Distributed Operator Placement for IoT Data Analytics Across Edge and Cloud Resources

Eduard Gibert Renart*, Alexandre da Silva Veith†, Daniel Balouek-Thomert*, Marcos Dias de Assunção†,
Laurent Lefèvre† and Manish Parashar*
Rutgers University, Piscataway NJ, USA*
Inria, LIP, ENS Lyon, France†
Corresponding Author: egr33@rutgers.edu

Abstract—**The number of Internet of Things applications is forecast to exponentially grow within the coming decade. Owners of such applications strive to make predictions from large streams of complex input in near real time. Cloud-based architectures often centralize storage and processing, generating high data movement overheads that penalize real-time applications. Edge and Cloud architecture pushes computation closer to where the data is generated, reducing the cost of data movements and improving the application response time. The heterogeneity among the edge devices and cloud servers introduces an important challenge for deciding how to split and orchestrate the IoT applications across the edge and the cloud. In this paper, we extend our IoT Edge Framework, called R-Pulsar, to propose a solution on how to split IoT applications dynamically across the edge and the cloud, allowing us to improve performance metrics such as end-to-end latency (response time), bandwidth consumption, and edge-to-cloud and cloud-to-edge messaging cost. Our approach consists of a programming model and real-world implementation of an IoT application. The results show that our approach can minimize the end-to-end latency by at least 38% by pushing part of the IoT application to the edge. Meanwhile, the edge-to-cloud data transfers are reduced by at least 38%, and the messaging costs are reduced by at least 50% when using the existing commercial edge cloud cost models.**

Index Terms—**Edge Computing, Stream Processing, Edge analytics, Operator Placement**

I. Introduction

The continued growth of Internet of Things (IoT) applications is generating massive amounts of data. There are 2.5 quintillion bytes of data created each day and by 2020, it is estimated that every person will create 1.7MB every second [1]. Under many frameworks, this data is handled continuously and in real time, where the dataflows are structured as directed graphs whose vertices are operators that execute a function over the incoming data, and the edges define how data flows between the operators. These dataflows have one or multiple data sources (*i.e.*, sensors or actuators), operators that perform transformations on the data (*e.g.*, filtering, and aggregation) and sinks (*i.e.*, queries that consume or store the data).

The conventional approach for implementing these IoT applications is to send data from all sources to the cloud and leverage its powerful resources to process the incoming data streams and perform data analytics. However, is quickly becoming unsustainable due to the resulting impact on latency, network congestion, storage cost, and privacy. Moreover, cloud infrastructures charge for computing, networking, storage resources, and messages exchanged between the edge and the cloud, making the cloud-centric approach expensive and can even limit the potential impact that IoT can have.

In recent years, non-trivial computational capabilities have proliferated across the computing service landscape [2]. In particular, edge services are emerging close to the data sources and can provide potential data-processing capabilities. Therefore, the edge resources can be leveraged to complement the computing capabilities of the cloud-centric approach and reduce the overall latency and bandwidth requirements. Using such architecture also allows for exploring different cost models in order to minimize the cost of performing data analytics.

The use of the edge and cloud architecture poses the following challenges:

- Deciding how to split such IoT applications among the edge and cloud resources;
- Exploring heterogeneous infrastructure for deploying dataflow applications has proved to be NP-hard [3].
- Moving operators from cloud to edge devices is challenging due to the devices' limitations with respect to memory, CPU, and often network bandwidth [4].

Solving the challenges presented above in a correct manner will allow for faster completion time, a reduction in edge to cloud data transfers, costs, and ensure an efficient use of the edge and cloud resources. Doing them incorrectly can be detrimental to throughput and exacerbate the time for handling data events.

Existing work has provided a range of solutions for placing operators covering partial metrics such as end-to-end latency, WAN traffic, and monetary cost. Some solutions consider homogeneous computational resources [5], [6] while others include certain application constraints [7], [8]. More recent work tackles operator placement on edge infrastructure, but the proposed techniques require user intervention [9] and do not consider memory and communication constraints [10], [11]. Many solutions are either in the architectural level or evaluate the proposed algorithms using discrete-event simulation. Moreover, most work considers all data sinks to be located in the cloud, with no feedback loop to actuators located at the edge infrastructure [12], [13]. Furthermore, none of the systems mentioned above offer a programming abstraction with the flexibility to specify a set of dataflow constrains and automatically deploy and orchestrate the dataflow between the edge and the cloud.

To address these limitations, we propose a programming model to provide developers with the ability to define **how** to automatically split the dataflow across the edge and the cloud by specifying a set of dataflow constraints. Our approach relies on an already existing programming system, R-Pulsar, which enables developers to specify "What data need to be processed?," "Where to deploy the computations?," and "When to deploy the computations?" by using a content-based programming abstraction [14], [15].

In this paper we extend our existing programming system to address **how** to seamlessly split an IoT application among the edge and cloud by presenting a distributed dataflow orchestrator that can obtain optimal operator placement. The R-Pulsar orchestrator allows for minimizing the end-to-end latency, bandwidth, and monetary cost and makes defining, deploying, and orchestrating IoT applications across the edge and the cloud a seamless task.

This paper hence makes the following contributions:

- A programming abstraction for specifying **how** to split a given dataflow and place operators across edge and cloud resources.
- An operator placement strategy that aims to minimize an aggregate cost which covers the end-to-end latency (time for an event to traverse the entire dataflow), the data transfer rate (amount of data transferred between the edge and the cloud) and the messaging cost (number of messages transferred between edge and the cloud).
- An implementation of the above capabilities as part of the R-Pulsar software stack and its evaluation using an IoT application obtained from a data stream processing benchmark [16] that resembles a real-world application presenting a decision-making cycle.

II. Case Study: Observe Orient Decide Act Loop

The Observe Orient Decide Act (OODA) loop refers to the decision-making cycle of observe, orient, decide, and act, developed by military strategists and the United States Air Force [17]. OODA is a decision-making cycle to process data streaming from sensors in real time, becoming an essential design characteristic for IoT applications.

Anshu *et al.* [18] offer a suite of IoT applications that follows the closed-loop OODA cycle. The applications are based on common IoT patterns for data pre-processing, statistical summarization, and predictive analytics. These are coupled with workloads sourced from real IoT observations. A high-level overview of the logical interaction of the IoT applications is depicted in Figure 1.

Extract-Transform-Load (ETL) consumes data from hundreds of thousands of edge sensors, and pre-processes, cleans, and archives the data. Further, the results are published to an edge broker so that clients interested in real-time monitoring can subscribe to it, while a copy is forked to the cloud for storage, and another to the next dataflow step.

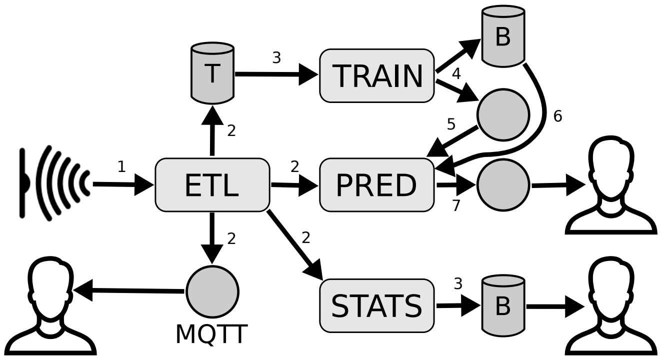

Fig. 1: RIoTBench IoT high-level logical interactions between different sensors, applications and users.

Statistical Summarization (STATS) performs higher order aggregation and plotting operations, and stores the generated plots into the cloud, from where webpages can load the visualization files on browsers.

Model Training (TRAIN) periodically loads the stored data from ETL step and trains forecasting models that are stored in the cloud, and notifies the message broker of an updated model being available.

The Predictive Analytics (PRED) subscribe to the message broker and downloads the new models from the cloud, and continuously operates over the pre-processed data stream from ETL to make predictions and classifications that can indicate actions to be taken on the domain. It then notifies the message broker of the predictions, which can independently be subscribed to by a user or device for action.

The ETL dataflow requires a low-latency cycle in order to achieve real-time monitoring, in addition it also requires some of its operators to be located in the cloud for storing messages and others to be at the edge of the network. This makes the ETL workflow the perfect candidate workflow for testing the operator placement strategy proposed.

III. Related Work

A. Operator Placement Problem

IoT applications have been deployed in multiple data centers and/or at the edge of the network (*i.e.*, edge and fog computing). Existing solutions introduce architectures which place certain operators on micro data centers located closer to where the data is generated [12] or employ mobile devices for processing IoT applications. The problem with placing IoT applications onto heterogeneous hardware is at least NP-Hard as shown by Benoit *et al.* [3]. To simplify the placement problem, communication is often neglected [10], although it is a relevant cost in geo-distributed infrastructure [19]. Likewise, the operator behavior and requirements are oversimplified using static splitting decisions as proposed by Sajjad *et al.* [9].

Meanwhile, many efforts have been made towards modeling the placement problem of IoT application on heterogeneous infrastructure [13] using Petri nets to formalize the application regions and the multiple response times that they produce.

On the other hand, Eidenbenz *et al.* [20] evaluated Series-Parallel-Decomposable Graphs (SPDG) from its parallelism degree to decompose the application graph and by an approximation algorithm to determine the placement. Cloud and edge have been explored to supply application requirements. For instance, Taneja *et. al.* [21] offer a naive approach deploying the application graph across cloud and edge using a constraint-sensitive approach.

Most of the existing work investigates solutions to improve the end-to-end latency, whereas Cardellini *et al.* [22] and Chen *et al.* [23] consider the monetary cost. The former approach introduces a model considering monetary price as the cost per unit of data transmitted along the network path between two machines. The latter covers prices for the IoT application placement using VMs, however, they create a uniform pricing model to VMs given that the cost of electricity, infrastructure, personnel, and taxes are similar within the same region.

This work considers operator placement of IoT applications across edge and cloud, and the restrictions created by their interactions. The target scenario includes real-time analytics, and our solution exploits the construction of the paths to optimize the end-to-end application latency, bandwidth consumption, and monetary costs by decomposing the dataflow and defining the operator's placement dynamically across edge and cloud. Furthermore, this work introduces a new model to control data transfers across the edge and cloud leading to monetary costs by applying the cost model of two major actors, AWS and Microsoft.

B. Programming model

Most of the existing programming models focus on supporting batch and real-time data processing efficiently in a cloud environment. For instance, MapReduce has become the *de facto* standard for batch data processing in the Apache Hadoop framework [24]. Apache Spark [25] is a distributed batch-processing framework but it also supports stream processing based on micro-batching. Other frameworks involve Apache Storm [26], which supports event-based stream processing, and Apache Flink [27], which enables batch and stream processing with its unified APIs. All of these frameworks are tailored to the cloud environment. Furthermore, none of these programming models allow developers the ability to decide **how** to split a dataflow, by specifying a set of constraints and automatically split the dataflow between the edge and the cloud of the network.

IV. Problem Description

We focus on three performance metrics for placing Internet of Things (IoT) applications across edge and cloud resources, *i.e.*, the end-to-end application latency [28], the WAN traffic, and the messaging cost (messages exchanged between the edge and the cloud). The IoT operator placement problem consists of defining how to accommodate the application components (*i.e.*, operators) on the available resources of the network topology to optimize one or more performance metrics.

Table I summarizes the notation used throughout the paper.

TABLE I: Main notation adopted for the problem description.

Symbol	Description
$\mathcal{R}$	Set of cloud and edge resources
$\mathcal{L}$	Set of network links
$i \leftrightarrow j$	A link connecting resources i and j
cpu_i^r, mem_i^r	CPU and memory capacities of resource i
$lat_{i \leftrightarrow j}$,$bdw_{i \leftrightarrow j}$	Latency and bandwidth of link $i \leftrightarrow j$
$\mathcal{O}$	Set of stream processing operators
$\mathcal{S}$	Set of event streams between operators
f_i	Function to determine if the operator is a source, sink and transformation
cpu_i^o, mem_i^o	CPU and memory req. of operator i
ψ_i^o	Selectivity of operator i
ω_i^o	Data compression rate of operator i
$s_{i \to j}^{\rho}$	Probability that a message emitted by operator i will flow to j
$\lambda_i^{in}, \lambda_i^{out}$	Input/output event rate of operator i
$\varsigma_i^{in}, \varsigma_i^{out}$	Input/output event size of operator i
$stime_{\langle i,k \rangle}$	Service time of operator i at resource k
$ctime_{\langle i,k \rangle \langle j,l \rangle}$	Communication time from operator i at resource k to j at l
$mem_{\langle i,k \rangle}$	Overall memory required by operator i when deployed at resource k
p_i, l_{p_i}	A graph path and its end-to-end latency
$\mathcal{P}$	The set of all paths in an application graph
$\mu_{\langle i,k \rangle}$	The rate at which operator i can process events at resource k

We define a computational resource (*i.e.*, cloud server or edge device) as a triple $r_k = \langle cpu_k^r, mem_k^r, f_k^r \rangle \in \mathcal{R}$, where cpu_k^r is the CPU capability in Millions of Instructions per Second (MIPS), mem_k^r is the memory capability in bytes, and $f_k^r \in \{0, 1\}$ signals whether r_k is a *cloud* resource. Similarly, the network link is drawn as a triple $l_{k \leftrightarrow l} = \langle bdw_{k \leftrightarrow l}, lat_{k \leftrightarrow l}, f_{k \leftrightarrow l} \rangle \in \mathcal{L}$, where $k \leftrightarrow l$ represents the interconnection between resource k and l, $bdw_{k \leftrightarrow l}$ the bandwidth capability in bits per second (bps), $lat_{k \leftrightarrow l}$ the latency in seconds, and $f_{k \leftrightarrow l}$ signals whether the link is part of a WAN. We consider the latency of a resource k to itself (*i.e* $lat_{k \leftrightarrow k}$) to be 0.

Each operator of the IoT application is a quintuple $o_i = \langle cpu_i^o, mem_i^o, \psi_i^o, \omega_i^o, f_i \rangle \in \mathcal{O}$, where cpu_i^o is the CPU requirement in Instructions per Second (IPS) to handle an individual event, mem_i^o is the memory requirement in bytes to load the operator, ψ_i^o is the ratio of number of input events to output events (*i.e.*, selectivity), ω_i^o is the ratio of the size of input events to the size of output events (*i.e.*, data compression/-expansion factor), and $f_i \in \{source, sink, transformation\}$ signals whether o_i is a *source*, *sink/output*, or *transformation*. The rate at which operator i can process events at resource k is denoted by $\mu_{\langle i,k \rangle}$ and is essentially $\mu_{\langle i,k \rangle} = cpu_k^r \div cpu_i^o$. An event stream $s_{k \to l}^{\rho} \in \mathcal{S}$ connects operator k to l with a probability ρ that an output event emitted by k will flow through to l.

The rate at which operator i produces events is denoted by λ_i^{out} and is a product of its input event rate λ_i^{in} and its selectivity (ψ_i^o). The output event rate of a source operator ($f_k = source$) and depends on the number of measurements

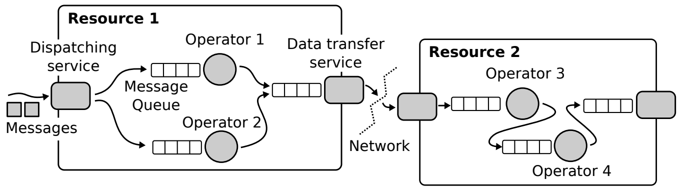

Fig. 2: Example of four operators and their respective queues placed on two resources.

it takes from a sensor or another monitored device. Likewise, we can recursively compute the average size ς_i^{in} of events that arrive at a downstream operator i and the size of events it emits ς_i^{out} by considering the upstream operators' event sizes and their respective compression/expansion factors (*i.e.*, ω_i^o).

A computational resource can host one or more operators; operators within a same host communicate directly whereas inter-node communication is done via a communication service as depicted in Figure 2. Events are handled in a First-Come, First-Served (FCFS) fashion both by operators and the communication service that serialises messages to be sent to another host. Both operators and the communication service follow an M/M/1 model for their queues which allows for estimating the waiting and service times for computation and communication. The computation or service time $stime_{\langle o_i, r_k\rangle}$ of an operator i placed on a resource k is hence given by:

$$stime_{\langle i,k\rangle} = \frac{1}{\mu_{\langle i,k\rangle} - \lambda_i^{in}} \tag{1}$$

while the communication time $ctime_{\langle i,k\rangle\langle j,l\rangle}$ for operator i placed on a resource k to send a message to operator j on a resource l is:

$$ctime_{\langle i,k\rangle\langle j,l\rangle} = \frac{1}{\left(\frac{bdw_{k\leftrightarrow l}}{\varsigma_i^{out}}\right) - \lambda_j^{in}} + l_{k\leftrightarrow l} \tag{2}$$

A mapping function $\mathcal{M} : \mathcal{O} \rightarrow \mathcal{R}, \mathcal{S} \rightarrow \mathcal{L}$ indicates the resource to which an operator is assigned and the link to which a stream is mapped. The function $mo_{\langle i,k\rangle}$ returns 1 if operator i is placed on resource k and 0 otherwise. Likewise, the function $ms_{\langle i\rightarrow j, k\leftrightarrow l\rangle}$ returns 1 when the stream between operators i and j has been assigned to the link between resources k and l, and 0 otherwise.

A *path* in the IoT application graph is a sequence of operators from a source to a sink. A path p_i of length n is a sequence of n operators and $n-1$ streams, starting at a source and ending at a sink:

$$p_i = o_0, o_1, \ldots, o_k, o_{k+1}, \ldots, o_{n-1}, o_n \tag{3}$$

Where $o_0 = source$ and $o_n = sink$. The set of all possible paths in the application graph is denoted by $\mathcal{P}$. The end-to-end latency of a path comprises the sum of the computation time of all operators along the path and the communication time required to stream events on the path. More formally, the end-to-end latency of path p_i, denoted by L_{p_i}, is:

$$\begin{aligned} L_{p_i} = &\sum_{\substack{o\in\mathcal{O}\\ r\in\mathcal{R}}} mo_{\langle o,r\rangle} \times stime_{\langle o,r\rangle} \\ &+ \sum_{r'\in\mathcal{R}} ms_{\langle o\rightarrow o+1, r\leftrightarrow r'\rangle} \times ctime_{\langle o,r\rangle\langle o+1,r'\rangle} \end{aligned} \tag{4}$$

The WAN traffic accumulates the sizes of messages that cross the WAN network where $\mathbb{1}_{\{f_{k\leftrightarrow l}=1\}}$ is the indicator that the link between the resource k and l is on a WAN. The WAN traffic of path p_i is calculated as:

$$W_{p_i} = \sum_{\substack{s_{i\rightarrow j}\in\mathcal{S}\\ k\leftrightarrow l\in\mathcal{L}}} \mathbb{1}_{\{f_{k\leftrightarrow l}=1\}} \times ms_{\langle i\rightarrow j, k\leftrightarrow l\rangle} \times \varsigma_i^{out} \tag{5}$$

Likewise, the messaging cost is calculated by the number of messages that reaches the cloud from the edge and vice versa. The indicator $\mathbb{1}_{\{f^r_{p_{i-1}}=0 \text{ and } f^r_{p_i}=1\}}$ indicates that the previous operator p_{i-1} is placed on edge ($f^r_{p_{i-1}} = 0$) and it sends messages to operator p_i in cloud ($f^r_{p_i} = 1$). The number of messages in p_i is given as:

$$C_{p_i} = \sum_{\substack{o\in\mathcal{O}\\ o'\in\mathcal{O}\\ r\in\mathcal{R}}} \mathbb{1}_{\{f^r_{o'}=0 \text{ and } f^r_o=1\}} \times mo_{\langle o,r\rangle} \times \lambda_o^{in} \tag{6}$$

The parameters latency (Par_{lat}), WAN traffic (Par_{wan}), and monetary cost (Par_{cost}) receive the current values of the running application. A single aggregate cost metric uses the parameters and *Simple Additive Weighting method* [29] (normalized in the interval [0,1]) offers a unified metric where w_l, w_w and w_c, with $w_l + w_w + w_c = 1$, are non-negative weights for the different costs. Each metric of path p_i is divided by its corresponding parameters and is then multiplied by its weight. The sum of the three metrics in the path p_i results in the aggregate cost. Formally, the $AggregateCost$ in p_i is determined as:

$$\begin{aligned} AggregateCost_{p_i} = w_l \times \frac{L_{p_i}}{Par_{lat}} + \\ w_w \times \frac{W_{p_i}}{Par_{wan}} + w_c \times \frac{C_{p_i}}{Par_{cost}} \end{aligned} \tag{7}$$

The problem of placing a distributed IoT application consists of finding a mapping that minimizes the aggregate cost.

$$\min \sum_{p_i\in\mathcal{P}} AggregateCost_{p_i} \tag{8}$$

Subject to:

$$\lambda_o^{in} < \mu_{\langle o,r\rangle} \qquad \forall o\in\mathcal{O}, \forall r\in\mathcal{R} | mo_{\langle o,r\rangle} = 1 \tag{9}$$

$$\lambda_o^{in} < \left(\frac{bdw_{k\leftrightarrow n}}{\varsigma_{o-1}^{out}}\right) \qquad \forall o\in\mathcal{O}, \forall k\leftrightarrow n\in\mathcal{L} | mo_{\langle o,k\rangle} = 1 \tag{10}$$

$$\sum_{o\in\mathcal{O}} mo_{\langle o,r\rangle} \times \lambda_o^{in} \leq cpu_r \qquad \forall r\in\mathcal{R} \tag{11}$$

$$\sum_{o\in\mathcal{O}} mo_{\langle o,r\rangle} \times mem_{\langle o,r\rangle} \leq mem_r \qquad \forall r\in\mathcal{R} \tag{12}$$

$$\sum_{\substack{s_{i\to j}\in\mathcal{S}\\ k\leftrightarrow l\in\mathcal{L}}} ms_{\langle i\to j,k\leftrightarrow l\rangle} \times \varsigma_i^{out} \leq bwd_{k\leftrightarrow l} \qquad \forall k\leftrightarrow l\in\mathcal{L} \tag{13}$$

$$\sum_{r\in\mathcal{R}} mo_{\langle o,r\rangle} = 1 \qquad \forall o\in\mathcal{O} \tag{14}$$

$$\sum_{k\leftrightarrow l\in\mathcal{L}} ms_{\langle i\to j,k\leftrightarrow l\rangle} = 1 \qquad \forall s_{i\to j}\in\mathcal{S} \tag{15}$$

Constraint 9 guarantees that a resource can provide the service rate required by its hosted operators whereas Constraint 10 ensures that the links are not saturated. The CPU and memory requirements of operators on each host are ensured by Constraints 11 and 12 respectively. Constraint 13 guarantees the data requirements of streams placed on links. Constraints 14 and 15 ensure that an operator is not placed on more than a resource and that a stream is not placed on more than a network link respectively.

V. R-Pulsar Framework

R-Pulsar is a data analytics software stack for collecting, processing, and analyzing data at the edge and/or at the cloud. R-Pulsar has been extended to allow developers the ability to decide **how** to split the application operators between the edge and the cloud, by specifying a set of constraints.

R-Pulsar consists of the associative rendezvous programming model(AR), an abstraction for content-based decoupled interactions (interactions defined in terms of semantic profiles instead of names) and rendezvous points [30]. The rendezvous point (RP) is a node where the dataflow computations occur, and it can be a gateway located at the edge of the network or a server in the cloud. R-Pulsar uses a peer-to-peer (P2P) network to connect and communicate with all the RP nodes.

A. R-Pulsar Layers

R-Pulsar has been extended with the following three layers in order to automatically split and orchestrate dataflows between the edge and the cloud.

R-Pulsar Infrastructure Controller: Designed to act similarly to software-defined networking (SDN) controllers, this layer keeps track of the network resources available in real time. Some of the basic tasks include inventorying devices within the R-Pulsar P2P network, their capabilities, locations, and network statistics.

R-Pulsar Plan Finder: This layer computes the most optimized operator placement plan. It uses a three-step approach for calculating the optimal operator placement plan for deploying dataflows between the edge and the cloud. Section V-C presents the three-step operator placement strategy developed for R-Pulsar.

R-Pulsar Executor/Monitor: The primary responsibility is to monitor dataflows running on the R-Pulsar P2P network, including dataflow deployment, task assignment, and task reassignment in case of failure.

B. R-Pulsar Nodes

Each rendezvous point (RP) in the R-Pulsar P2P network can be elected as a master or as a worker. R-Pulsar differs from other master/slave clusters such as Apache Storm [26] in the sense that R-Pulsar master and worker node roles are assigned dynamically every time a dataflow is deployed.

Master RP: The master RP's primary responsibility is to manage, coordinate, and monitor a dataflow running on the R-Pulsar P2P network, including dataflow deployment, task assignment, and task reassignment in the event of a failure. Each time a new dataflow is deployed in the P2P network a new master RP for that dataflow is elected.

Deploying a topology to the R-Pulsar P2P network involves submitting the pre-packaged dataflow file along with topology configuration. Then the information will be routed to the responsible RP using the content-based interactions [30]. The content-based interactions allow users to route dataflows to unknown RPs; the RP who receives the message will be automatically elected as the master RP for that dataflow. Once the master RP has been elected, it then uses the infrastructure controller layer to collect the network information of all the worker RPs. That information is then passed to the operator placement algorithm to generate a placement strategy. Once the operator placement algorithm has an efficient operator placement plan, then the master RP distributes the tasks to the worker RPs.

The master RP tracks the status of all worker nodes and the tasks assigned to each one. If the master RP detects that a specific worker node has failed to heartbeat or has become unavailable, it will reassign that worker RP tasks to other worker RP nodes in the federation.

The master RP is not a single point of failure in the strictest sense. This quality is because the master RP does not take part in the dataflow data processing, rather it merely manages the deployment, task assignment, and monitoring of the dataflow. In fact, if the master RP dies while a dataflow will continue to process data as long as the worker RPs assigned with tasks remain healthy.

Worker RP: Each worker node is responsible for creating, starting, and stopping worker tasks assigned to that node. Worker RPs are also responsible for once the master RP has died to perform a master RP election.

C. Placement Strategy

The strategy for operator placement on R-Pulsar applies statistics collected by profiling the application and the location of sinks and sources. The operator placement aims to minimize the AggregateCost (Equation 8) by splitting the IoT application across edge and cloud by considering priorities of operators according to the infrastructure to which the sinks are assigned. The operator placement strategy comprises three phases: (i) application profiling; (ii) candidate placement and (iii) final placement.

Phase 1 – Application Profiling:. In the first phase the worker RPs and the master RPs using the infrastructure

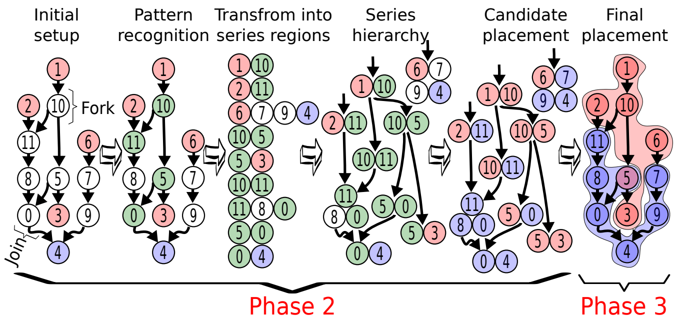

Fig. 3: Phases to determine the final placement using split points, where red means placed on edge, blue represents placed on cloud, and green delimits forks and joins.

controller layer continuously collects statistics [31] from the running dataflow. The collected data includes the following information about the operators:

- The arrival rate of events.
- Processing time per event.
- Number of MIPS required to process a tuple.
- Memory to run the operator.
- Arrival message size.
- Outcome message size.

This information is used to establish the selectivity, data compression/expansion factor, as well as, the CPU and memory requirements.

Phase 2 – Candidate Placement: In phase two, the user-predefined locations of sinks and sources are used to identify patterns in the dataflow (Section V-D). As depicted in Figure 3, a dataflow can comprise multiple patterns such as (i) forks, where messages can be replicated to multiple downstream operators or scheduled to downstream operators in a round-robin fashion, using message key hashes, or considering other criteria [13]; (ii) parallel regions that perform the same operations over different sets of messages or where each individual region executes a given set of operations over replicas of the incoming messages; and (iii) joins, which merge the outcome of parallel regions.

We consider that an IoT dataflow is a Series-Parallel-Decomposable Graphs which either consists of a series of linearly dependent operators, or operators that can be executed independently in parallel, or a combination thereof. Phase 2 uses related techniques to identify graph regions that present these patterns [20]. This information is used to build a hierarchy of region dependencies (*i.e.* downstream and upstream relations between regions) and assist in placing operators across cloud and edge resources. The streams in the graph paths that separate the operators are hereafter called the *split points*. Figure 3 illustrates the phases of the method to determine the split points (green circles), where red circles represent operators placed on edge resources whereas blue ones are on the cloud: (i) The method starts with sources and sinks whose placements are predefined by the user; (ii) split points are discovered (green circles) as well as sinks that correspond to actuators that can be placed on the edge; (iii) the branches between the existing patterns (green, red, and blue circles) are transformed into series regions; (iv) a hierarchy following the dependencies between regions is created; and (v) the regions provide information to split the operators on edge candidate placement evaluating if the operator flows events to actuators.

Algorithm 1 describes the function $GetCandidates$ used to identify the patterns and obtain the series regions. First, the function adds two virtual vertices to the graph: $virt\_src$ connected to all data sources and $virt\_sink$ to which all sinks are connected (line 2-4). The virtual vertices allow for recognizing all paths between sources and sinks. Second, each path is iterated moving operators to a temporary vector and classifying the operators as upstream and downstream according to the number of input and output edges (lines 5-8). If the operator is a split point, the temporary vector is converted into a subset of regions set, and the temporary vector receives the current operator (lines 9-10). Third, the function removes the redundant values (line 11). Fourth, the region set is iterated comparing the regions by the first and the last position values (equal values represent a connection) and consequently, they are stored in the hierarchy set (lines 12-16). At last, using the hierarchy and the placement of the sinks, the function evaluates if the operator flows events to sinks placed on edge device then the operators is added to the $candidate$ lines 17-23).

Phase 3 – Final Placement: Once phase two has completed and the profiling phase and has established the requirements from the different operators, an operator placement strategy is created and deployed. The strategy reduces the combinatorial space by estimating only once the computation (Equation 1) and communication (Equation 2) overheads to operators targeted to cloud (Phase 2). Otherwise, operators to edge have their overheads estimated for all edge devices evaluating their constraints. The strategy gives high priority to edge since cloud sinks often stores messages for batch processing, whereas the edge side hosts actuators. If edge devices cannot meet all operator requirements then the operator is moved to the cloud, hence, the cloud hosts its operator candidates and those that do not meet the constraints on edge. For instance, Operator 5 in Figure 3 was reallocated since the edge does not respect the resource constraints.

D. R-Pulsar API

In this section we present the API examples uses for evaluating and deciding **how** to split the ETL dataflow presented in section II, between the edge and the cloud resources.

Listing 1 is for specifying the operator constraints. In our case some of the operators need to be placed at the cloud and some others need to be placed at the edge. Note that if the wildcard or no placement is specified R-Pulsar will automatically decide the best placement for the operator. CloudTableInsert, MQTTPublish, and BloomFilterTask are tasks used in the ETL dataflow.

Algorithm 1: Algorithm to get the candidate placement.

```
1  Function GetCandidates($\mathcal{G} = (\mathcal{O}, \mathcal{S})$)
2      $\mathcal{O} \leftarrow \mathcal{O} \cup virt\_src \cup virt\_sink$
3      $\mathcal{S} \leftarrow \mathcal{S} \cup s_{virt\_src \rightarrow o}, \forall o \in \mathcal{O}$ and $f_o$ = source
4      $\mathcal{S} \leftarrow \mathcal{S} \cup s_{o \rightarrow virt\_sink}, \forall o \in \mathcal{O}$ and $f_o$ = sink
5      for $p \in$ GetAllPaths($\mathcal{G}, virt\_src, virt\_sink$) do
6          for $o \in p$ do
7              $temp \leftarrow temp \cup \{o\}, \forall o \notin \{virt\_src, virt\_sink\}$
8              $ups \leftarrow |\langle *, o \rangle \subset \mathcal{S}|, downs \leftarrow |\langle o, * \rangle \subset \mathcal{S}|$
9              if $ups > 1$ or $downs > 1$ and $o \notin \{virt\_src, virt\_sink\}$ then
10                 $regions \leftarrow regions \cup temp$,
                   $temp \leftarrow \{o\}$
11     Delete duplicate $regions$
12     for $src \in regions$ do
13         for $dst \in regions$ do
14             if $src \neq dst$ then
15                 if $src[|src| - 1] = dst[0]$ then
16                     $hierarchy \leftarrow hierarchy \cup \{src, dst\}$
17     for $operators \in regions$ do
18         for $o \in operators$ do
19             if $f_o \notin \{source, sink\}$ then
20                 for $sink \in$ GetSinks($o$) do
21                     if GetLocation($sink$) = $edge$ then
22                         $candidate = candidate \cup o$
23                         Break
24     return $candidate$
```

```
op1.map(CloudTableInsert()).placement(cloud);
op2.map(MQTTPublish()).placement(edge);
op3.map(BloomFilterTask()).placement(*);
```

Listing 1: User specified operator physical placements constraints.

Listing 2 is for specifying the optimizations to apply to the dataflow. The R-Pulsar operator placement algorithm offers three optimizations: minimize end-to-end latency, bandwidth, or messaging cost. Each of the functions requires a weight normalized in the interval [0,1]; the sum of all three weights must be one. By doing so, users have the ability to optimize the latency, data transfer rate and messaging cost at the same time.

```
topology.minEndToEndLatency(0.4);
topology.minDataTransferRate(0.3);
topology.minMessagingCost(0.3);
```

Listing 2: User specified dataflow optimizations (latency, data transfer rate and cost).

By specifying physical dataflow constraints and the optimizations desired R-Pulsar can obtain an optimal operator placement plan.

VI. Evaluation

This section presents an experimental evaluation of our system. First, we present the setup and the other approaches in which the experiments will be evaluated and compared against. Second, we present an evaluation of our system based on latency, data transfer rate, and messaging cost.

A. Setup

Our experiments are performed using the following edge and cloud setup:

- We used an experimental edge testbed developed by the authors, inspired by Hu et. al. [32] that consists of 13 Raspberry Pis; 5 Raspberry Pis model 3 (4x ARM Cortex-A53 1.2GHz, 1GB of RAM and 10/100 Ethernet), and 8 Raspberry Pis model 2 (4x ARM Cortex-A7 900MHz, 1GB of RAM and 100 Ethernet).
- For the cloud we used the Chameleon cloud [33] with 5 instances of type m1.medium (2 CPU and 4 GB RAM).

The 13 Raspberry Pis are connected to the same LAN. The Raspberry Pis use the external WAN [34] (the Internet) for connecting to cloud. The LAN has a latency 0.523 ms and a bandwidth of 15 Mbits/sec. The WAN has latency 66.75 ms, and bandwidth of 87.0 Mbits/sec.

In addition to the setup, each of our experiments is evaluated using three other strategies. We compared our system with the following approaches:

- **Cloud:** deploys all operators in the cloud, apart from operators provided in the initial placement.
- **LB (Taneja et. al. [21]):** iterate a vector containing the application operators, gets the middle host of the computational vector, and evaluates CPU, memory, and bandwidth constraints to obtain the operator placement.
- **Random:** simulates the user trying to guess the best placement for the dataflow between the edge and the cloud. Random is the average of 15 different dataflow deployments between the edge and the cloud resources.

All the tests are evaluated using the ETL dataflow. The ETL dataflow is an implementation of the ETL RIoTBench topology, it consists of: a single data source outputting data every 5 seconds, 2 sinks one located at the edge and one located at the cloud, and 7 tasks that need to be deployed between the edge and the cloud of the network. The experiments were conducted using Sense Your City dataset[1] which consists of transmitting data each minute from sensors in 7 cities across 3 continents, with about 12 sensors per city. The data content includes metadata on the sensor ID, geolocation, and five timestamped observations (outdoor temperature, humidity, ambient light, dust, and air quality).

[1]http://map.datacanvas.org

B. End-to-end Tuple Latency Evaluation

The end-to-end tuple latency corresponds to the sum of the mean times from the two paths in ETL dataflow (cloud and edge). The conducted experiment evaluates the end-to-end tuple latency using Equation 7 where w_l is equal to 1, and w_w and w_c are equal to 0. The experiment aims to evaluate how efficient the cloud, Random, and LB approaches are at minimizing the end-to-end tuple latency and compare the R-Pulsar operator placement approach approach. In addition, three failures were manually injected to showcase the dynamicity and flexibility to recover from node failures. The first failure makes 38% of the edge cluster unavailable (100 ms). The second failure affects the remaining 62% of the nodes (300 ms). Before the 62% of the nodes fail, the 38% of the nodes are back online. The third and last failure affects 50% of the cloud instances (505 ms).

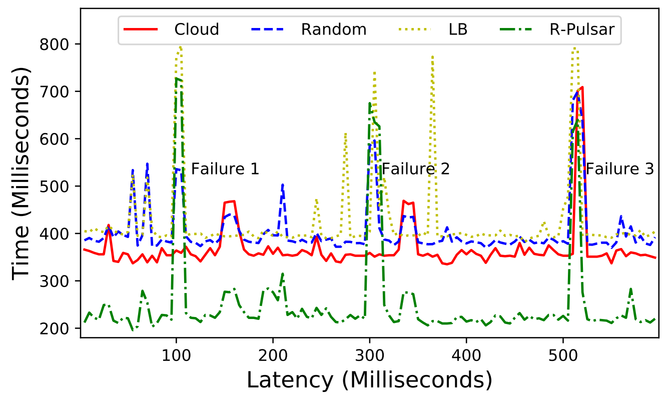

Fig. 4: End-to-end tuple latency optimization with 3 self injected failures affecting edge and cloud nodes, while comparing it with Cloud, Random and LB approaches.

Figure 4 shows that on average tuples are computed 31% faster when compared to the traditional cloud setup, and 38% faster than Random and the LB placement approaches. The reason why the Random failures recover much faster than LB when compared to R-Pulsar is because Random is the average of multiple different deployments and in some cases the first failure is not affected. Figure 4 demonstrates that R-Pulsar operator placement strategy is capable of splitting the dataflow efficiently between the edge and the cloud and reduce the end-to-end tuple latency.

The second experiment aims to evaluate how efficient the Cloud, Random, and LB approaches are at minimizing end-to-end tuple latency and compare it with R-Pulsar approach. In this experiment no failures were injected.

Figure 5 shows that when R-Pulsar operator placement approach is used 80% of the tuples see a reduction in the end-to-end tuple latency by 44% compared to the LB and Random approaches and 38% compared to the cloud approach.

C. Data Transfer Rate Evaluation

The Data transfer rate consists of the sum of all message sizes that traverse a WAN link per second. The values for

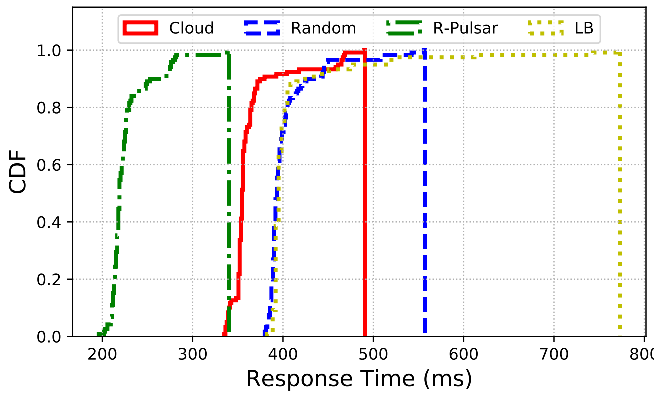

Fig. 5: End-to-end tuple latency optimization cumulative distribution function (CDF) comparison with Cloud, Random and LB approaches.

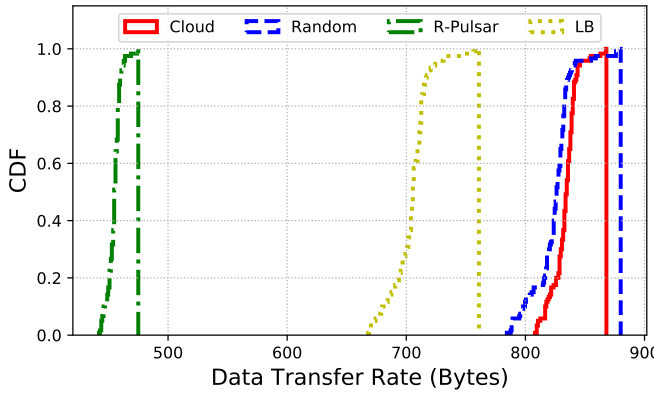

Fig. 6: End-to-end data transfer rate optimization cumulative distribution function (CDF) comparison with cloud, Random and LB approaches.

Equation 7 are w_w equal to 1, and w_l and w_c equal to 0. This third experiment aims to evaluate how efficient are the cloud, Random, and LB approaches at minimizing the transfer rate between the edge and the cloud and compare the results with R-Pulsar operator placement approach. Minimizing the transfer rate between the edge and the cloud is a critical point in order to achieve real-time analytics.

Figure 6 shows that 80% of the time R-Pulsar reduces the transfer rate between the edge and the cloud on average 35% when compared to the LB approach. And it reduces the data transfer rate by 45% when compared to the cloud and Random approaches.

This next experiment aims to evaluate the efficiency of minimizing the transfer rate and the end-to-end latency at the same time (w_w = .5, w_l = .5, and w_c = 0). This experiment was also carried out using the cloud, Random, and LB approaches.

Figure 7 shows that the R-Pulsar operator placement approach can also optimize the data transfer rate and the end-to-end latency by 46% and 38% respectively when compared to the cloud, 36% and 45% respectively when compared to the LB, 38% and 44% respectively when compared to the Random

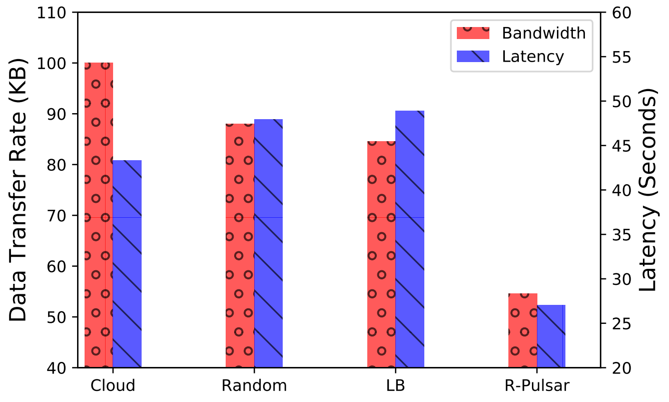

Fig. 7: Multi optimization evaluation, end-to-end tuple latency and data transfer rate comparison with cloud, Random, and LB approaches.

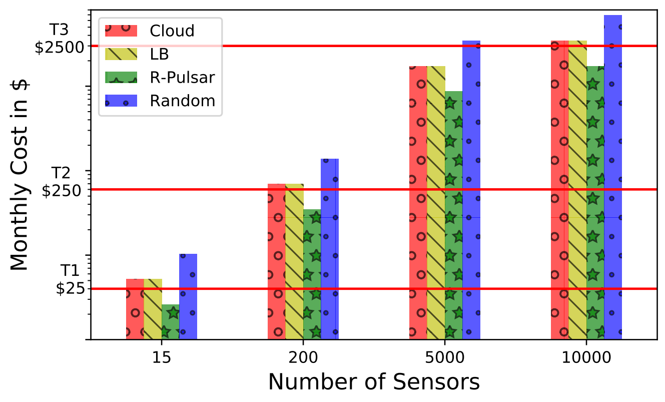

Fig. 8: Messaging cost savings evaluation based on the Microsoft Azure IoT Hub pricing model, for four different setups.

approach.

D. *Messaging Cost Evaluation*

The last two experiments aim to calculate the messaging cost of running the dataflow for a month using the cost models of two major actors, AWS and Microsoft, in a real life edge and cloud scenario. For this reason, we setup Equation 7 with w_c equal to 1, and w_l and w_l equal to 0. The goal of this optimization is to reduce the number of messages that reach the cloud servers.

TABLE II: Azure IoT Hub and Amazon IoT Core messaging pricing.

Microsoft IoT Hub Pricing	AWS IoT Core Pricing
Free Tier - 8,000 messages/day $0	Every 1 million messages/day $1.00
Tier 1- 400,000 messages/day $25	Up to 1 billion messages/day $1.00
Tier 2 - 6,000,000 messages/day $250	Next 4 billion messages/day $0.80
Tier 3 - 300,000,000 messages/day $2,500	Over 5 billion messages/day $0.70

Table II depicts two IoT cost models. The first cost model is the Microsoft Azure IoT Hub [35]. Each tier enables a maximum number of messages exchanged between the Azure IoT Edge and the Azure IoT Hub and vice versa per day. T1 allows up to 400,000 messages a day, T2 allows up to 6,000,000 messages a day, and T3 allows up to 300,000,000 messages a day.

The second cost model is the Amazon IoT Core [36] where messaging is metered by the number of messages transmitted between your devices and AWS IoT Core and vice versa per day. Amazon offers multiple costs for different regions, for this experiment we choose the cheapest region (N.Virginia) which charges $1 per million messages sent, and the cost per message decreases after the first 1 billion messages per day.

Figure 8 depicts the cost of deploying the ETL dataflow using the Microsoft cost model using the four different approaches presented earlier. When using a small setup (15 sensors), the monthly cost for our system will be $25 a month while the cloud, LB, or Random approaches will cost $250 a month, savings of 90%. A similar behavior happens with a medium (200 sensors) and extra large (10000 sensors) setups.

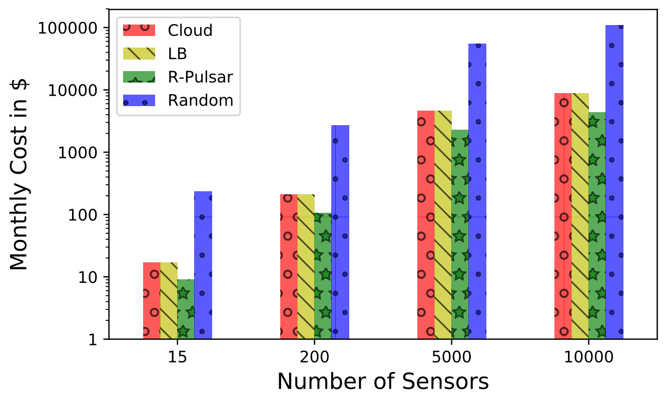

Fig. 9: Messaging cost savings evaluation based on the Amazon IoT pricing model, for four different setups.

Figure 9 depicts the cost of deploying the ETL dataflow using the Amazon cost model. Our system obtains a 50% cost reduction when compared to the cloud and LB approaches in all four different setups (15, 200, 5000 and 10000 sensors). In addition our system obtains a 97% savings when compared to the Random approach in all four different setups.

VII. Conclusion & Future work

This paper presents a framework for solving the operator placement problem of Internet of Things dataflows among edge and cloud resources. Authors proposed a new operator placement strategy to reduce end-to-end latencies, data transfer rates and messaging costs between the edge and the cloud, by using a three phase additive weighted model. We implemented the operator placement strategy in R-Pulsar, in the form of a programming model for specifying **how** IoT applications will be split across the edge and the cloud.

Evaluation of this work was performed using an experimental testbed comprised of edge and cloud resources to deploy and execute a real-world IoT applications. The operator placement approach was evaluated against three other strategies from the litterature, showing the ability of the framework to efficiently place the computations between the edge and the core of the network. Results show that the system is capable of reducing the end-to-end latency by at most 45%. In addition, our system can minimize the data transfer rate and reduce the number of messages exchanged.

As for future work, we intend to extend this approach by considering the power consumption of operators to achieve energy and performance trade-offs when taking placement decisions.

VIII. Acknowledgements

This work was performed within the framework of the LABEX MILYON (ANR-10-LABX-0070) of the University of Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007). This work was partly funded by SUSTAM through the Inria Associate Teams program and NSF via grant number OCE 1745246. SUSTAM is an associate Team between Inria (ENS Lyon) and RDI^2 (Rutgers University).

References

[1] "Data Our New Natural Resource - https://www.demandcaster.com/blog-news/data-our-new-resource/."

[2] M. AbdelBaky, M. Zou, A. R. Zamani, E. G. Renart, J. D. Montes, and M. Parashar, "Computing in the continuum: Combining pervasive devices and services to support data-driven applications," *2017 IEEE 37th Int. Conf. on Dstb Comp. Systems*, 2017.

[3] A. Benoit, A. Dobrila, J.-M. Nicod, and L. Philippe, "Scheduling linear chain streaming applications on heterogeneous systems with failures," *Future Gener. Comput. Syst.*, vol. 29, July 2013.

[4] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Net. and Computer Applications*, vol. 103, 2018.

[5] O. Runsewe and N. Samaan, "Cloud resource scaling for big data streaming applications using a layered multi-dimensional hidden markov model," in *Proc. of the 17th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, (Piscataway, NJ, USA), IEEE Press, 2017.

[6] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *10th ACM Int. Conf. on Dstb Event-Based Systems*, (New York, NY, USA), ACM, 2016.

[7] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *16th Annual Middleware Conf.*, Middleware '15, (New York, NY, USA), ACM, 2015.

[8] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: Traffic-aware online scheduling in storm," in *IEEE 34th Int. Conf. on Distributed Computing Systems (ICDCS)*, June 2014.

[9] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *2016 IEEE/ACM Symp. on Edge Comp.*, Oct 2016.

[10] B. Cheng, A. Papageorgiou, and M. Bauer, "Geelytics: Enabling on-demand edge analytics over scoped data sources," in *IEEE Int. Cong. on BigData*, 2016.

[11] C. Hochreiner, M. Vogler, P. Waibel, and S. Dustdar, "VISP: An ecosystem for elastic data stream processing for the internet of things," in *20th IEEE Int. Ent. Dstb Object Comp. Conf.*, Sept 2016.

[12] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli, "Distributed QoS-aware scheduling in Storm," in *9th ACM Int. Conf. on Dstb Event-Based Systems*, DEBS '15, (New York, USA), ACM, 2015.

[13] L. Ni, J. Zhang, C. Jiang, C. Yan, and K. Yu, "Resource allocation strategy in fog computing based on priced timed petri nets," *IEEE IoT Journal*, 2017.

[14] E. Gibert Renart, J. Diaz-Montes, and M. Parashar, "Data-driven stream processing at the edge," in *IEEE Int. Conf. on Fog and Edge Computing*, 2017.

[15] E. Gibert Renart, D. Balouek-Thomert, X. Hu, J. Gong, and M. Parashar, "Online decision-making using edge resources for content-driven stream processing," in *IEEE Int. Conf. on eScience*, 2017.

[16] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, 2017.

[17] B. Brehmer, "The dynamic ooda loop : Amalgamating boyd s ooda loop and the cybernetic approach to command and control assessment , tools and metrics," 2005.

[18] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: A real-time iot benchmark for distributed stream processing platforms," *CoRR*, vol. abs/1701.08530, 2017.

[19] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan, "Quantifying the impact of edge computing on mobile applications," in *7th ACM SIGOPS Asia-Pacific Wksp on Systems*, APSys '16, (New York, NY, USA), ACM, 2016.

[20] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *IEEE INFOCOM 2016*, April 2016.

[21] M. Taneja and A. Davy, "Resource aware placement of iot application modules in fog-cloud computing paradigm," in *IFIP/IEEE Symp. on Integrated Net. and Service Mgmt (IM)*, May 2017.

[22] V. Cardellini, F. LoPresti, M. Nardelli, and G. RussoRusso, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurrency and Computation: Practice and Experience*.

[23] W. Chen, I. Paik, and Z. Li, "Cost-aware streaming workflow allocation on geo-distributed data centers," *IEEE Transactions on Computers*, Feb 2017.

[24] "Apache Hadoop - https://hadoop.apache.org/."

[25] "Apache Spark - http://spark.apache.org/."

[26] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, Patel, *et al.*, "Storm@twitter," in *Proc. of the 2014 ACM SIGMOD Int. Conf. on Mgmt of Data*, SIGMOD '14, (New York, NY, USA), ACM, 2014.

[27] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, 2015.

[28] A. da Silva Veith, M. D. de Assuno, and L. Lefevre, "Latency-aware placement of data stream analytics on edge computing," in *16th Int. Conf. Service-Oriented Comp.*, ICSOC '18, Nov. 2018.

[29] K. Yoon, P. Yoon, C. Hwang, SAGE., and i. Sage Publications, *Multiple Attribute Decision Making: An Introduction*. Multiple Attribute Decision Making: An Introduction, SAGE Publications, 1995.

[30] E. G. Renart, D. Balouek-Thomert, and M. Parashar, "Edge based data-driven pipelines (technical report)," *CoRR*, vol. abs/1808.01353, 2018.

[31] N. Kaur and S. K. Sood, "Efficient resource management system based on 4vs of big data streams," *Big Data Res.*, 2017.

[32] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan, "Quantifying the impact of edge computing on mobile applications," in *APSys*, 2016.

[33] "Chameleon Cloud. https://www.chameleoncloud.org/."

[34] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan, "The impact of mobile multimedia applications on data center consolidation," in *IEEE Int. Conf. on Cloud Engineering (IC2E)*, March 2013.

[35] "Microsoft Azure IoT Hub Pricing - https://azure.microsoft.com/en-us/pricing/details/iot-hub/."

[36] "AWS IoT Core Pricing - https://aws.amazon.com/iot-core/pricing/."