i,j)\in E_{dsp}} C_{(i,j)}(\boldsymbol{y}) \tag{12}$$

where

$$C_i(\boldsymbol{x}) = \sum_{\mathcal{U}\in\mathcal{P}(V^i_{res};k_i)} C_{i,\mathcal{U}}x_{i,\mathcal{U}} \tag{13}$$

$$C_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{\mathcal{U}\in\mathcal{P}(V^i_{res};k_i)\\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)}} C_{(i,j),(\mathcal{U},\mathcal{V})}y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{14}$$

Availability: We define the application availability A as the availability of all the nodes and paths involved in the processing and transmission of the application data streams. For the sake of simplicity, we assume the availability of the different components to be independent. However, we acknowledge that independence does not hold true in general and that a more detailed model is needed to capture the dependency relationship among logical components sharing physical nodes and networks links; we postpone it to future work. With the independence assumption, we readily have:

$$A(\boldsymbol{z_V},\boldsymbol{z_E}) = \prod_{u\in V_{res}:z_u=1} A_u z_u \cdot \prod_{(u,v)\in E_{res}:z_{(u,v)}=1} A_{(u,v)}z_{(u,v)} \tag{15}$$

To obtain a linear expression, we consider the logarithm of the availability, obtaining:

$$\log A(\boldsymbol{z_V},\boldsymbol{z_E}) = \sum_{u\in V_{res}} a_u z_u + \sum_{(u,v)\in E_{res}} a_{(u,v)}z_{(u,v)} \tag{16}$$

where $a_u = \log A_u$ and $a_{(u,v)} = \log A_{(u,v)}$. It is worth observing that in (16) we can take the summation over all $u \in V_{res}$ and $(u,v) \in E_{res}$ since the terms not appearing in (15) are those corresponding to $z_u = 0$ or $z_{(u,v)} = 0$, which do not affect the summation in (16).

Network Related QoS Metrics: In the DSP literature, several alternative network-aware metrics have been defined, including the inter-node traffic T [1], the network usage N [26], and an approximation of the elastic energy EE [22]. Let $Z(\boldsymbol{y})$, $Z = T|N|EE$, denote the QoS attribute of the DSP application under the placement policy $\boldsymbol{y}$, we have:

$$Z(\boldsymbol{y}) = \sum_{(i,j)\in E_{dsp}} Z_{(i,j)}(\boldsymbol{y}) \tag{17}$$

where $Z_{(i,j)}(\boldsymbol{y})$ is defined as follows.

The *inter-node traffic* T is the overall amount of data exchanged per time unit between operators placed on different nodes. Therefore, using the placement policy $\boldsymbol{y}$, the stream $(i,j) \in E_{dsp}$ generates an inter-node traffic equals to:

$$T_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{u\in\mathcal{U},v\in\mathcal{V},u\neq v\\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)\\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)}} b_{(i,j)}\lambda_{(i,j),(\mathcal{U},\mathcal{V})}y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{18}$$

The *network usage* N is the amount of data that traverses the network at a given time; therefore, the stream $(i,j) \in E_{dsp}$ imposes a load expressed by:

$$N_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{u\in\mathcal{U},v\in\mathcal{V},u\neq v\\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)\\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)}} b_{(i,j)}\lambda_{(i,j),(\mathcal{U},\mathcal{V})}d_{(u,v)}y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{19}$$

where $d_{(u,v)}$ is the network delay among nodes $u, v \in V_{res}$, with $u \neq v$.

In their paper [22], Pietzuch et al. indirectly minimize the network usage through the minimization of the elastic energy, which results from the equivalent system of springs that represents the application. Basically, their solution minimizes the amount of data that traverses each link weighted by the latency of the link itself. Hence, the stream $(i,j) \in E_{dsp}$ contributes to the elastic energy of the system with:

$$EE_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{u\in\mathcal{U},v\in\mathcal{V},u\neq v\\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)\\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)}} b_{(i,j)}\lambda_{(i,j),(\mathcal{U},\mathcal{V})}d^2_{(u,v)}y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{20}$$

Observe that, in all cases, $Z(\boldsymbol{y})$ is a linear function of $\boldsymbol{y}$.

4.3 ODRP Formulation

Depending on the usage scenario, a DSP replication and placement strategy could be aimed at optimizing different, possibly conflicting, QoS attributes. To this end, we use the Simple Additive Weighting (SAW) technique [27] to define the utility function $F(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z_V},\boldsymbol{z_E})$ as a weighted sum of the normalized QoS attributes of the application, as follows:

$$\begin{aligned} F(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z_V},\boldsymbol{z_E}) = {} & w_r\frac{R_{\max}-R(\boldsymbol{x},\boldsymbol{y})}{R_{\max}-R_{\min}} + w_a\frac{\log A(\boldsymbol{z_V},\boldsymbol{z_E})-\log A_{\min}}{\log A_{\max}-\log A_{\min}}\\ & + w_c\frac{C_{\max}-C(\boldsymbol{x},\boldsymbol{y})}{C_{\max}-C_{\min}} + w_z\frac{Z_{\max}-Z(\boldsymbol{x},\boldsymbol{y})}{Z_{\max}-Z_{\min}} \end{aligned} \tag{21}$$

where $w_r,\ w_a,\ w_c,\ w_z \geq 0$, $w_r + w_a + w_c + w_z = 1$, are weights associated to the different QoS attributes. $R_{\max}$ ($R_{\min}$), $A_{\max}$ ($A_{\min}$), $C_{\max}$ ($C_{\min}$), and $Z_{\max}$ ($Z_{\min}$) denote, respectively, the maximum (minimum) value for the overall expected response time, availability, cost and network related metric. Observe that after normalization, each metric ranges in the interval $[0,1]$, where the value 1 corresponds to the best possible case and 0 to the worst case.

We formulate the ODRP problem as an Integer Linear Programming (ILP) model as follows:

$$\max_{\boldsymbol{x},\boldsymbol{y},r} F'(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z_V},\boldsymbol{z_E},r)$$

subject to:

$$\begin{aligned} r \geq {} & \sum_{\substack{p=1,\ldots,n_\pi\\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)}} R_{i_p,\mathcal{U}}x_{i,\mathcal{U}} + \\ & \sum_{\substack{p=1,\ldots,n_\pi-1\\ q=p+1\\ \mathcal{U}\in\mathcal{P}(V^{i_p}_{res};k_{i_p})\\ \mathcal{V}\in\mathcal{P}(V^{i_q}_{res};k_{i_q})}} d_{(\mathcal{U},\mathcal{V})}y_{(i_p,i_q),(\mathcal{U},\mathcal{V})} \qquad \forall\pi\in\Pi_{dsp} \end{aligned} \tag{22}$$

$$Res_u \geq \sum_{\substack{i \in V_{dsp} \\ \mathcal{U} \in \mathcal{P}(V^i_{res})}} \mathcal{U}(u) Res_i x_{i,\mathcal{U}} \qquad \forall u \in V_{res} \tag{23}$$

$$z_u \geq \frac{\sum_{i \in V_{dsp}} \sum_{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)} x_{i,\mathcal{U}}}{M} \qquad u \in V_{res} \tag{24}$$

$$z_{(u,v)} \geq \frac{\sum_{(i,j) \in E_{dsp}} \sum_{\substack{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \\ \mathcal{V} \in \mathcal{P}(V^j_{res}; k_j)}} y_{(i,j),(\mathcal{U},\mathcal{V})}}{N} \qquad (u,v) \in E_{res} \tag{25}$$

$$1 = \sum_{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)} x_{i,\mathcal{U}} \qquad \forall i \in V_{dsp} \tag{26}$$

$$x_{i,\mathcal{U}} = \sum_{\mathcal{V} \in \mathcal{P}(V^j_{res}; k_j)} y_{(i,j),(\mathcal{U},\mathcal{V})} \qquad \begin{matrix} \forall (i,j) \in E_{dsp}, \\ \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \end{matrix} \tag{27}$$

$$x_{j,\mathcal{V}} = \sum_{\mathcal{U} \in \mathcal{P}(V^i_{res}; k_u)} y_{(i,j),(\mathcal{U},\mathcal{V})} \qquad \begin{matrix} \forall (i,j) \in E_{dsp}, \\ \mathcal{V} \in \mathcal{P}(V^j_{res}; k_j) \end{matrix} \tag{28}$$

$$x_{i,\mathcal{U}} \in \{0,1\} \qquad \begin{matrix} \forall i \in V_{dsp}, \\ \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \end{matrix} \tag{29}$$

$$y_{(i,j),(\mathcal{U},\mathcal{V})} \in \{0,1\} \qquad \begin{matrix} \forall (i,j) \in E_{dsp}, \\ \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i) \\ \mathcal{V} \in \mathcal{P}(V^j_{res}; k_j) \end{matrix} \tag{30}$$

$$z_u \in \{0,1\} \qquad \forall u \in V_{res} \tag{31}$$

$$z_{(u,v)} \in \{0,1\} \qquad \forall (u,v) \in E_{res} \tag{32}$$

In the problem formulation we use the objective function $F'(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z_V}, \boldsymbol{z_E}, r)$ that is obtained from $F(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z_V}, \boldsymbol{z_E})$ by replacing $R(\boldsymbol{x}, \boldsymbol{y})$ with the auxiliary variable r, which represents the application response time in the optimization problem, in order to obtain a linear objective function. Observe that, indeed, while F is nonlinear in $\boldsymbol{x}$, $\boldsymbol{y}$ since $R(\boldsymbol{x}, \boldsymbol{y}) = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y})$ is a nonlinear term, F' is linear in r as well as in $\boldsymbol{x}$ and $\boldsymbol{y}$. Equation (22) follows from (8)–(11). Since r must be larger or equal than the response time of any path and, at the optimum, r is minimized, $r = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y}) = R(\boldsymbol{x}, \boldsymbol{y})$. The constraint (23) limits the placement of operators on a node $u \in V_{res}$ according to its available resources. Constraints (24) and (25) are the activation constraints for the variable z_u and $z_{(u,v)}$, respectively, with M and N large constants. Equation (26) guarantees that each operator $i \in V_{dsp}$ is placed on one and only one node $u \in V^i_{res}$. Finally, constraints (27)–(28) 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 ODRP problem is an NP-hard problem.*

PROOF. It is sufficient to observe that the Optimal DSP Replication and Placement problem is a generalization of the Optimal DSP Placement problem we presented in [4], which has been shown to be NP-hard. □

5. STORM INTEGRATION

To enable the usage of ODRP in a real DSP framework, we have developed a prototype scheduler for Apache Storm, named S-ODRP. We first briefly describe the main features of Storm and how it represents and executes DSP applications. Then, we present the prototype design in details.

5.1 Apache Storm

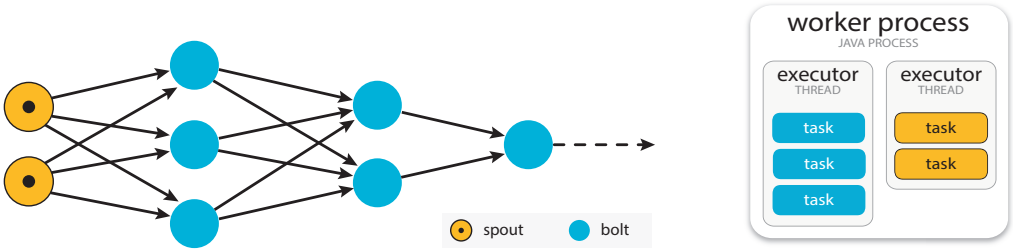
Storm is an open source, real-time, and scalable DSP system maintained by the Apache Software Foundation. It provides an abstraction layer where event-based applications can be executed over a set of worker nodes interconnected by an overlay network. A *worker node* is a generic computing resource (e.g., a physical host, a mobile device, a virtual machine, a Docker container), whereas the overlay network comprises the logical links among these nodes.

In Storm, an application is represented by its *topology*, which is a DAG with spouts and bolts as vertices and streams as edges. A *spout* is a data source that feeds the data into the system through one or more streams. A *bolt* is either a processing element, which extracts valuable information from incoming tuples, or a final information consumer; a bolt can also generate new outgoing streams, like spouts do. A *stream* is an unbounded sequence of *tuples*, which are key-value pairs. We refer to spouts and bolts as operators. Figure 2a shows an example of a DSP application.

Storm uses three types of entities with different grain to execute a topology: tasks, executors, and worker processes. 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. An *executor*, which is the smallest schedulable unit, can execute one or more tasks related to the same operator; in other words, t_i, the number of tasks of an operator, is always greater or equal than e_i, the number of executors of the same operator, that is, $t_i \geq e_i$. Since the operator executors run concurrently, they can increase the operator throughput when subject to heavy incoming load. A *worker process* is a Java process that runs a subset of the executors of the *same* topology, i.e., a topology can be distributed across different worker processes. As represented in Figure 2b, 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 in its turn serves as container on the worker node. To date, Storm leaves completely to the user the definition of the number of worker processes, executors, and tasks for the DSP application. Moreover, the framework enables the user to change at runtime the parallelism degree of an application through the `rebalance` API; however, its implementation is not efficient, because it restarts the whole application with new executors, leading to possible data loss.

Besides the computing resources, i.e., the worker nodes, the Storm architecture 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 all the worker nodes through *ZooKeeper*[2], which is a shared in-memory service for managing configuration information and enabling distributed coordination. Since each worker

[2] `http://zookeeper.apache.org/`



(a) Topology of a DSP application (b) Execution model

Figure 2: Storm abstractions

node can execute one or more worker processes, a *Supervisor* component on each worker node starts or terminates worker processes on the basis of the Nimbus assignments. Each worker node can concurrently run a limited number of worker processes, based on the number of available *worker slots*.

5.2 S-ODRP: ODRP in Storm

We develop a new scheduler for Storm, named **S-ODRP**, whose core is the model presented in Section 4. In order to design S-ODRP, we have to address two issues: (1) to adapt the DSP and resource model to consider the specific execution entities of Storm, and (2) to instantiate the ODRP model with the proper QoS information about computing and networking resources.

As regards the first issue, 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-ODRP defines $G_{dsp} = (V_{dsp}, E_{dsp})$, with V_{dsp} as the set of operators and E_{dsp} as the set of streams exchanged between 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-ODRP determines the number of executors for each operator $i \in V_{dsp}$, leveraging on the cardinality of $\mathcal{U}$ when $x_{i,\mathcal{U}} = 1$, with $\mathcal{U} \sqsubset V_{res}^i$. 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-ODRP considers the amount of available resources C_u on a worker node $u \in V_{res}$ 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.

As regards the second issue, Storm allows us to easily develop new centralized schedulers with the pluggable scheduler APIs. However, Storm is unaware of the QoS attributes of its networking and computing resources, except for the number of available worker slots. Since we need to know these QoS attributes in order to apply the ODRP model, we rely on Distributed Storm, a Storm extension[3] we presented in [2], that enables the QoS awareness of the scheduling system by providing intra-node (i.e., availability) and inter-node (i.e., network delay and exchanged data rate) information. This extension estimates network latencies using a network coordinate system, which is built through the Vivaldi algorithm [7], a decentralized algorithm having linear complexity with respect to the number of network locations. S-ODRP retrieves, from the monitoring components of the extended Storm, the information needed to parametrize the nodes and edges in G_{dsp} and G_{res}. Specifically, it considers: the average data rate exchanged between communicating executors (i.e., $\lambda_{(i,j)}, \forall (i,j) \in E_{dsp}$), the node availability ($A_u, \forall u \in V_{res}$), and the network latencies ($d_{(u,v)}, \forall u, v \in V_{res}$). Once built the ODRP model, S-ODRP relies on CPLEX©[4], the state-of-the-art solver for ILP problems, for its resolution.

From an operational prospective, Nimbus uses S-ODRP to compute the optimal operator replication and placement when a new application is submitted to Storm and when a failure of the worker process compromises the application execution. In the latter case, S-ODRP invalidates the existing assignment and computes the new optimal placement. Algorithm 1 summarizes the runtime execution of S-ODRP, which has to face two main issues: to collect the exchanged data rate between the operators, and to replicate the operators as needed. When information on the exchanged data rate is unknown (line 3), e.g., the first time the application is scheduled, S-ODRP defines an early assignment and monitors the application execution to harvest the needed information (lines 4–6). As soon as this information is available, S-ODRP reassigns the application by solving the updated ODRP model with the network-related QoS attributes (line 8). To enact the replication decision computed by ODRP (lines 5 and 9), S-ODRP leverages on the Storm API `rebalance`, which restarts the application with the correct number of executors, before assigning them to the worker nodes as specified by the computed placement solution.

Algorithm 1 Application placement with S-ODRP

```
1: function SCHEDULE(G_dsp, G_res)
2:     RP = [ ]                          ▷ replication and placement
3:     if not streamsDatarateAvailable(G_dsp) then
4:         RP ← ODRP(G_dsp, G_res)
5:         enact(RP)
6:         G_dsp ← collectStreamsDatarate(G_dsp)
7:     end if
8:     RP ← ODRP(G_dsp, G_res)
9:     enact(RP)
10: end function
```

[3]Source code available at `http://bit.ly/extstorm`

[4]`http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`

6. EXPERIMENTAL RESULTS

Our experiments revolve around S-ODRP, the prototype scheduler for Storm whose core is ODRP, and they aim to show the generality and flexibility of the proposed formulation as well as its impact in terms of achievable application performance. After introducing the experimental setup and the reference application (Section 6.1), we provide a general overview on the runtime execution of S-ODRP (Section 6.2). Then, in Section 6.3, we show the benefits of the joint optimization of placement and replication when the DSP application is subject to an increasing load. Finally, in Section 6.4, we evaluate how S-ODRP can optimize several QoS metrics, such as response time, cost, availability, and inter-node traffic.

6.1 Experimental Setup

We perform the experiments using Apache Storm 0.9.3 on a cluster of 6 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 define 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. We emulate wide-area network latencies among the Storm nodes using `netem`, which applies to outgoing packets a Gaussian delay with mean and standard deviation in the ranges $[12, 32]$ ms and $[1, 3]$ ms, respectively. As regards the pricing policy, we charge only the usage of computing resources, i.e., we set a unitary cost for each operator replica. We solve the ILP problem using CPLEX© (version 12.6.3) on the node hosting Nimbus.

As test-case application we developed a benchmarking application that solves the first query of the DEBS 2015 Grand Challenge [17]: 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. Its topology is represented in Figure 3. The *data source* reads the dataset from Redis, an in-memory data store, and pushes data towards a *parser* operator, 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, and *countByWindow* counts the route frequency in the last 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 following operators, *partialRank* and *globalRank*, compute the top-10 most frequent routes by leveraging on a two-step approach that enables to compute 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. Moreover, since we investigate the initial application placement, we have set the data source so to feed the topology with a constant data rate, defined a-priori.

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. Without loss of generality, in the ODRP model we estimate the response time R_i of 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 and the other configuration parameters have been obtained through preliminary experiments and are shown in Table 2.

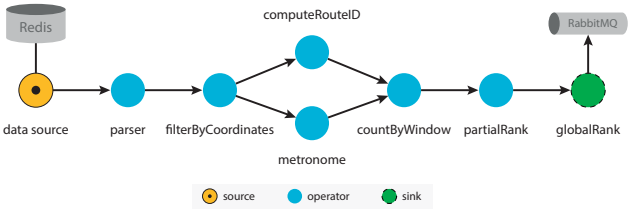


Figure 3: Reference DSP application

Table 2: Parameters of the experimental setup

Application: service rate per operator, expressed in tuples/s (tps)

Operator	μ_i	Operator	μ_i
data source	284 tps	metronome	190 tps
parser	233 tps	countByWindow	335 tps
filterByCoordinates	253 tps	partialRank	2371 tps
computeRouteID	253 tps	globalRank	185 tps

Normalization factors for the ODRP utility function

Parameter	Value	Parameter	Value
$R_{\min}$	5 ms	$R_{\max}$	450 ms
$A_{\min}$	95%	$A_{\max}$	100%
$C_{\min}$	8	$C_{\max}$	18
$Z_{\min}$	0 tps	$Z_{\max}$	3400 tps

6.2 Evaluation of S-ODRP

This first experiment aims at showing the runtime execution of the S-ODRP scheduler, described by Algorithm 1, and its impacts on the application performance. As baseline we use S-ODP, a prototype scheduler for Storm whose core is the ODP model that optimizes only the operator placement [4]. Note that ODP is a special case of ODRP where $k_i = 1, \forall i \in V_{dsp}$. To simplify the presentation, we consider only the response time R and the monetary cost C as QoS metrics; all worker nodes have an availability of 100%. Both the optimization models focus on the minimization of the application response time R (i.e., $w_r = 1$, $w_c = 0$), so we name them as **S-ODRP_R** and **S-ODP_R**, respectively. Differently from S-ODP_R, S-ODRP_R computes R relying on the exchanged data-rate between the operators; as presented in Section 5, it deploys the application with a preliminary placement so to harvest the relevant data; this preliminary placement is computed by minimizing the deployment cost (i.e., $w_c = 1$, $w_r = 0$). The rescheduling event takes place after 100 s of execution and is represented in Figure 4 with a vertical line.

We set the data rate of the source operator to 80 tuples/s and launch the application. Figure 4 reports the resulting application response time. We observe that as soon as the placement is defined, i.e., after 0 s and after 100 s (the latter for S-ODRP_R only), a transient period is experimented where R is quite high and exceeds 1 s. This behavior depends on a well-known issue of the Storm framework [26], which starts the operators as soon as they are ready without coordination at level of the whole application; as a consequence, data emitted by an operator wait in inter-operator

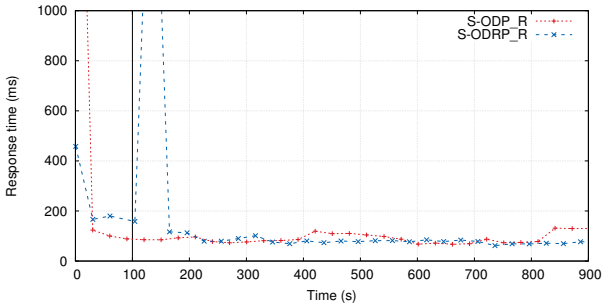


Figure 4: Runtime execution of S-ODRP and S-ODP

Table 3: Operator replication

Operator	S-ODP	S-ODRP		
		20, 40, 60 tuples/s	80, 100 tuples/s	120 tuples/s
data source	1	1	1	1
parser	1	1	1	3
filterByCoordinates	1	1	1	2
computeRouteID	1	1	2	2
metronome	1	1	1	1
countByWindow	1	1	1	3
partialRank	1	1	2	3
globalRank	1	1	1	1

buffers until the following operator is up and running for processing. When the transient period ends, after 200 s, the applications deployed with the two schedulers experience quite similar performance in terms of response time.

Table 3 reports the total number of operator replicas deployed by the two schedulers. S-ODP_R cannot replicate the operators, therefore it instantiates a replica for each of them. With a source data rate of 80 tuples/s, S-ODRP_R replicates twice two operators, namely *computeRouteID* and *partialRank*, and runs the application with a total of 10 executors. S-ODRP_R replicates the operators as much as possible while considering that, when a new replica has to be located on a new worker node, the latter introduces network latencies that can overcome the benefits of replication in reducing the operator execution time. In this experiment, it is worth to observe that, although the scheduler is forced to use a second worker node, it places the replicas so to minimize the response time; in particular, only the replicas of *computeRouteID*, which have the lowest data rate exchanged with the other operators, are located on a separate node.

6.3 Impact of Replication

In the second set of experiments, we want to investigate the replication benefits when the application is subject to different incoming loads. We use the same settings of the previous experiment except for the source data rate and we compare how S-ODRP_R and S-ODP_R determine the placement of the reference application. In each single experiment, which lasts 900 s, the source data rate is constant and is set in the range $[20, 120]$ tuples/s with step 20. We collect the resulting QoS metrics as soon as the transient period ends (i.e., after 200 s) and we summarize the results in Figure 5 leveraging on a boxplot, which represents the QoS metric distribution through the minimum value, the 5th percentile, 50th percentile, 95th percentile, and the maximum value; the average value is also represented using a full dot. We define as *active node utilization*, the average utilization of all the worker nodes involved in the application execution; each node contributes with its average utilization calculated on a time window of 60 s.

Figure 5a reports the application response time and Table 3 the number of replicas per operator. Although S-ODP_R cannot replicate the operators, it finds the optimal placement that minimizes R, as S-ODRP_R does. Indeed, when the replication is not needed, i.e., up to 60 tuples/s, the two schedulers achieve the same application performance. Note that, also in this case, network delays prevent S-ODRP_R from further replicating the operators: due to the absence of inter-node traffic (see Figure 5c), we can easily detect that all the 8 replicas run on the same node.

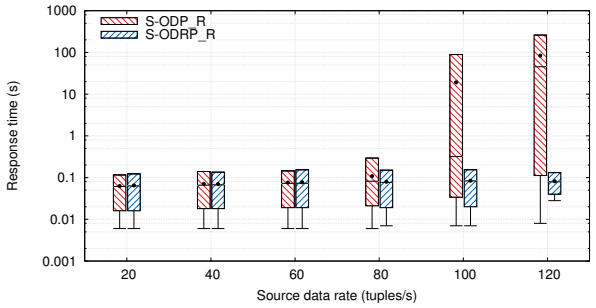
When the data source emits 80 tuples/s, the replication is needed: the *partialRank* operator represents a bottleneck, because it receives on average 2500 tuples/s, i.e., 5% more than its service rate. The utilization of the worker node that hosts the whole application for S-ODP_R, reported in Figure 5b, is around 20%, therefore the overload situation cannot be easily detected if not relying on fine-grain monitoring tools, which work at the level of single operator or CPU core. Conversely, S-ODRP_R detects the bottleneck and replicates the operator, which is then executed on a second worker node (see Table 3 and Figure 5c).

A similar behavior can be observed when the data source emits 100 tuples/s. This time the bottleneck operator, *partialRank*, receives on average 30% more tuples than a single replica can process per unit of time. With S-ODP_R, the application response time is unstable and continuously grows during the experiment, up to 106 s per single tuple. Conversely, thanks to replication, with S-ODRP_R the application maintains the same response time of the configuration with 80 tuples/s.

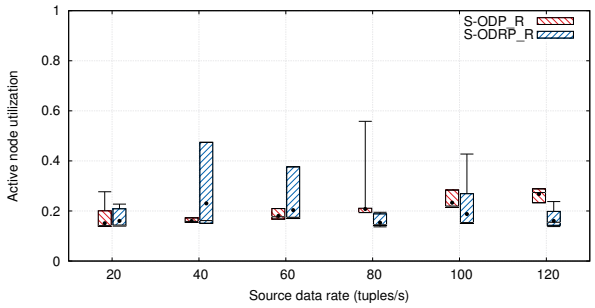
The need of replication is further exacerbated when the data source emits 120 tuples/s. With S-ODP_R, the application response time explodes up to $\sim$ 300 s per tuple. On the contrary, S-ODRP_R obtains an application response time that is not influenced by the increased load. To keep up with the incoming data rate, S-ODRP_R needs to replicate every operator. It instantiates 2 replicas for *filterByCoordinates* and *computeRouteID*, and 3 replicas for *parser*, *countByWindow*, and *partialRank*; comprising also the other operators, the application runs with a total of 16 executors (see Table 3) on 2 worker nodes. In spite of the increased incoming data rate, Figure 5c shows that the application deployment produces a fairly limited inter-node traffic. Finally, we observe from Figure 5b that, although the need of replication, the average value of the active node utilization is quite low. This behavior highlights that, for this specific use case, a static mapping between replicas and CPU cores does not lead to an efficient usage of resources; a possible solution to better exploit the available resources might be based on a dynamic multiplexing of replicas on the same CPU core. Since this optimization calls for the run-time adaptation of the application deployment, it falls out of the scope of this paper, but we plan to investigate more on it as future work.

6.4 Optimal Replication and Placement

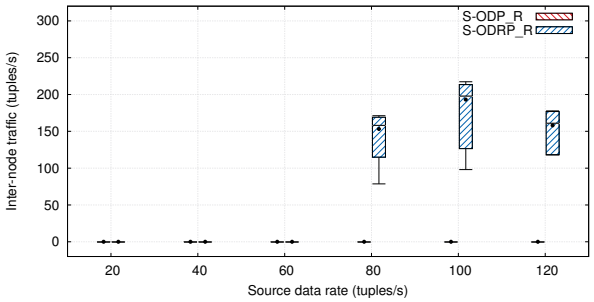
This experiment evaluates the effect of different optimization objectives on QoS metrics, i.e., availability A, cost C, response time R, and inter-node traffic T. To this end,



(a) Response time



(b) Active node utilization



(c) Inter-node traffic

Figure 5: Impact of replication on the application performance

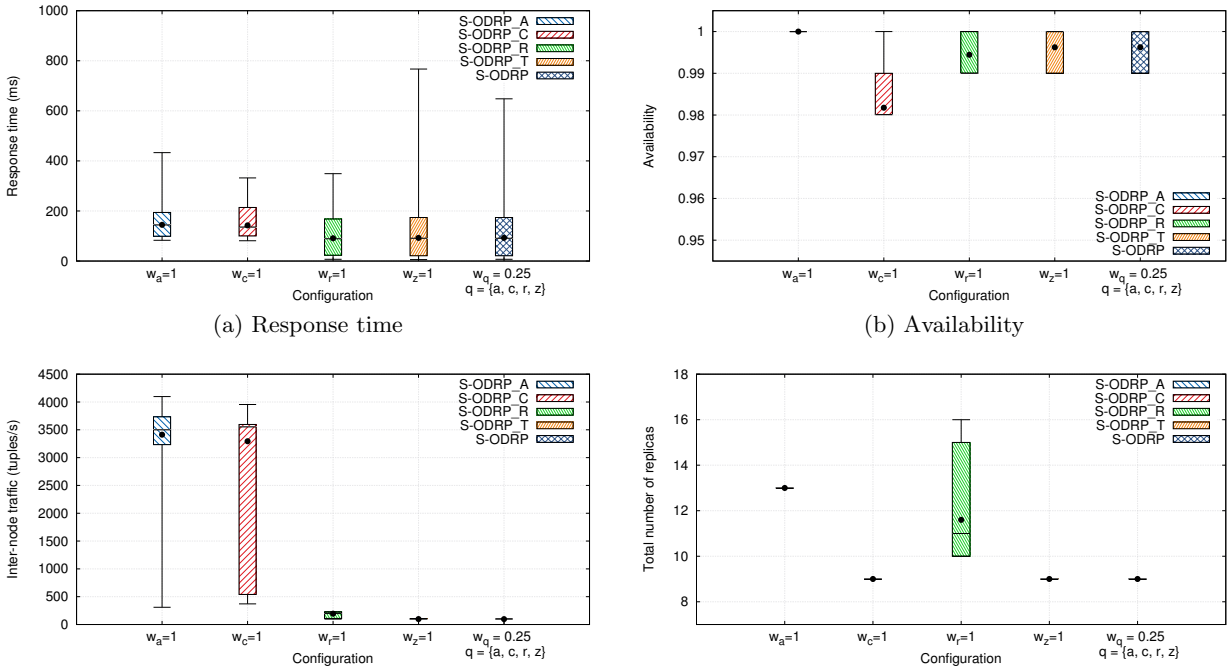
we set our experimental environment so that half of the worker nodes has an availability of 99% and the other of 100%, whereas the links are always available. We place the pinned operators (i.e., *data source* and *globalRank*) on a single worker node chosen randomly among those 100% available. S-ODRP determines the placement of the reference application when the data source emits 100 tuples/s. Figure 6 summarizes 25 runs, each of 900 s, where we collect QoS metrics after the initial transient period of 200 s. We first compute the replication and placement solution by optimizing a single QoS metric. For example, to optimize the response time we set the weights as $w_r = 1$, $w_a = w_c = w_z = 0$. Then, we optimize the multi-objective function by uniformly weighting each metrics contribution (i.e., $w_a = w_c = w_r = w_z = 0.25$); we report in Table 2 the normalization factors used in Equation (21). Figure 6 presents the results in term of the different QoS metrics.

When S-ODRP optimizes the application availability A (for short, **S-ODRP_A**), the solution finds the configuration where all the replicas are on the most available worker nodes. Observe that the placement of pinned operators, which are not relocated by S-ODRP, impacts on the overall application availability. In our experiments we placed these operators on nodes with 100% of availability. From Figure 6d and Equation (15), we observe that, although this configuration of S-ODRP does not optimize the number of operator instances, multiple replicas can be executed until a new worker node with availability lower than 100% has to be activated. This setting of weights does optimize neither the response time nor the inter-node traffic (see Figure 6a and 6c): on average, the response time is almost 1.6 times higher than the minimum achievable, whereas the inter-node traffic is almost 35 times higher than the optimal one.

When S-ODRP optimizes the cost C (**S-ODRP_C**), the placement solution tries to use fewer replicas as possible, as shown in Figure 6d. However, the explicit modeling of the operator service rate enables to instantiate a configuration that can properly handle the incoming traffic: S-ODRP instantiates 9 replicas, i.e., replicates twice the bottleneck operator (*partialRank*). Nothing can be concluded about the other QoS metrics, i.e., response time, application availability, and inter-node traffic (see Figures 6a, 6b, and 6c): since these metrics are not optimized, there is a set of equally optimal solutions that differ each other only for the operator placement on the same set of computing resources.

When S-ODRP optimizes response time (**S-ODRP_R**), the placement solution experiences the minimum achievable value for this metric, as shown in Figure 6a, but it uses up to 16 replicas (Figure 6d). Observe that 16 replicas completely occupy two worker nodes, and the presence of network delays prevents the scheduler from instantiating other replicas. Since the cost of running a configuration is directly proportional to the total number of operator instances, this is also the most expensive solution. From Figure 6c, we can also observe that the inter-node traffic is quite low (almost double with respect to the optimal value), because transmitting data over the network rather than locally introduces network delays that penalize the response time.

When S-ODRP optimizes the network-related QoS metric, that is the inter-node traffic T (**S-ODRP_T**), the solution tries to co-locate the operator instances on the least number of nodes, so to reduce the amount of data transmitted over the network (see Figure 6c). In this configuration the number of replicas is neither minimized nor maximized; however, if the load is equally split among replicas, S-ODRP might take advantage of replication in order to reduce the amount



(a) Response time

(b) Availability

(c) Inter-node traffic

(d) Total number of operator instances

Figure 6: Impact of different optimization objectives on the QoS metrics

of data transmitted on the network. As side effect of the co-location, the application response time is, on average, very close to the optimal one. Nothing can be concluded about the application availability (Figure 6b), which is not considered as an optimization objective of S-ODRP_T.

When S-ODRP optimizes all the considered QoS metrics, the resulting placement and replication solution has response time and inter-node traffic which are very close to the values achievable when optimizing a single-objective function. The total number of operator instances is equal to 9, as shown in Figure 6d, therefore only the bottleneck operator is replicated. The application availability ranges from 100% to 99%, and assumes 99.6% as average value. Although it might appear counter-intuitive, this result follows from the trade-off between the minimization of response and the maximization of the availability pursued in the utility function F. In particular, when the application availability is 99%, on the basis of the pinned operators placement, whose location is randomly defined a-priori, choosing a worker node that minimizes the response time, rather than one that maximizes the availability, provides a bigger contribution to the optimization of the utility function F.

On ODRP Resolution Time. We now discuss about the *resolution time* of ODRP and its relationship with the optimization goals. We consider as resolution time the time needed to compute the exact solution of the ILP problem. Although the investigated placement problem is fairly limited in size ($|V_{res}| = 6$, $|V_{dsp}| = 8$), the ODRP model includes about 55.8 K variables and, considering all the 25 runs of the last experiment, its average resolution time is 10.84 s. Figure 7 provides more details on the average resolution time of ODRP with respect to the different weights (w_a, w_c, w_r, and w_z) used for the utility function F. Determining a placement that minimizes the application response

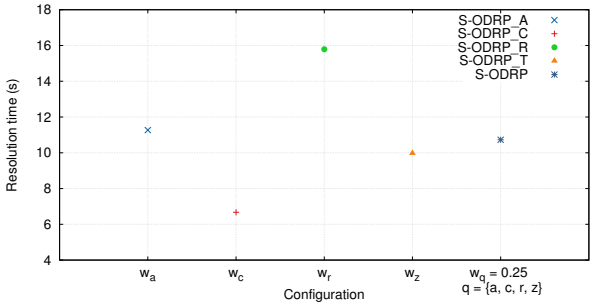


Figure 7: Average resolution time of ODRP with respect to the optimization objective

time is the most computationally demanding configuration; conversely, the minimization of the application deployment costs registers the fastest resolution time. Note that the former is twice slower than the latter. Being ODRP an NP-hard problem, as demonstrated in Theorem 1, it does not scale well as the problem instance increases in size. Nevertheless, by determining the optimal replication and placement of DSP operators, ODRP provides a benchmark for evaluating heuristics, for developing new ones, and for identifying the most suitable ones with respect to the specific optimization objectives.

7. CONCLUSIONS

In this paper, we have presented and evaluated ODRP, an ILP formulation that jointly optimizes the replication and placement of DSP applications. ODRP is a general and flexible model that can take into account the heterogeneity

of computing and networking resources and can be conveniently configured to optimize different QoS metrics, whose importance depends on the application scenario. With S-ODRP, we have developed an ODRP-based prototype scheduler for Apache Storm, one of the widely used open-source DSP frameworks. Then, relying on an application that processes real time data generated by taxis moving in a urban environment (DEBS 2015 Grand Challenge), we have conducted a thorough experimental evaluation. The latter has shown the benefits of a joint optimization of operators replication and placement on the application performance and how ODRP can contextually optimize several QoS metrics.

As future work, we plan to develop efficient heuristics to deal with large problem instances for the initial application replication and placement. Moreover, we plan to extend ODRP in order to support run-time adaptations of the operator deployment, so that a DSP application can efficiently handle input streams or execution environments with continuously changing characteristics. With the aim of optimizing reconfigurations, we will model their impact in terms of application performance degradation (e.g., temporary increment of response time due to operator state migrations).

8. ACKNOWLEDGMENTS

This publication is supported by COST Action ACROSS (IC1304).

9. REFERENCES

[1] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in Storm. In *Proc. of ACM DEBS '13*, pages 207–218, 2013.

[2] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Distributed QoS-aware scheduling in Storm. In *Proc. of ACM DEBS '15*, pages 344–347, 2015.

[3] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Joint operator replication and placement optimization for distributed streaming applications. In *Proc. of InfQ '16 (in conjunction with VALUETOOLS '16)*, 2016.

[4] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Optimal operator placement for distributed stream processing applications. In *Proc. of ACM DEBS '16*, pages 69–80, 2016.

[5] V. Cardellini, M. Nardelli, and D. Luzi. Elastic stateful stream processing in Storm. In *Proc. of HPCS '16*, pages 583–590. IEEE, 2016.

[6] A. Chatzistergiou and S. D. Viglas. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *Proc. of ACM CIKM '14*, 2014.

[7] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4), 2004.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of OSDI '04*, pages 137–150. USENIX Association, 2004.

[9] R. Eidenbenz and T. Locher. Task allocation for distributed stream processing. In *Proc. of IEEE INFOCOM '16*, 2016.

[10] L. Fischer, T. Scharrenbach, and A. Bernstein. Scalable linked data stream processing via network-aware workload scheduling. In *Proc. of SSWS '13*, pages 81–96, 2013.

[11] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, et al. DRS: Dynamic resource scheduling for real-time analytics over fast streams. In *Proc. of IEEE ICDCS 2015*, pages 411–420, 2015.

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

[13] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak. Cloud-based data stream processing. In *Proc. of ACM DEBS '14*, pages 238–245, 2014.

[14] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proc. of ACM DEBS '14*, pages 13–22, 2014.

[15] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *Proc. of IEEE ICDEW '14*, pages 296–302, 2014.

[16] T. Heinze, L. Roediger, A. Meister, Y. Ji, et al. Online parameter optimization for elastic data stream processing. In *Proc. of ACM SoCC '15*, pages 276–287, 2015.

[17] Z. Jerzak and H. Ziekow. The DEBS 2015 grand challenge. In *Proc. of ACM DEBS '15*, pages 266–268. ACM, 2015.

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

[19] T. Li, J. Tang, and J. Xu. A predictive scheduling framework for fast and distributed stream data processing. In *Proc. of IEEE Big Data '15*, 2015.

[20] B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. In *Proc. of IEEE ICDCS '15*, pages 399–410, 2015.

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

[22] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, et al. Network-aware operator placement for stream-processing systems. In *Proc. of IEEE ICDE '06*, 2006.

[23] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things J.*, 3(5):637–646, Oct. 2016.

[24] C. Thoma, A. Labrinidis, and A. Lee. Automated operator placement in distributed data stream management systems subject to user constraints. In *Proc. of IEEE ICDEW '14*, pages 310–316, 2014.

[25] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, et al. Storm@Twitter. In *Proc. of ACM SIGMOD '14*, pages 147–156, 2014.

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

[27] K. P. Yoon and C.-L. Hwang. *Multiple Attribute Decision Making: an Introduction*. Sage Pubs, 1995.

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of NSDI '12*. USENIX Association, 2012.