

# Multi-Objective Reinforcement Learning for Reconfiguring Data Stream Analytics on Edge Computing

Alexandre da Silva Veith, Felipe Rodrigo de Souza, Marcos Dias de Assuncao, Laurent Lefèvre, Julio C. S. dos Anjos

▶ **To cite this version:**

Alexandre da Silva Veith, Felipe Rodrigo de Souza, Marcos Dias de Assuncao, Laurent Lefèvre, Julio C. S. dos Anjos. Multi-Objective Reinforcement Learning for Reconfiguring Data Stream Analytics on Edge Computing. ICPP 2019 - 48th International Conference on Parallel Processing, Aug 2019, Kyoto, Japan. pp.1-10, 10.1145/3337821.3337894 . hal-02140844

**HAL Id: hal-02140844**
**https://inria.hal.science/hal-02140844**

Submitted on 24 Jul 2019

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Multi-Objective Reinforcement Learning for Reconfiguring Data Stream Analytics on Edge Computing

Alexandre da Silva Veith
Felipe Rodrigo de Souza
Marcos Dias de Assunção
Laurent Lefèvre
alexandre.veith@ens-lyon.fr
felipe-rodrigo.de-souza@ens-lyon.fr
marcos.dias.de.assuncao@ens-lyon.fr
laurent.lefevre@ens-lyon.fr
Univ. Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342
Lyon, France

Julio Cesar Santos dos Anjos
Institute of Informatics - PPGC (UFRGS)
Porto Alegre/RS, Brazil
jcsanjos@inf.ufrgs.br

## ABSTRACT

There is increasing demand for handling massive amounts of data in a timely manner via Distributed Stream Processing (DSP). A DSP application is often structured as a directed graph whose vertices are operators that perform transformations over the incoming data and edges representing the data streams between operators. DSP applications are traditionally deployed on the Cloud in order to explore the virtually unlimited number of resources. Edge computing has emerged as a suitable paradigm for executing parts of DSP applications by offloading certain operators from the Cloud and placing them close to where the data is generated, hence minimising the overall time required to process data events (*i.e.*, the end-to-end latency). The operator reconfiguration consists of changing the initial placement by reassigning operators to different devices given target performance metrics. In this work, we model the operator reconfiguration as a Reinforcement Learning (RL) problem and define a multi-objective reward considering metrics regarding operator reconfiguration, and infrastructure and application improvement. Experimental results show that reconfiguration algorithms that minimise only end-to-end processing latency can have a substantial impact on WAN traffic and communication cost. The results also demonstrate that when reconfiguring operators, RL algorithms improve by over 50% the performance of the initial placement provided by state-of-the-art approaches.

## KEYWORDS

data analytics, edge computing, markov decision process, reinforcement learning, multi-objective, monte carlo tree search

**ACM Reference Format:**
Alexandre da Silva Veith, Felipe Rodrigo de Souza, Marcos Dias de Assunção, Laurent Lefèvre, and Julio Cesar Santos dos Anjos. 2019. Multi-Objective Reinforcement Learning for Reconfiguring Data Stream Analytics on Edge Computing. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337894

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ICPP 2019, August 5–8, 2019, Kyoto, Japan*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337894

## 1 INTRODUCTION

Nowadays the most diverse areas of society are pervaded by monitoring systems, sensors, and Internet of Things (IoT) devices that generate unprecedented amounts of data. In order to extract valuable information from this data in a timely manner, data events must be treated as they are generated using techniques such as Distributed Stream Processing (DSP).

A DSP application is commonly structured as a directed graph whose vertices are data sources, operators, and data sinks, whereas edges represent the data streams between operators. The application has one or multiple data sources that produce an input data stream, operators that perform transformations over the streaming data (*e.g.*, filtering, aggregation, convolution) until the data reaches a data sink [3]. DSP applications are traditionally deployed on the Cloud in order to explore its virtually unlimited number of resources. Although the Cloud deployment can achieve the scale required by many applications, it incurs substantial processing delay for services such as in IoT due to data transfer from sources located at the edges of the Internet to the Cloud. More recently, *Edge computing* emerged as a paradigm that leverages devices located at the Internet edges (*edge devices*) for carrying out processing tasks. These devices, though much more constrained than their Cloud counterparts, have non-negligible processing capacity and are often geographically closer to data sources.

For deploying DSP applications one needs to decide how to assign operator tasks to available resources, a problem referred to as *operator placement* and that is known to be NP-hard [1]. The problem is exacerbated when considering hybrid cloud-edge infrastructure with highly heterogeneous resources. The non-negligible yet limited processing capabilities of *edge devices* add another constraint to the problem. Several solutions in the literature tackle the operator placement problem [12], some of which consider edge computing [2, 4, 10, 15, 22]. Most of these solutions consider only the initial assignment of operators to resources and optimise a single performance metric such as end-to-end latency, throughput,

or monetary cost. They ignore further optimisation during the application life-cycle, and neither modify the initial assignment nor reconfigure the already deployed application to deal with events such as load changes, device failures, among other issues [3].

Reconfiguring DSP applications and deciding which operators to migrate is a computationally hard problem, specially when considering multiple Quality of Service (QoS) metrics such as end-to-end latency, traffic volume that uses WAN links, monetary cost, and the overhead posed by saving the application state and moving operators to new resources. As the infrastructure and applications grow larger, trying to devise a reconfiguration plan while optimising multiple objectives can result in a large search space.

Reinforcement Learning (RL) and Monte-Carlo Tree Search (MCTS) have been used to tackle problems with large numbers of actions and states [6, 29], performing at human-level or better in games such as Go. In the present work, we model the operator migration problem as a Markov Decision Process (MDP) and investigate the use of RL and MCTS algorithms to create reconfiguration plans that aim to improve multiple QoS metrics, including the end-to-end latency of data stream events, monetary costs, data traffic over WAN links, and reconfiguration overhead.

The contributions of this work are therefore to:

- Present a model for reconfiguring DSP applications in heterogeneous infrastructure considering a multi-objective problem optimisation (§2);
- Introduce a model for the problem using MDP and investigate the use of RL algorithms (§3); and
- Evaluate the RL algorithms considering the multi-objective solution and compare their performance against the initial placement provided by traditional and state-of-the-art approaches (§4).

## 2 THE RECONFIGURATION PROBLEM

Distributed data stream processing applications are often long-running and can experience variable load requirements that change the working conditions of operators. Unlike the Cloud, edge resources are often more constrained and less reliable, with higher failure rates. To preserve the application performance within acceptable bounds it is important to adjust the initial placement and conveniently reassign operators to available resources. Similarly to solving the DSP operator placement problem, addressing reconfiguration consists of accommodating the application operators onto the available resources in order to optimise one or multiple QoS metrics. In this section we extend a model proposed in our previous work [27] by addressing the reconfiguration of DSP applications considering multiple QoS metrics. Table 1 summarises the notation used throughout the paper.

The infrastructure is viewed as a graph $\mathcal{N} = (\mathcal{R}, \mathcal{L})$ where $\mathcal{R}$ is the union set of Cloud compute resources ($\mathcal{R}_c$) and edge resources ($\mathcal{R}_e$), and $\mathcal{L}$ is the set of logical links interconnecting the resources, comprising WAN interconnections ($\mathcal{L}_w$) and LAN links ($\mathcal{L}_l$). A computational resource is defined as a tuple $r_k = \langle cpu_k^r, mem_k^r \rangle \in \mathcal{R}$, where $cpu_k^r$ is the CPU capability in Millions of Instructions per Second (MIPS) and $mem_k^r$ is the memory capability in bytes. Similarly, a network link is a tuple $k \leftrightarrow l = \langle bdw_{k \leftrightarrow l}, lat_{k \leftrightarrow l} \rangle \in \mathcal{L}$, where $k \leftrightarrow l$ represents the interconnection between resource $k$ and $l$, $bdw_{k \leftrightarrow l}$ the bandwidth capability in bits per second (bps), and $lat_{k \leftrightarrow l}$ the latency in seconds. We consider the latency of a resource $k$ to itself (*i.e.*, $lat_{k \leftrightarrow k}$) to be 0.

**Table 1: Main notation adopted in the problem description.**

| Symbol | Description |
|---|---|
| $\mathcal{R}$ | Set of cloud ∪ edge resources |
| $\mathcal{R}_c, \mathcal{R}_e$ | Sets of cloud and edge resources |
| $\mathcal{L}$ | Set of all network links |
| $\mathcal{L}_w, \mathcal{L}_l$ | Set of WAN and LAN links |
| $\mathcal{N}, \mathcal{G}$ | Network and application graphs |
| $k \leftrightarrow l$ | A link connecting resources $k$ and $l$ |
| $cpu_k^r$, $mem_k^r$ | CPU and memory capacities of resource $k$ |
| $lat_{k \leftrightarrow l}$,$bdw_{k \leftrightarrow l}$ | Latency and bandwidth of link $k \leftrightarrow l$ |
| $\mathcal{O}$ | Set of stream processing operators |
| $\mathcal{E}$ | Set of event streams between operators |
| $cpu_i^o$, $mem_i^o$ | CPU and memory req. of operator $i$ |
| $ws_i^o$ | Length of operator $i$'s window in number of events |
| $\psi_i^o$ | Selectivity of operator $i$ |
| $\omega_i^o$ | Data compression rate of operator $i$ |
| $e_{i \to j}^{\rho}$ | Event stream with probability $\rho$ that an event 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 $k$ |
| $\mathcal{W}$ | Set of non-negative weights |
| $\mathbb{1}_{\langle i \to j, k \leftrightarrow l \rangle}$ | Indicates when the stream between operators $i$ and $j$ has been assigned to the link between resources $k$ and $l$ |
| $\mathbb{1}_{\langle i,k \rangle}$ | Indicates whether operator $i$ is placed on resource $k$ |
| $AL, D, C, W$ | QoS metrics of aggregate end-to-end latency, reconfiguration overhead, monetary cost, and WAN traffic |
| $C^m, C^c$ | Monetary cost for events and connections following the pricing policy |
| $T_{code}^i, T_{state}^i$ | Time for transferring operator $i$'s code and state |
| $\mathcal{M}$ | Reconfiguration plan |
| $M$ | Set of QoS metrics |

A DSP application is a graph $\mathcal{G} = (\mathcal{O}, \mathcal{E})$ of *operators* $\mathcal{O}$ that execute functions over the incoming data, and streams $\mathcal{E}$ of data events flowing between operators. Each operator is a tuple $o_i = \langle cpu_i^o, mem_i^o, \psi_i^o, \omega_i^o, ws_i^o \rangle \in \mathcal{O}$, where $cpu_i^o$ is the CPU requirement in MIPS to handle an individual event, $mem_i^o$ is the memory

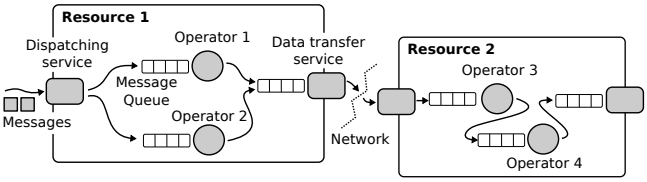



**Figure 1: Example of four operators and their respective queues placed on two resources.**

requirement in bytes to load the operator, $\psi_i^o$ is the ratio of number of input events to output events (*i.e.*, selectivity), $\omega_i^o$ is the ratio of the size of input events to the size of output events (*i.e.*, data compression/expansion factor), and $ws_i^o$ is the length of the operator's window in number of events. 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 operator can have one or multiple output streams. An event stream $e_{i \to j}^{\rho} \in \mathcal{E}$ connects operator $i$ to $j$ with a probability $\rho$ that an output event emitted by $i$ will flow through to $j$. If $e_{i \to j}^{\rho}$ is the only output stream of operator $i$, then $\rho = 1$.

The rate at which operator $i$ produces events ($\lambda_i^{out}$) is a product of its input event rate $\lambda_i^{in}$ and its selectivity $\psi_i^o$. The output event rate of a source operator depends on the number of measurements that it takes from a sensor or another monitored device. Likewise, we can recursively compute the average size $\varsigma_i^{in}$ of events that arrive at a downstream operator $i$ and the size of events it emits $\varsigma_i^{out}$ by considering the upstream operators' event sizes and their respective compression/expansion factors (*i.e.*, $\omega_i^o$).

A computational resource can host one or more operators. Operators within a same host communicate directly whereas inter-node communication occurs via a communication service as depicted in Figure 1. Events are handled in a First-Come, First-Served (FCFS) fashion both by operators and the communication service that serialises events 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. Moreover, a stateful operator can have an impact on the computation time as it waits until it receives a number of events before considering the window complete ($ws_i^o$). The computation or service time $stime_{\langle i,k \rangle}$ of an operator $i$ placed on resource $k$ is hence given by:

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

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

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

The function $\mathbb{1}_{\langle i,k \rangle}$ indicates whether operator $i$ is placed on resource $k$ whereas $\mathbb{1}_{\langle i \to j, k \leftrightarrow l \rangle}$ indicates when the stream between operators $i$ and $j$ has been assigned to the link between resources $k$ and $l$. The goal is to find a reconfiguration plan $\mathcal{M} : \mathcal{O} \to \mathcal{R}, \mathcal{E} \to \mathcal{L}$ where operators are reassigned to computational resources and streams to link(s) in a way that minimises $M = \{m_0, ..., m_k\}$ QoS metrics. We employ the Simple Additive Weighting method [31] that computes a value, hereafter termed as the aggregate cost (*agg_cost*), over normalised metric values by assigning non-negative weights $\mathcal{W} = \{w_0, ..., w_k | w_0 + ... + w_k = 1\}$ to the multiple metrics being considered. The aggregate cost is therefore:

$$agg_cost = \sum_{z \in W} w_z \times m_z, \tag{3}$$

As described further in Section 2.2, we consider four QoS metrics, namely the aggregate end-to-end latency, WAN traffic, monetary cost, and the reconfiguration overhead

## 2.1 Constraints

Edge devices are limited in terms of memory, computing, and communication capabilities. A reconfiguration mapping $\mathcal{M}$ needs to respect the following constraints:

$$\lambda_i^{in} < \mu_{\langle i,k \rangle} \qquad \forall i \in \mathcal{O}, \forall k \in \mathcal{R} | \mathbb{1}_{\langle i,k \rangle} = 1 \tag{4}$$

$$\lambda_i^{out} < \left(\frac{bdw_{k \leftrightarrow l}}{\varsigma_i^{out}}\right) \qquad \forall i \in \mathcal{O}, \forall k \leftrightarrow l \in \mathcal{L} | \mathbb{1}_{\langle i,k \rangle} = 1 \tag{5}$$

The CPU and memory requirements of operators on each host are ensured by constraints 6 and 7:

$$\sum_{i \in O} \mathbb{1}_{\langle i,k \rangle} \times \lambda_i^{in} \leq cpu_k^r \qquad \forall k \in \mathcal{R} \tag{6}$$

$$\sum_{i \in O} \mathbb{1}_{\langle i,k \rangle} \times (mem_i^o + ws_i^o \times \varsigma_i^{in}) \leq mem_k^r \qquad \forall k \in \mathcal{R} \tag{7}$$

Data requirements of streams placed on links are guaranteed by:

$$\sum_{\substack{s_{i \to j} \in \mathcal{E} \\ k \leftrightarrow l \in \mathcal{L}}} \mathbb{1}_{\langle i \to j, k \leftrightarrow l \rangle} \times \varsigma_i^{out} \leq bwd_{k \leftrightarrow l} \qquad \forall k \leftrightarrow l \in \mathcal{L} \tag{8}$$

Constraints 9 and 10 ensure that an operator is not placed on more than one resource and that a stream is not placed on more than a network link respectively:

$$\sum_{k \in \mathcal{R}} \mathbb{1}_{\langle i,k \rangle} = 1 \qquad \forall i \in \mathcal{O} \tag{9}$$

$$\sum_{k \leftrightarrow l \in \mathcal{L}} \mathbb{1}_{\langle i \to j, k \leftrightarrow l \rangle} = 1 \qquad \forall s_{i \to j} \in \mathcal{E} \tag{10}$$

## 2.2 Quality of Service Metrics

As DSP applications must handle incoming data events under short delays, the goal of the operator reconfiguration is to minimise the response time while reducing one or multiple metrics including the monetary cost, the WAN traffic and the reconfiguration overhead.

*2.2.1 Aggregate End-to-End Application Latency (End-to-End Latency).* A *path* in a DSP 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:

$$p_i = o_0, o_1, \ldots, o_k, o_{k+1}, \ldots, o_{n-1}, o_n \tag{11}$$

where $o_0$ is a source and $o_n$ is a sink. The set of all possible paths in the application graph is denoted by $\mathcal{P}$. The end-to-end latency

of a path is 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:

$$L_{p_i} = \sum_{j \in p_i, k \in \mathcal{R}} \mathbb{1}_{\langle j,k \rangle} \times stime_{\langle j,k \rangle} + \sum_{l \in \mathcal{R}} \mathbb{1}_{\langle j \to j+1, k \leftrightarrow l \rangle} \times ctime_{\langle j,k \rangle \langle j+1,l \rangle} \quad (12)$$

The aggregate end-to-end latency $AL$ is therefore given by:

$$AL = \sum_{p_i \in \mathcal{P}} L_{p_i} \quad (13)$$

*2.2.2 WAN Traffic.* WAN communication usually operates through the Internet where the lack of network guarantees and instability might introduce delay and delay-jitter when transferring data between geographically distributed sites. WAN traffic accumulates the sizes of events that cross the WAN interconnections $\mathcal{L}_w$:

$$W = \sum_{\substack{s_{i \to j} \in \mathcal{E} \\ k \leftrightarrow l \in \mathcal{L}_w}} \mathbb{1}_{\langle i \to j, k \leftrightarrow l \rangle} \times \varsigma_i^{out} \quad (14)$$

*2.2.3 Monetary Cost of Communication.* The monetary cost of event exchange is based on the main elements[1] of IoT services of two major Cloud-edge players, namely Amazon IoT Core[2] and Microsoft Azure IoT Hub[3]. The price comprises the cost of the number of connections and that of exchanging events. The price for exchanging events is calculated as the number of events that reach the cloud from the edge and vice-versa.

The first part of the cost represents events arriving from the edge to the cloud or vice-versa. The number of events exchanged between edge $\mathcal{R}_e$ and cloud $\mathcal{R}_c$ resources is given by:

$$C^m = \sum_{\substack{i \in O, j \in O \\ k \in \mathcal{R}_e, l \in \mathcal{R}_c}} \left( \mathbb{1}_{\langle i \to j, k \leftrightarrow l \rangle} \times \mathbb{1}_{\langle j,l \rangle} \times \mathbb{1}_{\langle i,k \rangle} \times \lambda_j^{in} + \mathbb{1}_{\langle j \to i, k \leftrightarrow l \rangle} \times \mathbb{1}_{\langle i,k \rangle} \times \mathbb{1}_{\langle j,l \rangle} \times \lambda_j^{out} \right) \quad (15)$$

The number of connections between edge and cloud is:

$$C^c = \sum_{\substack{i \in O, j \in O \\ k \in \mathcal{R}_e, l \in \mathcal{R}_c}} \left( \mathbb{1}_{\langle i \to j, k \leftrightarrow l \rangle} \times \mathbb{1}_{\langle j,l \rangle} \times \mathbb{1}_{\langle i,k \rangle} + \mathbb{1}_{\langle j \to i, k \leftrightarrow l \rangle} \times \mathbb{1}_{\langle i,k \rangle} \times \mathbb{1}_{\langle j,l \rangle} \right) \quad (16)$$

Hence the total cost is given by:

$$C = C^c \times price_connections + C^m \times price_events \quad (17)$$

where $price_connections$ and $price_events$ are a provider's prices for connections and events, respectively.

[1]For simplicity, we consider the two main costs: connections and events
[2]AWS IoT Core - https://aws.amazon.com/iot-core/pricing/
[3]Microsoft Azure IoT Hub - https://azure.microsoft.com/en-us/pricing/details/iot-hub/

*2.2.4 Reconfiguration Overhead.* Reconfiguring an application consists of migrating operators across compute resources to optimise the QoS metrics described above [3]. Reconfiguration here is a pause-and-resume approach which involves the following operations. First, the DSP system terminates the operator running on the current location and pauses its upstream operators to avoid emitting data towards the operator being reconfigured. Then, the operator is migrated to the new location along with its internal state in case it is stateful. Finally, the DSP system starts the new operator and resumes the application execution.

Formally, the reconfiguration overhead consists of the total downtime incurred by migrating operator code ($T^i_{code}$) and state ($T^i_{state}$), where $T^i_{code}$ and $T^i_{state}$ refer to the time required to move the $mem^o_i$ and the $ws^o_i$ of operator $i$, respectively. The migration time comprises the transfer time using an available route in the infrastructure, the sum of the link data transfers considering the bandwidth capability and their latencies. Since operator migrations happen in parallel, the total downtime is the longest migration time, denoted by:

$$D = \max_{\substack{i \in O \\ k \in \mathcal{R}}} [\mathbb{1}_{\langle i,k \rangle} \times (T^i_{code} + T^i_{state})] \quad (18)$$

# 3 REINFORCEMENT LEARNING FOR OPERATOR RECONFIGURATION

The application reconfiguration is structured as a Markov Decision Process (MDP) that represents an agent's decision-making process (*i.e.*, *DSP scheduler*) when performing the operator reassignment. The MDP maintains possible states of a simulated environment that uses the model described in Section 2 for evaluating the impact of operator migrations. The agent employs MCTS approaches to keep track of state transitions when evaluating the DSP application using the simulated environment and by applying algorithms described in this section to balance *exploration* of new solutions and *exploitation* of good and well-known ones.

## 3.1 Markov Decision Process

The MDP provides a decision-making framework where an agent makes decisions by interacting with a simulated environment over a number of steps. The MDP often comprises a set of environment states $\mathcal{S}$ including the initial state $s_0$ and a terminal state $s_{|\mathcal{S}|-1}$, where each state $s$ has a set of possible actions $\mathcal{A}(s)$ and a reward function $R(s)$. At a non-terminal state, the agent picks an available action and interacts with the simulated environment to determine the state and reward for the next step. For instance, at step $t$ in Figure 2 the system is at state $s_t$ and transitions to $s_{t+1}$. Such transition corresponds to performing a possible action $a_t$ at $s_t$; and receiving a reward $r_t$ by evaluating the transition to state $s_{t+1}$.

For building the set of possible environment states for the reconfiguration, we first create a deployment sequence $\mathcal{D}$, which consists of a sorted list of operators that need to be reassigned to resources. The sequence is built by using breadth-first search [18] to traverse the application graph, where the system gives priority to upstream operators. A state $s_t$ at time step $t$ is a tuple $s_t = \langle \mathcal{M}_t, \mathcal{R}_t, \mathcal{L}_t, d_t, c_t \rangle \in \mathcal{S}$, where $\mathcal{M}_t$ contains a mapping of operator/stream onto resource/link(s), $\mathcal{R}_t$ and $\mathcal{L}_t$ consist of the

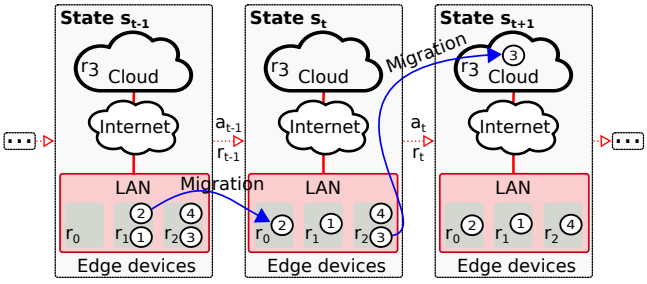



**Figure 2: Example of MDP-based operator reconfiguration.**

residual capacities of resources and links respectively, $d_t$ is an index to the operator deployment sequence $\mathcal{D}$, and $c_t \in \{0, 1\}$ indicates whether the mapping $\mathcal{M}_t$ violates a constraint (Section 2.1).

An action $a$ under state $s_t$ consists in assigning the operator referred to by $d_t$ (*i.e.*, $o_i$) to a resource $a$ such that $\mathbb{1}_{\langle i,a \rangle} = 1$. Each possible action can consist in maintaining the current mapping of operator $o_i$ or migrating it to another resource. The number of actions is equal to the number of compute resources $\mathcal{R}_t$ with enough memory to meet operator $o_i$ requirements, *i.e.*, $\mathcal{A}(s) = \{a \in \mathcal{R}_t | mem_a^r \geq mem_i^o\}$.

A transition from state $s_t$ to $s_{t+1}$ also changes the index to the deployment sequence from $d_t$ to $d_{t+1}$ where $s_t$ is a non-terminal state and $d_t < |\mathcal{D}|$ and $c_t \neq 1$. In Figure 2, which depicts the MDP-simulated deployment of the example given in Figure 1, the state $s_t$ has index $d_t$ pointing to operator 2, whereas state $s_{t+1}$ provides $d_{t+1}$ referring to operator 3. In this example, action $a_t$ is taken to reassign operator 3 to $r_3$. After taking action $a_t$ at state $s_t$, the system yields a new state $s_{t+1}$. If the new state maintains the current mapping, the agent copies all the information from $s_t$ and updates $d_{t+1}$ to consider the next operator in sequence $\mathcal{D}$. If the operator referred to by $d_t$ is migrated, the agent evaluates the expected *agg_cost* of the new state using the model of Section 2.

When simulating the operator migration while transitioning from state $s_t$ to state $s_{t+1}$, the agent updates the operator/stream mapping $\mathcal{M}_{t+1}$, the residual capacities of resources $\mathcal{R}_{t+1}$ and links $\mathcal{L}_{t+1}$, and whether constraints are violated $c_{t+1}$. Using the location of the migrated operator and the locations of upstream operators, the agent reassesses the stream mapping as it directly effects the arrival rate (*i.e.*, the network bandwidth) of the migrated operator. Since there can be multiple paths between two compute resources, the agent picks the one with the most residual bandwidth to support the volume of events emitted by the upstream operator and with the shortest latency. If the paths violate a constraint, the agent sets $c_{t+1}$ to 1 and skips the rest of the evaluation altogether. Otherwise, the agent continues and evaluates the operator mapping, where it calculates the input event rate in the target computational resource considering its upstream operators to verify if the resource can support the memory and CPU requirements, setting $c_{t+1}$ to 1 if any constraint is violated. The simulation then returns either $agg_cost = -1$ indicating constraint violations, or the *agg_cost* obtained in the simulation. At last, the agent updates the residual capacities of compute resources $\mathcal{R}_{t+1}$ and links $\mathcal{L}_{t+1}$.

The reward $R(s_{t+1})$ of a state $s_{t+1}$ is given by the difference between the $agg_cost_{s_0}$ of the original mapping and the $agg_cost_{s_{t+1}}$ of the state $s_{t+1}$. In other words:

$$R(s) = agg_cost_{s_0} - agg_cost_{s_{t+1}} \tag{19}$$

By solving the MDP one obtains a policy $\pi(s) : s \in \mathcal{S} \mapsto a \in \mathcal{A}(s)$ with the migrations needed to reconfigure the operator deployment. An *optimal policy* is a solution that maximises the expected reward.

## 3.2 Reinforcement Learning Algorithms

Known RL approaches to approximate the optimal state transition rewards in MDP problems rely on the Monte-Carlo Tree Search (MCTS) algorithm. MCTS, depicted in Algorithm 1, is a search mechanism consisting of running a number of simulations to build a search tree with the results [6]. Each node $n(s)$ of the search tree $\mathcal{T}$ represents a state $s$ that has been explored during simulation. A node maintains a count $N(s)$ of the number of visits for the state, an action value $Q(s, a)$ for each action $a \in \mathcal{A}(s)$ and a count $N(s, a)$ with the number of times the action was chosen.

Within a given number of iterations, MCTS builds the decision tree one node at a time, starting with the *root node* with the state given by the current deployment. At each iteration, the algorithm takes a set of actions resulting in an episode whose evaluation will force the creation of a new node. The episode comprises a set of tuples $\langle s, a, r, s' \rangle$ where $s$ is the evaluated state, $a$ is the action to be taken, $r$ is the reward given by $R(s)$, and $s'$ is the state resulting from taking action $a$ in state $s$.

In the context of operator reconfiguration, the episode creation (lines 8–19) begins at the root node and iterates over the deployment sequence $\mathcal{D}$. At each iteration, the algorithm picks a possible action from $\mathcal{A}(s)$ that either maintains the placement or migrates the operator. Herein a possible action evaluates existing nodes and ignores actions that would result in assigning operators to constrained nodes. When a state node exists in the search tree, a *tree policy* is employed to evaluate actions; otherwise a *default policy* is used. The simplest combination of policies comprises *greedy* selection $max_a Q(s, a)$ as tree policy and random choice of actions as default policy. After selecting an action, the algorithm evaluates it using the simulated environment and appends the resulting state to the episode, along with the reward and the action itself (lines 16–17). If the new state is invalid, the algorithm considers it as terminal. Otherwise, the new state is evaluated in the next iteration (line 18) and this process is repeated until the end of the deployment sequence.

The first state observed in the tuples of the episode that does not contain a node in the decision tree (line 21), will result in the creation of a new node in the tree (line 22). The action value $Q(s, a)$ and the counter with the number of times the action was chosen $N(s, a)$ are initialised with 0 (lines 23–24). After expanding the decision tree, the value obtained in the terminal state of the episode (line 26) will be used to update the $Q(s, a)$ and $N(s, a)$ in the traversal nodes towards the root node (lines 27–29). The update method varies depending on the variant of the MCTS algorithm – *e.g.*, MCTS-UCT and TDTS-Sarsa($\lambda$).

**Algorithm 1:** The MCTS algorithm.

```
Function MCTS(s_0)
    T ← n(s_0)
    while within computational budget do
        episode ← GenerateEpisode (T, s_0)
        ExpandTree (T, episode)
        Backup (T, episode)
    return n(s) ∈ T with the best reward
Function GenerateEpisode(T, s_0)
    episode ← {}
    s ← s_0
    while s is not terminal do
        if n(s) ∈ T then
            a ← TreePolicy (s)
        else
            a ← DefaultPolicy (s)
        (s, a, r, s') ← SimulateTransition (s, a)
        append (s, a, r, s') to episode
        s ← s'
    return episode
Procedure ExpandTree(T, episode)
    (s, a, r, s') ← first tuple where n(s') ∉ T ∧ s' ∈ episode
    T ← T ∪ n(s')
    Q(s', a) ← 0
    N(s', a) ← 0
Procedure Backup(T, episode)
    R ← r from episode(|episode| − 1)
    for i = Length(episode) down to 1 do
        (s, a, r, s') ← episode(i)
        UpdateTreeValues (T, s, R)
```

The MCTS algorithm, in its basic form, can take many steps to converge to a good solution, thus making it costly. This is mostly due to the effect that the large search space has over the action-values. Another issue is the inaccurate estimation of the action-values; MCTS accounts only for the final outcome to update the $Q(s, a)$ while some methods sample predicted future states and bootstrap to adjust the estimations. To solve the basic MCTS drawbacks, variants of the algorithm were proposed. For instance, the algorithms MCTS-UCT and TDTS-Sarsa($\lambda$) change the MCTS behaviour to obtain the action-value $Q(s, a)$. Each algorithm mainly varies the $TreePolicy$ and *Backup* – different approaches to $UpdateTreeValues$ – methods by employing Upper Confidence Bound (UCB) and/or Temporal-Difference Tree Search (TDTS).

*3.2.1 MCTS-UCT.* MCTS with Upper Confidence Bounds for Trees (UCT) is an algorithm that applies UCB in the *tree policy* to avoid the inefficiencies of the greedy approach that might stick to a limited number of actions after a few poor choices. The UCT algorithm uses an *optimistic approach in the face of uncertainty* by giving a bonus that represents the uncertainty in the $Q(s, a)$ value, hence aiming to explore actions less frequently visited which can favour potential action-values.

The UCT algorithm handles each state of the decision tree as a multi-armed bandit, in which each action available to the operator corresponds to an arm of the bandit. The tree policy chooses $a^*$ action using UCB1 algorithm [24] to maximise a UCT on the value of actions $Q(s, a)$ to balance the exploitation of known good reconfiguration mappings evaluating $Q(s, a)$ and the exploration of untried reconfigurations. The constant $C$ is the factor used to control the impact of the exploration on node selection:

$$UCT(s, a) = Q(s, a) + C\sqrt{\frac{2lnN(s)}{N(s, a)}} \tag{20}$$

$$a^* = max_a Q(s, a) \tag{21}$$

In our version of MCTS-UCT, $DefaultPolicy$ uses a random walk approach to choose actions. In the $UCT(s, a)$ function of the $TreePolicy$ the exploitation is replaced by $\frac{Q(s,a)}{N(s,a)}$ as it is the most common approach when using MCTS-UCT. The *Backup* the algorithm increments the number of node visits $N(s, a)$ and adds the reward from the last tuple to $Q(s, a)$.

*3.2.2 TDTS-Sarsa($\lambda$).* A combination of Sarsa($\lambda$) and UCT [29], TDTS-Sarsa($\lambda$) uses the same $DefaultPolicy$ and $TreePolicy$ of MCTS-UCT, but with a different *Backup* method. MCTS-UCT back-propagates the reward of the episode's last tuple and updates the action-values $Q(s, a)$. TDTS-Sarsa($\lambda$)'s *Backup* backtracks the reward of the last tuple of the episode and iterates the episode applying rates of discount reward and eligibility trace decay to after states $Q(s\prime, a\prime)$ and updates the $Q(s, a)$.

## 4 PERFORMANCE EVALUATION

This section describes the experimental setup, the performance metrics, and the obtained results.

### 4.1 Experimental Setup

A built-in-house framework atop OMNET++ [27] is used to model and simulate DSP applications. We resort to simulation as it provides a controllable and repeatable environment.

The edge devices are modelled as Raspberry PI's 2 (RPi) (*i.e.*, 4,74 MIPS at 1 GHz and 1 GB of RAM), and the cloud as AMD RYZEN 7 1800x (*i.e.*, 304,51 MIPS[4] at 3.6 GHz and 1 TB of memory). The infrastructure comprises two edge sites (*i.e., cloudlets*) with 20 RPi's each and a *Cloud* with 2 servers. A gateway interfaces each edge site's LAN and the external WAN [7] (the Internet). The LAN latency is uniformly distributed between 0.015 and 0.8 ms with a bandwidth of 100 Mbps. The WAN latency is drawn uniformly between 65 and 85 ms, and bandwidth of 1 Gbps [11].

Eleven application graphs with single and multiple data paths are considered. The graphs were crafted using a Python library[5] and their orders are based on the size of Realtime IoT Benchmark (RIoTBench) topologies [23], which are representative of today's data stream processing applications. The number of stateful operators corresponds to 20% of the whole application, also based on

[4]https://reddit.com/r/BOINC/comments/5xog5v/boinc_performance_on_amd_ryzen
[5]https://gist.github.com/bwbaugh/4602818

RIoTBench topologies. The operator behaviours vary with their parameters uniformly drawn from the values shown in Table 2. The sources and sinks are placed on edge sites except for the sink on the critical path, which is hosted on the cloud. This is the typical behaviour of IoT applications that collect data from sensors located on the edge of the Internet and have to provide response to nearby actuators, whereas part of the processing is performed on the cloud.

**Table 2: Operator parameters in the application graphs.**

| Parameter | Value | Unit |
|---|---|---|
| *cpu* | 1000-10000 | Instructions per second |
| Data compression rate | 10-100 | % |
| *mem* | 100-7500 | bytes |
| Input event size | 100-2500 | bits/second |
| Selectivity | 10-100 | % |
| Input event rate | 1000-10000 | Number of messages |
| *ws* (window size) | 1-100 | Number of messages |

**Performance Metrics:** As explained earlier, the following QoS requirements are considered as performance metrics:

- **Aggregate end-to-end latency**, which is the end-to-end latency in seconds from the time events are generated to the time they are processed by the sinks;
- **Monetary cost** that represents the cost in dollars by using the Microsoft Azure IoT Hub Pricing policy;
- **WAN traffic** which corresponds the amount of bytes transferred inter cloudlets, or/and among cloudlets and cloud communication; and
- **Reconfiguration overhead** which is the maximum time (seconds) to reassign operators and states across the infrastructure.

One iteration of a Monitoring, Analysis, Planning and Execution (MAPE) loop is considered for operator reconfiguration. The DSP scheduler *monitors* the application performance metrics while running the application under the placement provided by a deployment algorithm (described next). The QoS requirements are *analysed* when the execution reaches 300 seconds or all application paths have 500 messages; whichever comes last. Based on the analysis, the scheduler *plans* the operator reconfiguration using the RL algorithms with a computational budget of 10,000 iterations. At last, the scheduler *executes* the reconfiguration plan.

The RL algorithms are compared against a traditional deployment approach (cloud-only) and Taneja's Cloud-Edge Placement (TCEP) [25] from the state-of-the-art that performs cloud-edge placement. *Cloud-only* deploys all operators on the cloud, apart from data source and data sink locations. TCEP iterates a vector containing the application operators, and for each operator the algorithm ranks the computational resources by CPU, gets the host of the middle of the rank, and evaluates CPU, memory, and bandwidth constraints to obtain the operator placement. If there exists any constraint the operator is assigned to the cloud.

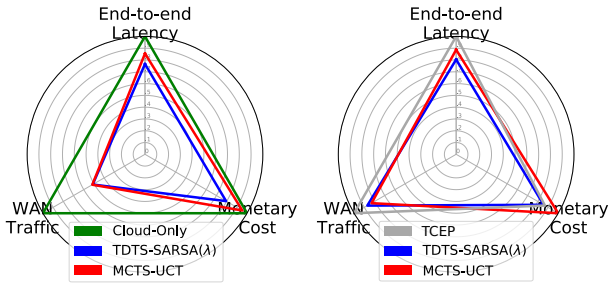



**Figure 3: Cloud-only and TCEP, with end-to-end latency weight = 1.**

## 4.2 Performance Evaluation

The performance evaluation was conducted by analysing the RL algorithms under three scenarios. First, the RL algorithms are compared against traditional and state-of-the-art solutions that are oblivious to multi-objective optimisation. Second, the impact of the reconfiguration overhead is measured. Finally, we examine the behaviour of RL algorithms when applying multiple weights to the QoS metrics.

*4.2.1 RL algorithms versus traditional and state-of-the-art deployment:* Figure 3 summarises the results for application reconfiguration while optimising the end-to-end latency only (*i.e.*, latency weight equal to 1). The presented values consist of weighted means where weights are the number of produced messages in each of the eleven evaluated applications normalised by the maximum observed value for all solutions (cloud-only, TCEP and RL algorithms). The left-hand graph in Figure 3 demonstrates that RL algorithms can achieve ≈ 20% better end-to-end latency, and reduce the WAN traffic by over 50% and the monetary cost by 15%. The downside of employing end-to-end latency as single-criterion optimisation is the lack of monetary cost and WAN traffic guarantees. The right-hand graph in Figure 3 shows the aforementioned drawback where the RL algorithms increase the monetary cost by over 15%.

The multi-objective approach provides a holistic view of the environment and allows for optimising multiple metrics simultaneously while avoiding unwanted spikes of monetary cost and WAN traffic. Figure 4 summarises the values when applying equal weights to end-to-end latency, WAN traffic and monetary cost. The results show that the RL algorithms outperform the state-of-the-art and traditional approach in terms of end-to-end latency and bring guarantees to WAN traffic and monetary cost when addressing the problem as a multi-objective optimisation. For instance, the RL algorithms reduce the end-to-end latency by 15% on average while bringing the monetary cost down by 70% and reducing the communication by 65%.

*4.2.2 Overhead of the reconfiguration decisions:* Although the WAN traffic, end-to-end latency, and monetary cost have a non-negligible impact on the quality of operator placement, it is important to consider the overhead that a reconfiguration decision might incur. The reconfiguration overhead comprises the time required

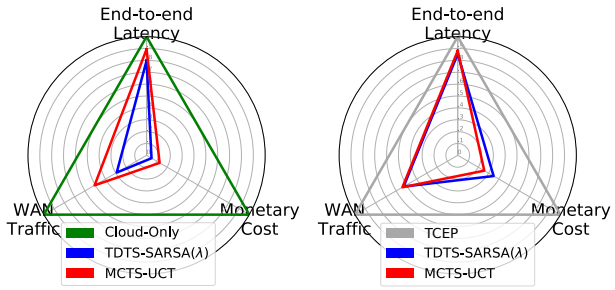



**Figure 4: Cloud-only and TCEP with weights for end-to-end latency, monetary cost and WAN traffic equal to 0.33.**

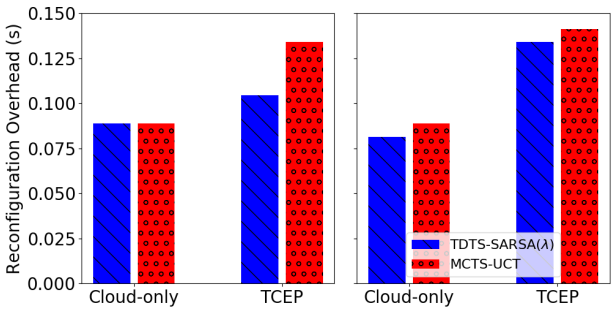



**Figure 5: Reconfiguration overhead of Cloud-only and TCEP. On the left-hand graph the end-to-end latency weight is 1; the right-hand graph shows results of equal weights to end-to-end latency, monetary cost and WAN traffic.**

to move operator states and code from one computational resource to another and restart the operator at the destination resource. Using a pause-and-resume approach, the system will be paused until the reconfiguration finishes while operators located upstream to those being migrated will store the events being ingested by the application data sources. For instance, if the system takes one second for reconfiguring an application, and a data source generates 10,000 events per second, then these events will be held by upstream operators resulting in long synchronisation overhead that can be unacceptable for time-sensitive applications.

Figure 5 presents the reconfiguration overhead for the weighted scenarios of Section 4.2.1. The RL algorithms achieved lower reconfiguration overhead in over 40% when starting from the cloud-only approach. As TCEP handles the infrastructure as a single set of computational resources (*i.e.*, the computational resources are considered to be in the same local area) achieving a disperse deployment as presented by Veith *et al.* [27]. On the one hand, TCEP operator deployment needs to migrate operators from multiple areas (between edge sites and among edge sites and the cloud), and on the other hand, starting with cloud-only, the RL algorithms have to move operators just from the cloud to the edge.

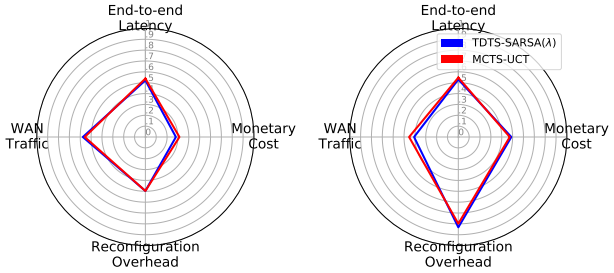



**Figure 6: Cloud-only and TCEP with 0.25 weight for end-to-end latency, monetary cost, WAN traffic and reconfiguration overhead.**

*4.2.3 RL algorithm behaviours when applying multiple weights to QoS metrics:* A set of experiments is conducted covering various combinations of metric weights to investigate how the RL algorithms react under multiple optimisation criteria. The metric values were obtained from the same weighted average of Section 4.2.1, but normalised by the maximum observed values from all experiments (Section 4.2.1 and 4.2.3). Hereafter, the term *base values* is used as a reference to the end-to-end latency, monetary cost and WAN traffic obtained from TCEP and Cloud-only operator deployments. For the reconfiguration overhead we consider the maximum value from RL algorithms when oblivious to this metric. Figure 6 summarises the results of equal weight of 0.25 to all metrics when starting with a deployment using either cloud-only or TCEP. The RL algorithms reduce in over 45% the end-to-end latency, monetary cost and WAN traffic while achieving ≈ 30% less reconfiguration overhead against the base values.

As DSP applications are time-sensitive, we evaluate a scenario focusing on improving the end-to-end latency along with the other QoS metrics. Figure 7 introduces the results with a weight of 0.7 to end-to-end latency and 0.1 to the other metrics. The set of weights allows for reducing the reconfiguration overhead by over 30%, the end-to-end latency by over 50%, the WAN traffic by more than 50% and the monetary cost in ≈ 45% compared to the base values. The current set of weights significantly reduces the reconfiguration overhead and slightly improves the end-to-end latency at the cost of raising the WAN traffic and the monetary cost when compared to the scenario that assigns equal weights to all QoS metrics. The high priority given to the end-to-end latency, the Equation 19, and the optimistic approach in the face of uncertainty of UCT-based algorithms – TDTS-Sarsa($\lambda$) and MCTS-UCT – led to explore actions that improve the reconfiguration overhead.

The previous scenarios provided some insights regarding the trade-off between end-to-end latency and the other metrics. The priority assigned to the end-to-end latency did not converge to a significant improvement whereas assigning equal weights of 0.25 achieves certain stability that does not vary when changing end-to-end latency to 0.7. However, focusing only on the end-to-end latency introduces a degradation of WAN traffic and monetary cost metric. Hence, we merged the two previous scenarios by giving 0.4 importance to the end-to-end end-to-end latency and 0.2 to WAN traffic, monetary cost, and reconfiguration overhead. Figure 8

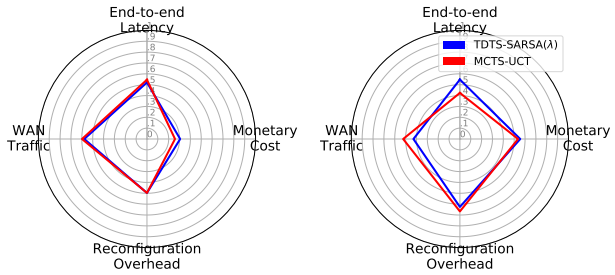



**Figure 7: Cloud-only and TCEP with weights 0.7, 0.1, 0.1, and 0.1 for end-to-end latency, monetary cost, WAN traffic and reconfiguration overhead, respectively.**

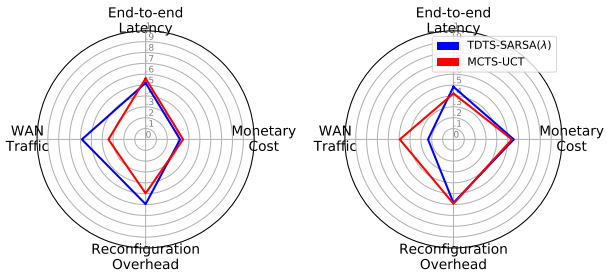



**Figure 8: Cloud-only and TCEP with weights 0.4, 0.2, 0.2, and 0.2 for end-to-end latency, monetary cost, WAN traffic and reconfiguration overhead, respectively.**

shows that as expected the end-to-end latency remains stable and the weights assigned to the QoS metrics allow for keeping the monetary cost down while reducing the reconfiguration overhead and the WAN traffic.

One can observe that the set of metric weights as well as the initial placement (cloud-only or TCEP) dictate the performance of the RL algorithms. In this sense, the best configuration was achieved by employing 0.4 to end-to-end latency and splitting 0.6 equally among the other QoS metrics. Also, both MCTS approaches had a similar behaviour as they are UCT-based and hence avoid the inefficiency of the greedy approach that may stick to a limited number of actions. In addition, TDTS had a slight degradation of performance when compared to the basic MCTS method. This happens because the algorithm SARSA($\lambda$) requires a fine tune of its parameters (reward discount and eligibility trace decay rates) as presented in Section 3.2.2. Overall, the proposed MDP model and the multi-objective reward brought a holistic view to the RL algorithms allowing for outperforming the state-of-the-art and the traditional approach.

## 5 RELATED WORK

Emerging IoT and monitoring services require the use of DSP applications for handling events under short delays and to enable quick decision making. An example consists of identifying the periodicity in traffic patterns using taxi GPS traces and leverage this for managing traffic lights efficiently and prevent traffic jams [8].

DSP applications are often fully deployed on the Cloud to take advantage of the virtually unlimited number of resources that it provides [20, 26]. However, data sources of emerging services such as in IoT are commonly geographically distant from the cloud, located at the edges of the Internet. The time to transfer events from data sources to the Cloud can become an issue to time-sensitive applications [10]. Recent work leverages the edges of the Internet or mobile devices for hosting parts of an application to minimise the end-to-end application latency or meet other QoS requirements [2, 4, 15, 22].

The process of assigning DSP operators to resources of a heterogeneous infrastructure (*i.e.*, operator placement) is challenging and has proven to be NP-Hard [1]. Some approaches simplify the problem whilst neglecting requirements such as communication [4], or ignoring certain operator behaviours and needs [22]. Usually, solutions address the placement as a single-objective optimisation problem while focusing mostly on end-to-end latency and therefore avoiding other important aspects such as WAN traffic and monetary cost [25].

The literature proposes solutions that address the problem as a multi-objective optimisation. For instance, Cardellini *et al.* [3] handle the DSP application deployment and replication by proposing a deterministic approach with a linear programming model that addresses metrics such as end-to-end latency, monetary cost and the reconfiguration overhead. Our work, on the other hand, considers a stochastic scenario with additional optimisation criteria and realistic pricing schemes. Moreover, given the benefits of applying RL to problems with large search spaces, such as the game of Go [6], previous work also applied RL to scheduling tasks to Edge computing [14]. RL has been used for solving deterministic and stochastic scheduling problems [14, 16] and elasticity of DSP applications. Vengerov *et al.* [28] show an RL framework for performing adaptive reconfiguration of dynamic resource allocations with Fuzzy. The decisions are taken in unknown stochastic dynamic environments assuming that the state space can be mitigated with time intervals much larger than the resource transfer time.

During the life-cycle of a DSP application, variations on the workload or infrastructure failures raise the need for reconfiguration. Liu *et al.* [13] propose a single-objective (end-to-end latency) model to reconfigure DSP applications based on collected statistics and Deep Neural Networks. Russo *et al.* [21] offer a multi-objective optimisation (monetary cost and QoS violations) Q-Learning model for reconfiguration. Panerati *et al.* [17] propose an autonomic manager using reinforcement learning algorithms and Q-learning applied to sensors and actuators to provide self-optimisation considering a multi-objective approach. Heinze *et al.* [9] proposed a multi-objective reconfiguration model based on the bin packing heuristic that explores the trade-off between monetary cost and end-to-end latency to predict and adapt the application to workload variations. The holistic view provided by a multi-objective approach for solving the operator reconfiguration problem fosters the use of RL [5, 19, 30].

Our solution embraces the multi-objective problem covering common metrics such as end-to-end latency and monetary cost,

and metrics neglected by other solutions, namely WAN traffic and reconfiguration overhead. Moreover, we use queueing theory and a realistic IoT pricing policy to model the DSP application behaviour and use it for simulation and learning using MCTS.

## 6 CONCLUSION & FUTURE WORK

In this paper, we modelled the problem of reconfiguring DSP applications onto heterogeneous computational and network resources by considering multiple QoS metrics. We introduced an MDP model and employed MCTS-UCT and TDTS-Sarsa($\lambda$), which are RL algorithms, to devise application reconfiguration plans. The RL algorithms tackled a multi-objective problem optimisation by considering end-to-end latency, WAN traffic, monetary cost, and reconfiguration overhead. The evaluated scenario covered multiple and complex applications varying the number of operators and their behaviours (selectivity, data compression and expansion factor, window size and processing requirements) employed to a cloud-edge infrastructure with heterogeneous computational resources (Raspberry Pi's and cloud servers).

The behaviour of RL algorithms was evaluated and compared against state-of-the-art solutions. The results show that our MDP model and multi-objective reward enable RL algorithms to have a holistic view of the operator reconfiguration and allow for reducing all target QoS metrics by over 50%.

As future work, we aim at building on the presented solutions and results and focus on minimising the search space by reducing the number of actions evaluated at each state. We aim at using heuristics that reduce the number of resources and operators used to build the set of possible actions.

## ACKNOWLEDGMENTS

This work was performed within the framework of the LABEX MILYON (ANR-10-LABX-0070) of Universite de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007).

## REFERENCES

[1] Anne Benoit, Alexandru Dobrila, Jean-Marc Nicod, and Laurent Philippe. 2013. Scheduling Linear Chain Streaming Applications on Heterogeneous Systems with Failures. *Future Gener. Comput. Syst.* 29, 5 (July 2013), 1140–1151.

[2] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2015. Distributed QoS-aware Scheduling in Storm. In *9th ACM Int. Conf. on Dstb Event-Based Systems (DEBS '15)*. ACM, New York, USA, 344–347.

[3] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurrency and Computation: Practice and Experience* 30, 9 (2018), e4334. https://doi.org/10.1002/cpe.4334 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4334 e4334 cpe.4334.

[4] Bin Cheng, Apostolos Papageorgiou, and Martin Bauer. 2016. Geelytics: Enabling On-Demand Edge Analytics over Scoped Data Sources. In *IEEE BigData*. 101–108.

[5] Daniele Foroni, Cristian Axenie, Stefano Bortoli, Mohamad Al Hajj Hassan, Ralph Acker, Radu Tudoran, Goetz Brasche, and Yannis Velegrakis. 2018. Moira: A goal-oriented incremental machine learning approach to dynamic resource cost estimation in distributed stream processing systems. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. ACM, 2.

[6] Sylvain Gelly and David Silver. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence* 175, 11 (2011), 1856–1875.

[7] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan. 2013. The Impact of Mobile Multimedia Applications on Data Center Consolidation. In *IEEE Int. Conf. on Cloud Engineering (IC2E)*. 166–176.

[8] Z. He, D. Zhang, J. Cao, X. Liu, X. Fan, and C. Xu. 2016. Exploiting Real-Time Traffic Light Scheduling with Taxi Traces. In *45th Int. Conf. on Parallel Processing*. 314–323. https://doi.org/10.1109/ICPP.2016.43

[9] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. 2015. Online Parameter Optimization for Elastic Data Stream Processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 276–287. https://doi.org/10.1145/2806777.2806847

[10] C. Hochreiner, M. Vogler, P. Waibel, and S. Dustdar. 2016. VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things. In *20th IEEE Int. Ent. Dstb Object Comp. Conf.* 1–11.

[11] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2016. Quantifying the Impact of Edge Computing on Mobile Applications. In *7th ACM SIGOPS Asia-Pacific Wksp on Systems*. ACM, New York, USA, Article 5, 8 pages.

[12] Geetika T. Lakshmanan, Ying Li, and Rob Strom. 2008. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing* 12, 6 (November 2008), 50–60.

[13] Z. Liu, H. Zhang, B. Rao, and L. Wang. 2018. A Reinforcement Learning Based Resource Management Approach for Time-critical Workloads in Distributed Computing Environment. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 252–261. https://doi.org/10.1109/BigData.2018.8622393

[14] Long Mai, Nhu-Ngoc Dao, and Minho Park. 2018. Real-Time Task Assignment Approach Leveraging Reinforcement Learning with Evolution Strategies for Long-Term Latency Minimization in Fog Computing. *Sensors* 18, 9 (2018), 2830.

[15] L. Ni, J. Zhang, C. Jiang, C. Yan, and K. Yu. 2017. Resource Allocation Strategy in Fog Computing Based on Priced Timed Petri Nets. *IEEE IoT Journal* PP (2017), 1–1.

[16] Alexandru Iulian Orhean, Florin Pop, and Ioan Raicu. 2018. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *J. Parallel and Distrib. Comput.* 117 (2018), 292–302.

[17] J. Panerati, F. Sironi, M. Carminati, M. Maggio, G. Beltrame, P. J. Gmytrasiewicz, D. Sciuto, and M. D. Santambrogio. 2013. On self-adaptive resource allocation through reinforcement learning. In *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)*. 23–30. https://doi.org/10.1109/AHS.2013.6604222

[18] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-Storm: Resource-Aware Scheduling in Storm. In *16th Annual Middleware Conf. (Middleware '15)*. ACM, New York, NY, USA, 149–161.

[19] D. Perez, S. Mostaghim, S. Samothrakis, and S. M. Lucas. 2015. Multiobjective Monte Carlo Tree Search for Real-Time Games. *IEEE Transactions on Computational Intelligence and AI in Games* 7, 4 (Dec 2015), 347–360. https://doi.org/10.1109/TCIAIG.2014.2345842

[20] Olubisi Runsewe and Nancy Samaan. 2017. 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)*. IEEE Press, Piscataway, NJ, USA, 848–857.

[21] Gabriele Russo Russo, Matteo Nardelli, Valeria Cardellini, and Francesco Lo Presti. 2018. Multi-Level Elasticity for Wide-Area Data Streaming Systems: A Reinforcement Learning Approach. *Algorithms* 11, 9 (2018), 134.

[22] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov. 2016. SpanEdge: Towards Unifying Stream Processing over Central and Near-the-Edge Data Centers. In *2016 IEEE/ACM Symp. on Edge Comp.* 168–178.

[23] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. 2017. RIoTBench: An IoT benchmark for distributed stream processing systems. *CCPE* 29, 21 (2017), e4257.

[24] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An introduction*. MIT press.

[25] M. Taneja and A. Davy. 2017. Resource aware placement of IoT application modules in Fog-Cloud Computing Paradigm. In *IFIP/IEEE Symp. on Integrated Net. and Service Mgmt (IM)*. 1222–1228.

[26] Rafael Tolosana-Calasanz, José Ángel Bañares, Congduc Pham, and Omer F. Rana. 2016. Resource management for bursty streams on multi-tenancy cloud environments. *Future Generation Computer Systems* 55 (2016), 444 – 459. https://doi.org/10.1016/j.future.2015.03.012

[27] Alexandre Veith, Marcos Dias de Assuncao, and Laurent Lefevre. 2018. Latency-Aware Placement of Data Stream Analytics on Edge Computing. In *16th International Conference on Service Oriented Computing (ICSOC 2018)*. Springer Int. Publishing, 215–229.

[28] David Vengerov. 2007. A reinforcement learning approach to dynamic resource allocation. *Engineering Applications of Artificial Intelligence* 20, 3 (2007), 383–390. https://doi.org/10.1016/j.engappai.2006.06.019

[29] Tom Vodopivec, Spyridon Samothrakis, and Branko Ster. 2017. On Monte Carlo Tree Search and Reinforcement Learning. *Journal of Artificial Intelligence Research* 60 (2017), 881–936.

[30] Weijia Wang and Michèle Sebag. 2012. Multi-objective Monte-Carlo Tree Search. In *Proceedings of the Asian Conference on Machine Learning (Proceedings of Machine Learning Research)*, Steven C. H. Hoi and Wray Buntine (Eds.), Vol. 25. PMLR, Singapore Management University, Singapore, 507–522. http://proceedings.mlr.press/v25/wang12b.html

[31] K.P. Yoon, P.K. Yoon, C.L. Hwang, SAGE., and inc Sage Publications. 1995. *Multiple Attribute Decision Making: An Introduction*. Number nos. 102-104 in Multiple Attribute Decision Making: An Introduction. SAGE Publications. https://books.google.fr/books?id=Fo47SWBuEyMC