


Numéro National de Thèse : 2019LYSEN050

THESE de DOCTORAT DE L'UNIVERSITE DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale *N°*512

Ecole Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 23/09/2019, par :

Alexandre DA SILVA VEITH

Quality of Service Aware Mechanisms for (Re)Configuring Data Stream Processing Applications on Highly Distributed Infrastructure

Mécanismes prenant en compte la qualité de service pour la (re)configuration d'applications de traitement de flux de données sur une infrastructure hautement distribuée

Devant le jury composé de :

Omer RANA	Professeur Université de Cardiff (UK)	*Rapporteur*
Patricia STOLF	Maître de Conférences Université Paul Sabatier	*Rapporteure*
Hélène COULLON	Maître de Conférences IMT Atlantique	*Examinatrice*
Frédéric DESPREZ	Directeur de Recherche Inria	*Examinateur*
Guillaume PIERRE	Professeur Université Rennes 1	*Examinateur*
Laurent LEFEVRE	Chargé de Recherche Inria	*Directeur*
Marcos DIAS DE ASSUNCAO	Chargé de Recherche Inria	*Co-encadrant*

Contents

Acknowledgments

Firstly, I would like to express my sincere gratitude to one of my advisors, Dr. Marcos Dias de Assunção, for guiding me endlessly and furnishing valuable research insights and ceaseless support throughout my Ph.D., without which I could never reach this milestone of writing acknowledgment. Also, I wish to thank my principal advisor, Dr. Laurent Lefevre, for enthusiastically parting with me his expertise and vision when I need a pair of professional eyes to analyse the real nature of the problem. My supervisors helped me to grow as a person and as a researcher, through many productive talks and vital encouragement they helped me to the achievement of this thesis and shed lights on future research.

I am also immensely grateful to my family. I cannot imagine a greater fortune other than having them in my life. My wife, Ediane Lodetti Veith, and my son, Pietro Lodetti Veith, for supporting me in my decisions, following me to this challenge in a new country where we all had to learn the language and traditions together. Thanks also my parents, Noeli da Silva Veith and Artur Veith, for teaching me with lessons on honesty and ethics that no university could ever rival. Their love and care are indispensable to any of my achievements.

I would like to thank my peer and office mate, Felipe Rodrigo de Souza, for long discussions and debates to improve my thesis. Also, to the AVALON team for having welcomed me and providing me with the knowledge to grow professionally. Special thanks go to the members of the teams which I collaborated: RDI2 of Rutgers University, ARCOS of University Carlos III of Madrid and GPPD of Federal University of Rio Grande do Sul.

I also express gratitude to INRIA and LABEX MILYON (ANR-10-LABX-0070) of the University of Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007), for providing me with financial support to pursue this doctoral degree. I am also grateful to ENS de Lyon for providing me with an office, and computer facilities during the course of my Ph.D.

French Abstract

La déploiement massif de capteurs, de téléphones mobiles et d'autres appareils a entraîné une explosion du volume, de la variété et de la vitesse des données générées (messages, événements, tuplets), et qui nécessitent d'être analysées.

Notre société devient de plus en plus interconnectée et produit de grandes quantités de données provenant des processus métier instrumentés, de la surveillance de l'activité des utilisateurs [43, 150], des objets connectés et assistants portables [69], des capteurs, des processus financiers, des systèmes à large échelle, d'expériences scientifiques, entre autres. Ce déluge de données est souvent qualifié de *big data* en raison des problèmes qu'il pose aux infrastructures existantes en matière de transfert, de stockage et de traitement de données [19].

Une grande partie de ces données volumineuses ont plus de valeur lorsqu'elles sont analysées rapidement, au fur et à mesure de leur génération. Dans plusieurs scénarios d'application émergents, tels que les villes intelligentes, la surveillance opérationnelle de grandes infrastructures et l'Internet des Objets (IoT, *Internet of Things*) [21], des flux continus de données doivent être traités dans des délais très brefs. Dans plusieurs domaines, ce traitement est nécessaire pour détecter des modèles, identifier des défaillances [115] et pour guider la prise de decision. Les données sont donc souvent rassemblées et analysées par des environnements logiciels conçus pour le traitement de flux continus de données.

Ces environnements logiciels pour le traitement de flux de données déploient les applications sous-la forme d'un graphe orienté ou de *dataflow*. Un *dataflow* contient une ou plusieurs sources (*i.e.* capteurs, passerelles ou actionneurs); opérateurs qui effectuent des transformations sur les données (*e.g.*, filtrage et agrégation); et des *sinks* (*i.e.*, éviers qui consomment les requêtes ou stockent les données).

La plupart des transformations effectuées par les opérateurs complexes – appelées aussi opérateurs avec état – stockent des informations en mémoire entre les executions. Un flux de données peut également avoir des opérateurs sans état qui prennent en compte uniquement les données requises par l'exécution actuelle. Traditionnellement, les applications de traitement de flux de données ont été conçues pour fonctionner sur des grappes de ressources homogènes (*i.e., cluster computing*) ou sur le *cloud* [18]. Dans un déploiement *cloud*, l'application entière est placée sur un seul fournisseur *cloud* pour que l'application puisse bénéficier d'un nombre virtuellement infini de ressources. Cette approche permet aux applications élastiques de traitement de flux de données d'allouer des ressources supplémentaires ou de libérer une capacité inactive à la demande pendant l'exécution d'une application, afin de répondre dynamiquement aux besoins.

Dans de nombreux scénarios, l'élasticité des nuages ne suffit pas à respecter les contraintes de temps du traitement de flux de données en raison de l'emplacement des sources de données et des changements lors du cycle de vie de l'application. Dans les scénarios IoT, les sources de données sont principalement situées aux extrémités de l'Internet et les données sont transférées vers le

cloud par des liens longue distance, ce qui augmente la *latence de bout en bout* des applications, aussi appelé *temps de réponse* ; c'est à dire, la différence entre le temps où les données sont générées jusqu'au moment où elles atteignent les *sinks*). La surcharge de communication liée au transfert de données par des liens Internet à haute latence rend impossible le traitement en temps quasi réel sur des architectures composées uniquement de *clouds*.

Une infrastructure *cloud* souvent utilisée pour les scénarios IoT – appelée ici infrastructure massivement distribuée – est celle où les données sont produites en permanence par plusieurs capteurs et contrôleurs, puis transmises à des passerelles, des commutateurs ou des concentrateurs situés à la périphérie du réseau et finalement traitées dans dans les noeuds de calcul des centres de calcul et données. Une infrastructure *edge* en périphérie comprend généralement des périphériques avec des capacités de mémoire et de processeur faibles, mais non négligeables, regroupés en fonction de leur emplacement ou de la latence du réseau. Un groupe de périphérie peut transférer des données vers un autre groupe ou vers le cloud, et le canal utilisé pour la communication est souvent l'Internet.

Plus récemment, des environnements logiciels [11, 109] et des architectures ont été proposés pour le traitement de flux de données sur des infrastructures hautement distribuées afin d'améliorer l'évolutivité et les latences de bout en bout des applications. Les ressources périphériques, souvent appelés *edge computing*, peuvent être exploités pour compléter les capacités informatiques du *cloud* et réduire la latence de bout en bout globale des applications, leurs besoins en bande passante et d'autres mesures de performances. L'exploration d'une telle infrastructure permet d'utiliser plusieurs modèles de performance et de minimiser les coûts liés à l'exécution de traitement de flux de données. L'utilisation d'une infrastructure *cloud* présente des défis supplémentaires en matière de planification des applications, d'élasticité des ressources et de modèles de programmation. La tâche de planification ou de configuration des opérateurs de traitement de flux de données sur une infrastructure hautement distribuée avec des ressources hétérogènes est généralement appelée *placement d'opérateur*, et s'est révélée être NP-difficile [27]. Déterminer *comment* déployer les opérateurs ou les migrer du *cloud* aux ressources périphériques du type *edge computing* est également un défi en raison des limitations des périphériques (en termes de mémoire, processeur, bande passante réseau) et du réseau (*i.e.*, Internet).

Les applications de traitement de flux de données ont une longue durée de fonctionnement pendant laquelle les conditions de charge et d'infrastructure peuvent changer. Après leur placement, il peut être nécessaire de réaffecter des opérateurs en raison de charges de travail variables ou de défaillances de périphériques. Le processus de réorganisation ou de migration des opérateurs d'une application de traitement de flux de données sur des ressources de calcul est appelé ici *reconfiguration*. Cette reconfiguration et le choix des opérateurs à réaffecter sont également NP-difficile.

L'espace de recherche de solution permettant de déterminer le placement des opérateurs de traitement de flux de données ou leur reconfiguration peut être énorme en fonction du nombre d'opérateurs, de flux, de ressources et de liens réseau. Le problème devient encore plus complexe lorsqu'on considère plusieurs métriques de qualité de service, par exemple, la latence de bout en bout, le volume de trafic utilisant des liens réseau *edge-cloud*, le coût monétaire lié au déploiement de l'application et la surcharge liée à la sauvegarde de l'application. À mesure que l'infrastructure et les applications *cloud* prennent de l'ampleur, essayer de concevoir un plan de (re)configuration tout en optimisant plusieurs objectifs peut entrainer un espace de recherche plus consequent.

Nous proposons dans cette thèse un ensemble de stratégies pour placer les opérateurs dans une infrastructure massivement distribuée *cloud-edge* en tenant compte des caractéristiques des

ressources et des exigences des applications. En particulier, nous décomposons tout d'abord le graphe d'application en identifiant quelques comportements tels que des *forks* et des *joints*, puis nous le plaçons dynamiquement sur l'infrastructure. Des simulations et un prototype prenant en compte plusieurs paramètres d'application démontrent que notre approche peut réduire la latence de bout en bout de plus de 50% et aussi améliorer d'autres métriques de qualité de service.

L'espace de recherche de solutions pour la reconfiguration des opérateurs peut être énorme en fonction du nombre d'opérateurs, de flux, de ressources et de liens réseau. De plus, il est important de minimiser le coût de la migration tout en améliorant la latence. Des travaux antérieurs, Reinforcement Learning (RL) et Monte-Carlo Tree Search (MCTS) ont été utilisés pour résoudre les problèmes liés aux grands nombres d'actions et d'états de recherche. Nous modélisons le problème de reconfiguration d'applications sous la forme d'un processus de décision de Markov (MDP) et étudions l'utilisation des algorithmes RL et MCTS pour concevoir des plans de reconfiguration améliorant plusieurs métriques de qualité de service.

Les principales contributions de cette thèse sont énumérées ci-dessous :

- Une exploration de l'élasticité des applications de traitement de flux de données et le placement d'opérateurs d'application de traitement de flux de données dans des infrastructures hétérogènes;
- Un modèle décrivant le calcul et les services de communication en se concentrant sur le temps de latence de bout en bout, ainsi que sur les contraintes de ressources en matière d'application et de calcul;
- Des stratégies de configuration des applications de traitement de flux de données prenant en compte l'optimisation mono et multi-objectifs;
- Une modélisation MDP utilisée par les algorithmes RL en considérant l'optimisation d'un ou de plusieurs objectifs pour reconfigurer les applications de traitement de flux de données.

Chapter 1

Introduction

Contents

Society is becoming more interconnected producing vast amounts of data as result of instrumented business processes, monitoring of user activity [43, 150], wearable assistance [69], sensors, finance, large-scale scientific experiments, among other reasons. This has led to an explosion in the volume, variety and velocity of data generated (*i.e.* messages, events, tuples) and that requires analysis of some type. This data deluge is often termed as *big data* due to the challenges it poses to existing infrastructure regarding data transfer, storage, and processing [19].

A large part of this big data is most valuable when it is analysed quickly, as it is generated. Under several emerging application scenarios, such as in smart cities, operational monitoring of large infrastructure, and Internet of Things (IoT) [21], continuous data streams must be processed under very short delays. In multiple domains, there is a need for processing data streams to detect patterns, identify failures [115], and gain insights. Streaming data is often gathered and analysed by Data Stream Processing Engines (DSPEs).

A DSPE commonly structures an application as a directed graph or dataflow, as depicted in Figure 1.1. A dataflow has: one or multiple sources (*i.e.*, sensors, gateways 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). Most complex operator transformations – hereafter called stateful operators – store information about previous executions as new data is streamed in. A dataflow can also have stateless operators that consider only the current execution and its input data. Traditionally, Data Stream Processing (DSP) applications were conceived to run on clusters of homogeneous resources or on the *cloud* [18]. In a cloud deployment, the whole application is placed on a single cloud provider to benefit from virtually unlimited resources.

This approach allows for elastic DSP applications with the ability to allocate additional resources or release idle capacity on demand during runtime to match the application requirements.

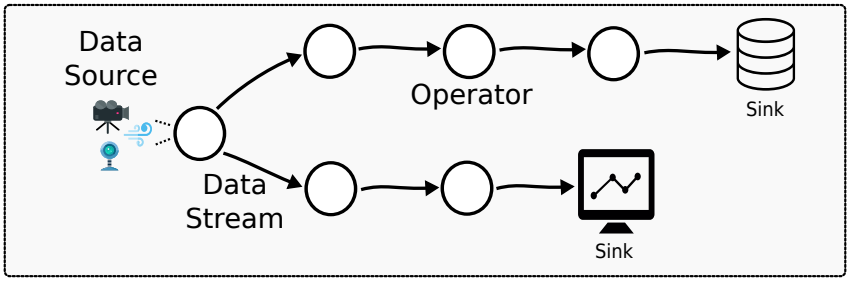

Figure 1.1: Abstract view of a DSP application with one data source, multiple operators and two sinks.

In many scenarios, cloud elasticity is not enough to meet DSP time requirements due to the location of the data sources and changes during the application life-cycle. In IoT scenarios data sources are mainly located at the edges of the Internet, and data is transferred to the cloud via long-distance links that increase the *end-to-end application latency*[1] (*i.e.*, the time data is generated to the time it reaches the sinks). The communication overhead incurred by transferring data through high-latency Internet links makes it impossible to achieve near real-time processing on clouds alone. The joint exploration of the cloud and devices at the edges of the Internet, called cloud-edge infrastructure, has led to new concepts in DSP. Figure 1.2 presents a common cloud-edge infrastructure for IoT scenarios [28] – termed here as highly distributed infrastructure – where data is continuously produced by multiple sensors and controllers, forwarded to gateways, switches, or hubs on the network edge using low latency networks and eventually to the cloud for processing. The edge commonly comprises devices with low, but non-negligible, memory and CPU capabilities grouped according to their location or network latency. One edge group can transfer data to another group or to the cloud, and the communication channel is often the Internet.

More recently, software frameworks [11, 109] and architecture have been proposed for carrying out DSP using hybrid cloud-edge infrastructure for improving the scalability and aiming to achieve short end-to-end latencies. The edge devices can be leveraged to complement the computing capabilities of the cloud and reduce the overall end-to-end application latency, bandwidth requirements and other performance metrics. Exploring such infrastructure allows for employing multiple performance models and minimise the overhead and costs of performing DSP. Using cloud-edge infrastructure, however, introduces additional challenges regarding application scheduling, resource elasticity, and programming models. The task of scheduling DSP operators on highly distributed infrastructure with heterogeneous resources is generally referred to as *operator placement* (*i.e.*, application configuration) and has proven to be NP-hard [27]. Determining *how* to deploy the operators or move them from the cloud to edge devices is also challenging because of the device limitations (*i.e.*, memory, CPU and network bandwidth) and the network (*i.e.*, Internet).

As DSP applications are long-running, the load and infrastructure conditions can change during their execution. After their initial placement, operators may need to be reassigned due to variable workload or device failures. The process of reorganising or migrating DSP application operators across compute resources is hereafter called *application reconfiguration*. Reconfiguring

[1]Response time and end-to-end latency are used interchangeably throughout this thesis for referring to end-to-end application latency.

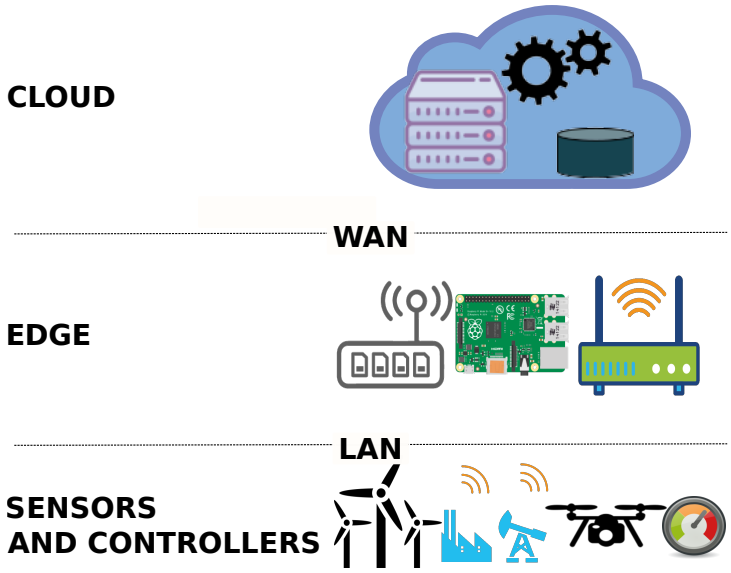

Figure 1.2: Overview of cloud-edge infrastructure which consists of cloud, edge, and sensors and controllers.

DSP applications and deciding what operators to be reassigned to which resources is also NP-Hard. The DSP application reconfiguration is guided by some strategies [128], for instance, *pause-and-resume* [36] or *parallel track* [73]. *Pause-and-resume* shutdowns or pauses the original operator before its state and code are migrated and execution efficiently restored on the new resource. A drawback of this approach is the increase in end-to-end latency, which is caused by need to pause upstream operators which will store the incoming data until the migration finishes, and then synchronise the data. *Parallel track* creates a new instance of the operator that is ran concurrently until the old and the new operator instances synchronise their states. Although it is faster than pause-and-resume, it requires enhanced mechanisms for state migration.

Moreover, the solution search space for determining a DSP application (re)configuration can be enormous depending on the number of operators, streams, resources and network links. The problem becomes even more complex when considering multiple Quality of Service (QoS) metrics such as end-to-end latency, volume of traffic using WAN links, monetary cost incurred by the application deployment, and the overhead posed by saving the application state and migrating operators to new resources. As the cloud-edge infrastructure and applications grow larger, trying to devise a (re)configuration plan while optimising multiple objectives can result in a larger search space.

1.1 Challenges in DSP Applications Deployment

A DSP application comprises operators that are heterogeneous regarding their behaviour and memory, CPU, and bandwidth requirements. An operator's behaviour means that it can discard or replicate data, maintain states, or change the size of the data it handles. Also, the CPU and memory capacity of a resource can impact the operator's behaviour. One challenge when deploying a DSP application is to model the involved communication and computation considering heterogeneous compute resources and DSP application operators.

To complicate matters further, DSP applications are latency-sensitive, meaning that data

must be handled in a timely manner. On one hand, cloud follows a *pay-as-you-go model* offering unlimited virtual resources for processing and storing events, hence supporting all kinds of operator requirements and behaviours. On the other hand, edge devices are often closer to where data is generated, but cannot support operators that have large compute and memory requirements due to their low processing and memory capacity. In addition, transferring data from cloud servers to edge devices, or vice-versa is done through WAN links, which incurs a significant increase in end-to-end latency.

The decisions on how to (re)configure DSP applications across cloud-edge infrastructure is challenging due to the heterogeneity of compute resources, QoS requirements, operators' behaviours, and limitations of computing resources. We discuss them as we explain each infrastructure and the steps for (re)configuring DSP operators.

1.1.1 Challenges in Edge Computing

Edge computing comprises devices with low capacity that are geographically distributed. A group of edge devices connected to the same LAN network is also called a *cloudlet* in this thesis. Edge computing comprises several cloudlets interconnected by WAN networks. When considering DSP applications, the data often is ingested by these devices and must flow to different cloudlets or to the cloud. Accounting for this infrastructure, there are multiple challenges to deploy the entire application on edge devices. One challenge derives from device limitations since DSP applications often have processing and/or memory-intensive operators, as well as operators with high volumes of data to store, which edge devices cannot fully support. Another challenge is on how to deploy the DSP application because operators have heterogeneous requirements and behaviours.

1.1.2 Challenges in Cloud Computing

Using a cloud to host an entire DSP application has limitations with respect to handling data in near real-time. Cloud services can provide unlimited virtual resources, but the communication overhead impacts the application performance. Since edge devices are responsible for data ingestion to the dataflow, the data might traverse WAN links to reach cloud servers, and hence create an issue in meeting execution requirements.

1.1.3 Challenges in DSP Operator Placement

The problem of initially placing the DSP on heterogeneous infrastructure has proven to be NP-Hard [27]. One issue is how to split the application across the available resources. Since the application and the infrastructure are heterogeneous, it is difficult to model the system, infrastructure, and QoS metrics. Another issue is how to define the application configuration scenario, which can be considered as either a single-objective or a multi-objective optimisation problem.

1.1.4 Challenges in DSP Application Reconfiguration

The DSP application reconfiguration consists of determining a new placement based on an initial configuration. To reconfigure an application, the system must stop or pause the flow between operators, and then reconfigure the operators following the new deployment plan. DSP applications have operators that keep states and generally process data following time semantics.

The process of stopping or pausing an application to move operators and states requires storing the data until the reconfiguration ends. After reconfiguring the application, operators must catch up and process all stored data without discarding them. The limited capacity of edge device becomes a challenge when storing large volumes of data during operator migration. Also, the reconfiguration overhead can be high, thus becoming a challenge to meet near real-time processing due to the time required to synchronise stored data.

1.2 Research Problem and Objectives

This thesis focuses on (re)configuration of DSP applications on cloud-edge infrastructure with the goal of improving single and multi-QoS metrics. To tackle the above-mentioned challenges, we seek to meet the following objectives:

- **How to model a DSP application on cloud-edge infrastructure?** The most common approach used to design a model is to employ Queueing Theory [22, 59, 96, 100, 147, 149]. However many existing models ignore connections between operators (*i.e.* streams) and that these operator connections follow rules (*i.e.* round robin) to distribute data among the connected down streams; the communication is hence neglected or oversimplified. Moreover, models disregard the changes in size and number of data tuples caused by each operator. Infrastructure limitations are generally overlooked, such as ignoring the memory limitations of edge devices. A model is therefore needed to help decide how to match operators and compute resources.

- **How to configure or place DSP applications considering single or multi-objective optimisation?** The default scheduler in many DSPEs is agnostic of matching the resource requirements with availability. In addition, existing resource-aware schedulers are static and oblivious to considering multiple QoS metrics. Therefore, a resource-efficient scheduler is required to tackle user-defined QoS metrics and infrastructure's limitations.

- **How to reconfigure a DSP application on cloud-edge infrastructure?** There are several strategies to control the dataflow when reconfiguring the DSP application by employing *pause-and-resume* or *parallel track* [73]. However, the overhead to migrate operators and states is often neglected. Furthermore, schedulers oversimplify the problem of reassigning operators to compute resources in order to reduce the search space. An approach that considers a large search space is required.

1.3 Evaluation Methodology

This thesis describes mechanisms for (re)configuring DSP applications across cloud-edge infrastructure that enable single or multi-objective optimisation. To evaluate the mechanisms proposed in this thesis, we perform discrete-event simulations as well as empirical experiments on real-life testbeds. Simulations create controllable environments for repeatable experiments. Real-life testbeds provide the uncertainty on communication and computation when running DSP applications.

We ran application benchmarks with synthetic applications covering multiple communication patterns, resource consumption requirements, and time-space operator complexities. Also, we included real-world DSP applications from the Realtime IoT Benchmark (RIoTBench) [129].

The details on the methodology followed to investigate each proposed method is described in the respective chapter.

1.4 Thesis Contribution

The **key contributions** of this thesis are listed below:

1. A survey of DSP application elasticity and DSP application (re)configuration in heterogeneous infrastructure.
2. A mathematical model that describes the computation and the communication services focusing on single and multi-QoS metrics, and existing constraints of cloud-edge infrastructure.
3. Strategies for configuring DSP applications considering single and multi-objective optimisation.
4. A Markov Decision Process (MDP) framework employed in Reinforcement Learning (RL) algorithms covering single and multi-objective optimisation to reconfigure DSP applications.

1.5 Thesis Organisation

The organisation of the thesis chapters is shown in Figure 1.3. Chapter 2 provides a taxonomy and survey of the state-of-the-art on resource elasticity and (re)configuration of DSP applications. Chapter 3 introduces the system model and problem statement providing details on how each system component and QoS metrics are modelled. Chapter 4 focuses on operator placement of DSP applications on cloud-edge infrastructure, covering single and multi-QoS objective solutions. Chapter 5 introduces RL methods employing single and multi-QoS metrics solution to deal with DSP application reconfiguration on cloud-edge infrastructure. Chapter 6 presents the conclusions and future directions.

The core chapters of this thesis are mainly derived from conference and journal publications completed during my Ph.D., which are listed as follows:

- Chapter 2 presents a survey and taxonomy of resource elasticity, and (re)configuration of DSP applications on heterogeneous infrastructure. It defines the scope of this thesis and positions its contribution in the area. This chapter is partially derived from:
  - **Alexandre da Silva Veith**, Marcos Dias de Assunção, and Laurent Lefèvre, "Assessing the Impact of Network Bandwidth and Operator Placement on Data Stream Processing for Edge Computing Environments", *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*, 2017.
  - Marcos Dias de Assunção, **Alexandre da Silva Veith**, and Rajkumar Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions", *Journal of Network and Computer Applications*, Volume 103, Pages: 1–17, Elsevier, February, 2018.
  - **Alexandre da Silva Veith**, and Marcos Dias de Assunção, "Apache Spark", *Encyclopedia of Big Data Technologies*, Springer International Publishing, Pages: 77–81, 2019.

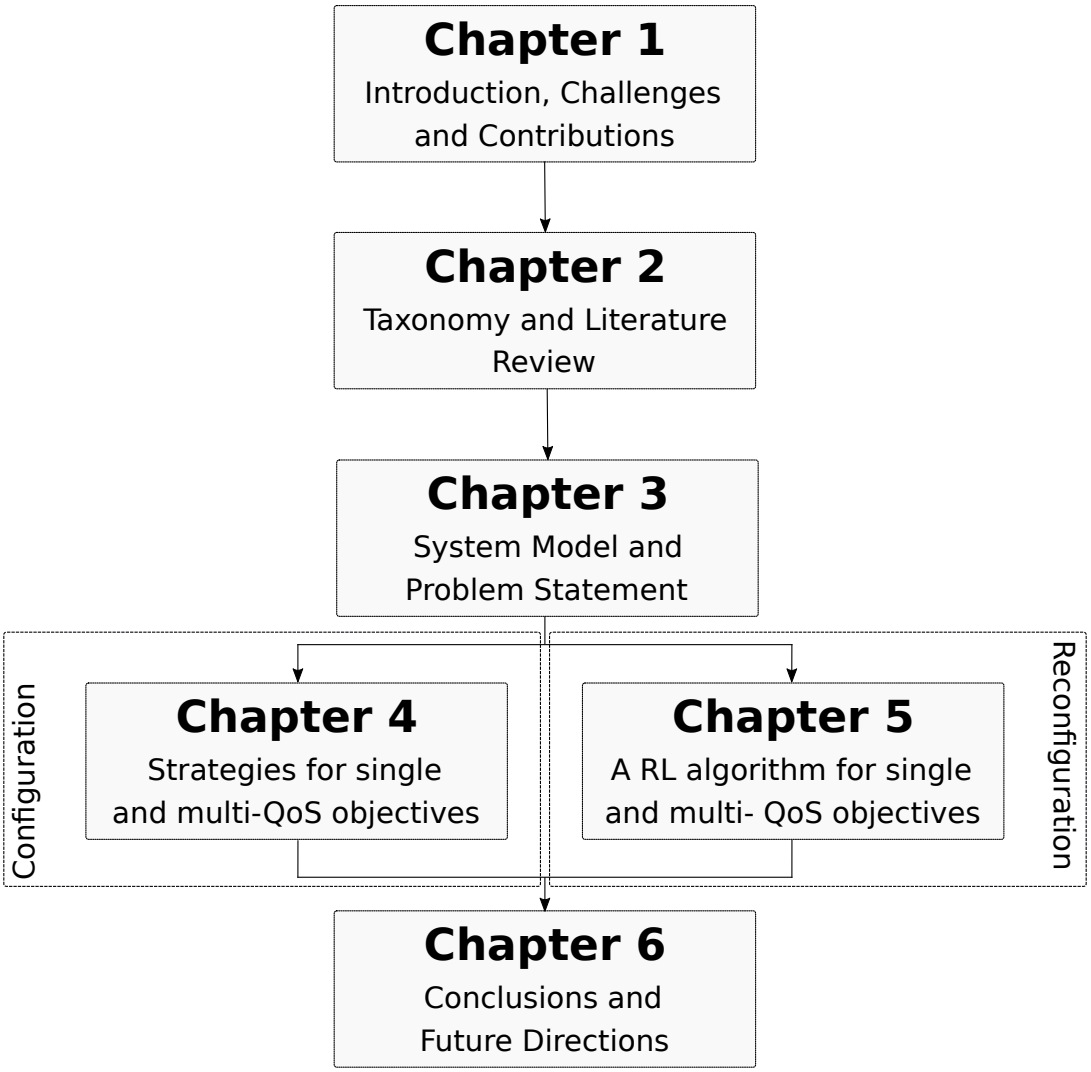

Figure 1.3: The thesis organisation.

- Chapter 3 introduces the system model for application (re)configuration on cloud-edge infrastructure considering both a single and a multi-objective approach. This chapter is derived from:

  – **Alexandre da Silva Veith**, Marcos Dias de Assunção, and Laurent Lefèvre, "Latency-Aware Strategies for Placing Data Stream Analytics onto Edge Computing", *USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, Boston, USA, 2018.

  – **Alexandre da Silva Veith**, Marcos Dias de Assunção, and Laurent Lefèvre, "Latency-Aware Placement of Data Stream Analytics on Edge Computing", in Proceedings of the 16th *International Conference on Service-Oriented Computing (ICSOC)*, Hangzhou, China, Pages: 215–229, Springer International Publishing, 2018.

  – Eduard Gibert Renart, **Alexandre da Silva Veith**, Daniel Balouek-Thomert, Marcos Dias de Assunção, Laurent Lefèvre, and Manish Parashar "Distributed Operator Placement for IoT Data Analytics Across Edge and Cloud Resources", in Proceedings

of the 19th Annual *IEEE/ACM International Symposium in Cluster, Cloud, and Grid (CCGrid)*, Pages: 459–468, Lanarca, Cyprus, May, 2019.

- Chapter 4 proposes strategies for DSP application configuration on cloud-edge infrastructure considering end-to-end application latency and multiple QoS metrics. This chapter is derived from:
  - **Alexandre da Silva Veith**, Marcos Dias de Assunção, and Laurent Lefèvre, "Latency-Aware Strategies for Placing Data Stream Analytics onto Edge Computing", *USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, Boston, USA, 2018.
  - **Alexandre da Silva Veith**, Marcos Dias de Assunção, and Laurent Lefèvre, "Latency-Aware Placement of Data Stream Analytics on Edge Computing", in Proceedings of the 16th *International Conference on Service-Oriented Computing (ICSOC)*, Hangzhou, China, Pages: 215–229, Springer International Publishing, 2018.
  - Eduard Gibert Renart, **Alexandre da Silva Veith**, Daniel Balouek-Thomert, Marcos Dias de Assunção, Laurent Lefèvre, and Manish Parashar "Distributed Operator Placement for IoT Data Analytics Across Edge and Cloud Resources", in Proceedings of the 19th Annual *IEEE/ACM International Symposium in Cluster, Cloud, and Grid (CCGrid)*, Pages: 459–468, Lanarca, Cyprus, May, 2019.
- Chapter 5 introduces a Reinforcement Learning (RL) algorithm to reconfigure DSP application considering end-to-end application latency and multiple QoS metrics on cloud-edge infrastructure. This chapter is partially derived from:
  - **Alexandre da Silva Veith**, Felipe Rodrigo de Souza, Marcos Dias de Assunção, Laurent Lefèvre, and Julio Cesar Santos dos Anjos, "Multi-Objective Reinforcement Learning for Reconfiguring Data Stream Analytics on Edge Computing", in Proceedings of 48th *International Conference on Parallel Processing (ICPP)*, Kyoto, Japan, 2019.
  - **Alexandre da Silva Veith**, Marcos Dias de Assunção, and Laurent Lefèvre, "Monte-Carlo Tree Search and Reinforcement Learning for Reconfiguring Data Stream Processing on Edge Computing", in Proceedings of *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Campo Grande, Brazil, 2019.

Chapter 2

State-of-the-art and Positioning

Contents

2.1 Introduction

Over the past, many Data Stream Processing Engines (DSPEs) have been deployed on clouds [18] aiming to benefit from characteristics such as resource elasticity. Elasticity, when properly exploited, refers to the ability of a cloud to allow a service to allocate additional resources or release idle capacity on demand to match the application workload. Although efforts have been

made towards making Data Stream Processing (DSP) more elastic, many issues remain unaddressed. There are challenges regarding the placement of DSP operators on available resources, identification of bottlenecks, and application reconfiguration. These challenges are exacerbated when services are part of a larger infrastructure that comprises multiple execution models (*e.g.* lambda architecture, workflows or resource-management bindings for high-level programming abstractions [30, 61]) or hybrid environments comprising both cloud and edge computing resources [81, 82].

More recently, software frameworks [11, 109] and architectures have been proposed for carrying out DSP using constrained resources located at the edge of the Internet. This scenario introduces additional challenges regarding application scheduling, resource elasticity, and programming models. This chapter surveys DSP solutions and approaches for deploying DSP on cloud computing and edge environments, discusses open issues, and positions this thesis in the subject. The chapter hence makes the following contributions:

- It reviews multiple generations of DSPEs, describing their architectural and execution models.
- It analyses and classifies existing work on exploiting elasticity to adapt resource allocation to match the demands of DSP services. Previous work has surveyed DSP solutions without discussing how resource elasticity is addressed [153]. This chapter provides a more in-depth analysis of existing solutions and discusses how they attempt to achieve resource elasticity.
- It describes ongoing efforts on resource elasticity for DSP and their deployment on edge computing environments.
- It highlights open issues and positions the thesis in the area of DSP application (re)configuration on cloud-edge infrastructure.

2.2 Data Stream Processing Architecture and Elasticity

This section describes background on DSP systems for big-data. It first discusses how layered real-time architecture is often organised and then presents a historical summary of how such systems have evolved.

2.2.1 Online Data Processing Architecture

Architecture for online[1] data analysis is generally composed of multi-tiered systems that comprise many loosely coupled components [3, 50, 90]. While the reasons for structuring architecture in this way may vary, the main goals include improving maintainability, scalability, and availability. Figure 2.1 provides an overview of components commonly found in a DSP architecture. Although an actual system might not have all these components, the goal here is to describe how a DSP architecture may look like and position the DSP solutions discussed later.

The *Data Sources* (Figure 2.1) that require timely processing and analysis include Web analytics, infrastructure operational monitoring, online advertising, social media, and Internet of Things (IoT). Most *Data Collection* is performed by tools that run close to where the data is and that communicate the data via TCP/IP connections, UDP, or long-range communication

[1]Similar to Boykin *et al.*, hereafter use the term *online* to mean that *"data are processed as they are being generated"*.

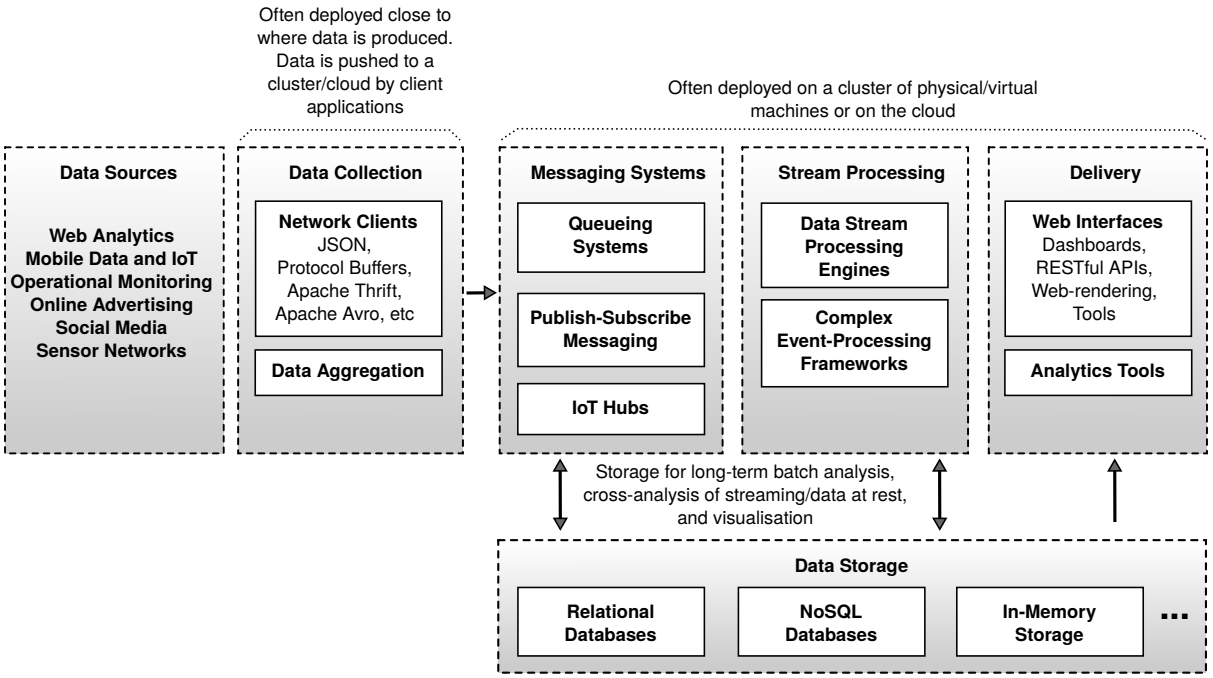

Figure 2.1: Overview of an online data-processing architecture.

[37]. Solutions such as JavaScript Object Notation (JSON) are used as a data-interchange format. For more structured data, wire protocols such as Apache Thrift [16] and Protocol Buffers [110], can be employed. Other messaging protocols have been proposed for IoT, some of which are based on HTTP [21]. Most data collection activities are executed at the edges of a network, and some level of data aggregation is often performed via, for instance Message Queue Telemetry Transport (MQTT), before data is passed through to be processed and analysed.

An online data-processing architecture can comprise multiple tiers of collection and processing, with the connection between these tiers made on an ad-hoc basis. To allow for more modular systems, and to enable each tier to grow at different paces and hence accommodate changes, the connection is at times made by message brokers and queuing systems such as Apache ActiveMQ [9], RabbitMQ [112] and Kestrel [84], publish-subscribe based solutions including Apache Kafka [14] and DistributedLog [47], or managed services such as Amazon Kinesis Firehose [6] and Azure IoT Hubs [24]. These systems, termed here as "Messaging Systems", enable for instance, the processing tier to expand to multiple data centres and collection to be changed without impacting processing.

Over the years several models and frameworks have been created for processing large volumes of data, among which MapReduce is one of the most popular [46]. Although most frameworks process data in a batch manner, numerous attempts have been made to adapt them to handle more interactive and dynamic workloads [29, 41]. Such solutions handle many of today's use cases, but there is an increasing need for processing collected data always at higher rates and providing services with short response time. DSP systems are commonly designed to handle and perform one-pass processing of unbounded streams of data. This tier, the main focus of this chapter, includes solutions that are commonly referred to as stream management systems and complex-event processing systems. The next sections review data streams and provide a historic overview of how this core component of the data processing pipeline has evolved over time.

Moreover, a data processing architecture often stores data for further processing, or as support to present results to analysts or deliver them to other analytics tools. The range of *Data Storage* solutions used to support a real-time architecture are numerous, ranging from relational databases, to key-value stores, in-memory databases, and NoSQL databases [70]. The results of data processing are delivered (*i.e.* Delivery tier) to be used by analysts or machine learning and data mining tools. Means to interface with such tools or to present results to be visualised by analysts include RESTful or other Web-based APIs, Web interfaces and other rendering solutions. There are also many data storage solutions provided by cloud providers such as Amazon, Azure, Google, and others.

2.2.2 Data Streams and Models

The definition of a data stream can vary across domains, but in general, it is commonly regarded as input data that arrives at a high rate, often being considered as big data, hence stressing communication and computing infrastructure. The type of data in a stream may vary according to the application scenario, including discrete signals, event logs, monitoring information, time series data, video, among others. Moreover, it is important to distinguish between streaming data when it arrives at the processing system via, for instance, a log or queueing system, and intermediate streams of *tuples* resulting from the processing by system elements. When discussing solutions, this work focuses on the resource management and elasticity aspects concerning the intermediate streams of tuples created or/and processed by elements of a DSP system.

An input data stream is an online and unbounded sequence of data elements [26, 60]. The elements can be homogeneous, hence structured, or heterogeneous, thus semi-structured or unstructured. More formally, an input stream is a sequence of data elements $e_1, e_2, \ldots$ that arrive one at a time, where each element e_i can be viewed as $e_i = (t_i, D_i)$ where t_i is the time stamp associated with the element, and $D_i = \langle d_1, d_2, \ldots \rangle$ is the element payload, here represented as a *tuple* of data items.

As mentioned earlier, many DSPEs use a dataflow abstraction by structuring an application as a graph, generally a Directed Acyclic Graph (DAG), of *operators*. These operators perform functions such as counting, filtering, projection, and aggregation, where the processing of an input data stream by an element can result in the creation of subsequent streams that may differ from the original stream in terms of data structure and rate.

Frameworks that structure DSP applications as dataflow graph generally employ a logical abstraction for specifying operators and how data flows between them; this abstraction is termed here as *logical plan* [87] (see Figure 2.2). As explained in detail later, a developer can provide parallelisation hints or specify how many instances of each operator should be created when building the *physical plan* that is used by a scheduler or another component responsible for placing the operator instances on available cluster resources. As depicted in the figure, physical instances of a same logical operator may be placed onto different physical or virtual resources.

With respect to the selectivity of an operator (*i.e.* the number of items it produces per number of items consumed) it is generally classified [56] (Figure 2.3) as *selective*, where it produces less than one; *one-to-one*, where the number of items is equal to one; or *prolific*, in which it produces more than one. Similarly, an operator can transform the data after its execution – Data Transformation Pattern – and this pattern result in *expansion*, when the output data size is greater than the input data size; *stable*, the operator transformation does not change the data size; and *compression*, where the output data size is smaller than the input data. Regarding state, an operator can be *stateless*, in which case it does not maintain any state

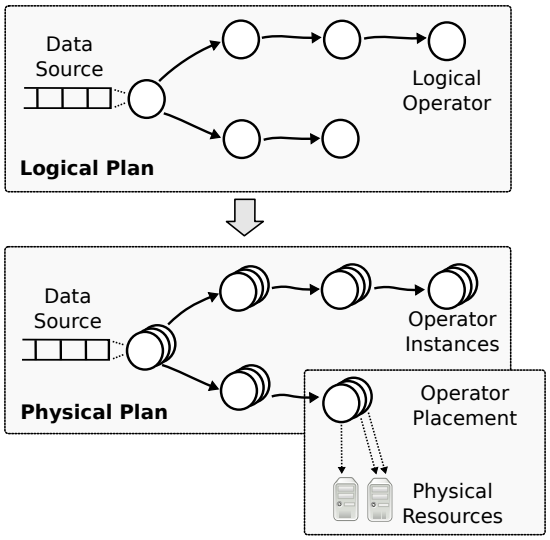

Figure 2.2: Logical and physical operator plans.

between executions; *partitioned stateful* where a given data structure maintains state for each downstream based on a partitioning key, and *stateful* where no particular structure is required.

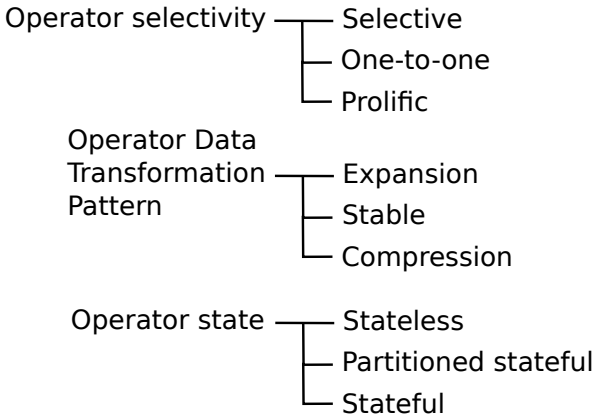

Figure 2.3: Types of operator selectivity and state.

Organising a DSP application as a graph of operators allows for exploring certain levels of parallelism (Figure 2.4) [135]. For example, *pipeline parallelism* enables an operator to process a tuple while an upstream operator can handle the next tuple concurrently. Graphs can contain segments that execute the same set of tuples in parallel, hence exploiting *task parallelism*. Several techniques also aim to use *data parallelism*, which often requires changes in the graph to replicate operators and adjust the data streams between them. For example, parallelising regions of a chain graph [56] may consist of creating multiple pipelines preceded by an operator that partitions the incoming tuples across the downstream pipelines – often called a *splitter* – and followed by an operator that merges the tuples processed along the pipelines – termed as *mergers*. Although parallelising regions can increase throughput, they may require mechanisms to guarantee time semantics, which can make splitters and mergers block for some time to guarantee, for instance, time order of events.

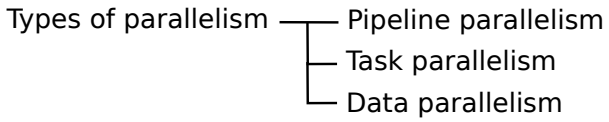

Figure 2.4: Some types of parallelism enabled by dataflow based DSP.

2.2.3 Distributed Data Stream Processing

Several systems have been developed to process dynamic or streaming data [90, 122] (*i.e.*, DSPEs). One of the categories under which such systems fall is often called Data Stream Management System (DSMS), analogous to DataBase Management Systems (DBMSs) which are responsible for managing disk-resident data usually providing users with means to perform relational operations among table elements. DSMSs include operators that perform standard functions, joins, aggregations, filtering, and advanced analyses. Early DSMSs provided SQL-like declarative languages for specifying long-running queries over unbounded streams of data. Complex Event Processing (CEP) systems [141], a second category, support the detection of relationships among events, for example, temporal relations that can be specified by correlation rules, such a sequence of specific events over a given time interval. CEP systems also provide declarative interfaces using event languages like SASE [67] or following dataflow specifications.

The first generation of DSPEs provided extensions to the traditional DBMS model by enabling long-running queries over dynamic data, and by offering declarative interfaces and SQL-like languages that allowed a user to specify algebra-like operations. Most engines were restricted to a single machine and were not executed in a distributed fashion. The second generation of engines enabled distributed processing by decoupling processing entities that communicate with one another using message-passing processes. This enhanced model could take advantage of distributed hosts, but introduced challenges about load balancing and resource management. Despite the improvements in distributed execution, most engines of these two generations fall into the category of DSMSs, where queries are organised as operator graphs. IBM proposed System S [133], an engine based on data-flow graphs where users could develop operators of their own. The goal was to improve scalability and efficiency in stream processing, a problem inherent to most DSMSs. Achieving horizontal scalability while providing declarative interfaces still remained a challenge not addressed by most engines.

More recently, several DSPEs were developed to perform distributed DSP while aiming to achieve scalable and fault-tolerant execution on cluster environments. Many of these engines do not provide declarative interfaces, requiring a developer to program applications rather than write queries. Most engines follow a one-pass processing model where the application is designed as a data-flow graph. Data items of an input stream, when received, are forwarded throw a graph of processing elements, which can, in turn, create new streams that are redirected to other elements. These engines allow for the specification of User Defined Functions (UDFs) to be performed by the processing elements when an application is deployed. Another model that has gained popularity consists in discretising incoming data streams and launching periodical micro-batch executions. Under this model, data received from an input stream is stored during a time window, and towards the end of the window, the engine triggers distributed batch processing. Some systems trigger recurring queries upon bulk appends to data streams [71]. This model aims to improve scalability and throughput for applications that do not have stringent requirements regarding processing delays.

We are currently witnessing the emergence of a fourth generation of DSPE, where certain

processing elements are placed on the edges of the network. Architectural models [119], DSPEs [39, 109], and engines for certain application scenarios such as IoT are emerging. Architecture that mixes elements deployed on edge computing resources and the cloud is provided in the literature [39, 77, 119].

The generations of DSPEs are summarised in Figure 2.5. This thesis focuses on state-of-the-art frameworks and technology for DSP and solutions for exploiting resource elasticity for DSPEs that accept UDFs. We discuss the third generation of DSPEs, but focus on challenges inherent to the fourth.

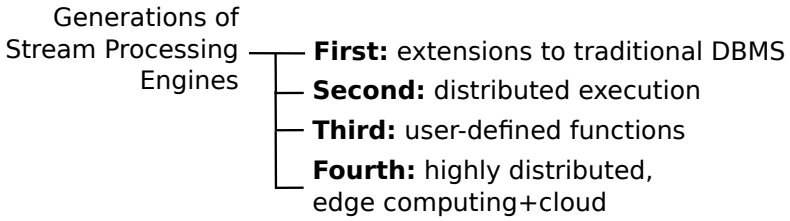

Figure 2.5: Generations of DSPEs.

2.3 Data Stream Processing Engines and Tools

While the first generation of DSPEs were analogous to DBMSs, developed to perform long running queries over dynamic data and consisted essentially of centralised solutions, the second generation introduced distributed processing and revealed challenges on load balancing and resource management. The third generation of solutions resulted in more general application frameworks that enable the specification and execution of UDFs. This section discusses third-generation solutions that enable the processing of unbounded data streams across multiple hosts and the execution of UDFs. Numerous frameworks have been proposed for distributed processing following essentially two models (Figure 2.6):

- the *operator-graph* model described earlier, where a processing system is continuously ingesting data that is processed at a by-tuple level by a DAG of operators; and
- a *micro-batch* in which incoming data is grouped during short intervals, thus triggering a batch processing towards the end of a time window. The rest of this section provides a description of select systems that fall into these two categories.

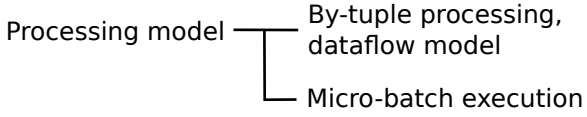

Figure 2.6: Data stream processing models.

2.3.1 Apache Storm

An application in Apache Storm [136], also called a *Topology*, is a computation graph that defines the processing elements (*i.e.* *Spouts* and *Bolts*) and how the data (*i.e.* *tuples*) flows between them. A topology runs indefinitely, or until a user stops it. Similarly to other application

models, a topology receives an influx of data and divides it into chunks that are processed by tasks assigned to cluster nodes. The data that nodes send to one another is in the form of sequences of *Tuples*, which are ordered lists of values.

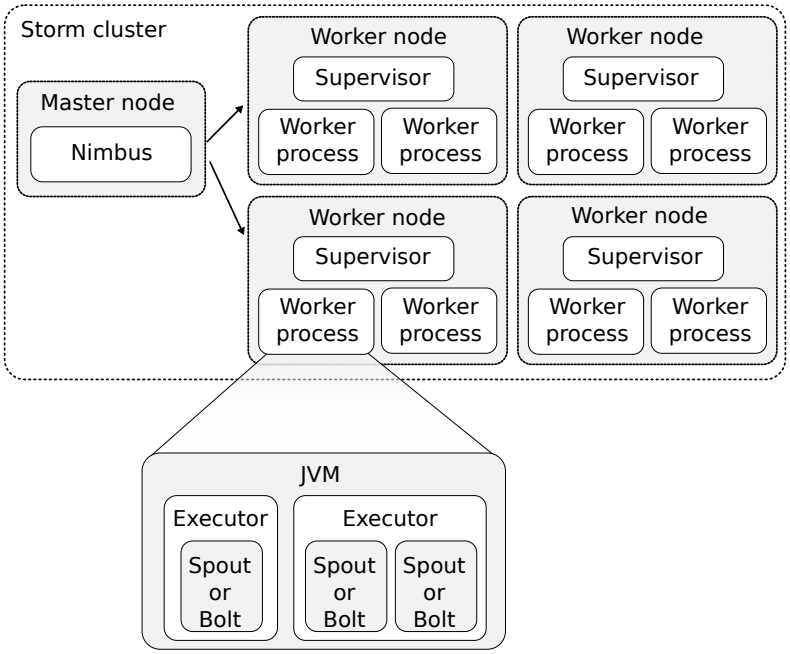

Figure 2.7: Main components of an Apache Storm cluster [3].

Figure 2.7 depicts the main components of an Apache Storm cluster [3]. Apache Storm uses a master-slave execution architecture where a *Master Node*, which runs a daemon called *Nimbus*, is responsible for scheduling tasks among *Worker Nodes* and for maintaining a membership list to ensure reliable data processing. Nimbus interacts with Zookeeper [17] to detect node failure and reassign tasks accordingly if needed. An Apache Storm cluster comprises multiple worker nodes, each worker representing a virtual or physical machine. A worker node runs a *Supervisor* daemon, and one or multiple *Worker Processes*, which are processes (*i.e.* a JVM) spawned by Apache Storm and able to run one or more *Executors*. An executor thread executes one or more tasks. A *Task* is both a realisation of a topology node and an abstraction of a Spout or Bolt. A Spout is a data stream source; it is the component responsible for reading the data from an external source and generating the data influx processed by the topology nodes. A Bolt listens to data, accepts a tuple, performs a computation or transformation – *e.g.* filtering, aggregation, joins, databases queries, and other UDFs – and optionally emits a new tuple.

Apache Storm has many configuration options to define how topologies make use of host resources. An administrator can specify the number of worker processes that a node can create, also termed slots, as well as the amount of memory that slots can use. To parallelise nodes of an Apache Storm topology a user needs to provide hints on how many concurrent tasks each topology component should run or how many executors to use; the latter influences how many threads will execute spouts and bolts. Tasks resulting from parallel Bolts perform the same function over different sets of data but may execute in different machines and receive data from different sources. Apache Storm's scheduler, which is run by the Master, assigns tasks to workers in a round-robin fashion.

Apache Storm allows for new worker nodes to be added to an existing cluster on which

new topologies and tasks can be launched. It is also possible to modify the number of worker processes and executors spawned by each process. Modifying the level of parallelism by increasing or reducing the number of tasks that a running topology can create or the number of executors that it can use is more complex and, by default, requires the topology to be stopped and rebalanced. Such operation is expensive and can incur a considerable downtime. Moreover, some tasks may maintain state, perform grouping or hashing of tuple values that are henceforth assigned to specific downstream tasks. Stateful tasks complicate the dynamic adjustment of a running topology even further. As described in Section 2.5, existing work has attempted to circumvent some of these limitations to enable resource elasticity.

2.3.2 Twitter Heron

While maintaining API compatibility with Apache Storm, Twitter's Heron [87] was built with a range of architectural improvements and mechanisms to achieve better efficiency and to address several of Storm issues highlighted in previous work [136]. Heron topologies are process-based with each process running in isolation, which eases debugging, profiling, and troubleshooting. By using its built-in back pressure mechanisms, topologies can self-adjust when certain components lag.

Similarly to Storm, Heron topologies are directed graphs whose vertices are either *Spouts* or *Bolts* and edges represent streams of *tuples*. The data model consists of a *logical plan*, which is the description of the topology itself and is analogous to a database query; and the *physical plan* that maps the actual execution logic of a topology to the physical infrastructure, including the machines that run each spout or bolt. When considering the execution model, Heron topologies comprise the following main components: *Topology Master*, *Container*, *Stream Manager*, *Heron Instance*, *Metrics Manager*, and *Heron Tracker*.

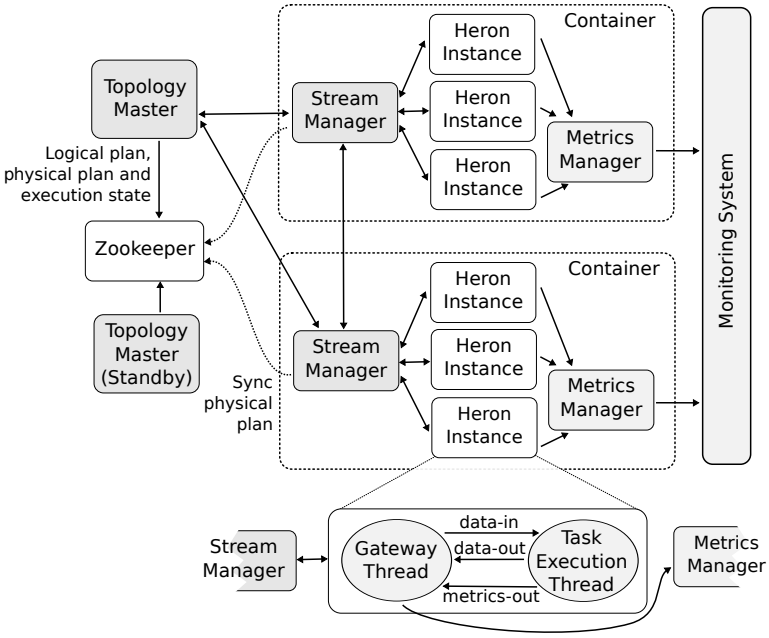

Figure 2.8: Main architecture components of a Heron topology [87].

Heron provides a command-line tool for submitting topologies to the *Aurora Scheduler*, a scheduler built to run atop Mesos [76]. Heron can also work with other schedulers including

YARN, and Amazon EC2 Container Service (ECS) [5]. Support to other schedulers is enabled by an abstraction designed to avoid the complexity of Storm Nimbus, often highlighted as an architecture issue in Storm. A topology in Heron runs as an Aurora job that comprises multiple *Containers*.

When a topology is deployed, Heron starts a single *Topology Master (TM)* and multiple containers (Figure 2.8). The TM manages the topology throughout its entire life cycle until a user deactivates it. Zookeeper [17] is used to guarantee that there is a single TM for the topology and that it is discoverable by other processes. The TM also builds the physical plan and serves as a gateway for topology metrics. Heron allows for creating a StandBy TM in case the main TM fails. Containers communicate with the TM hence forming a fully connected graph. Each container hosts multiple *Heron Instances (HIs)*, a *Stream Manager (SM)*, and a *Metrics Manager (MM)*. An SM manages the routing of tuples, whereas SMs in a topology form a fully connected network. Each HI communicates with its local SM when sending and receiving tuples. The work for a spout and a bolt is carried out by HIs, which unlike Storm workers, are JVM processes. An MM gathers performance metrics from components in a container, which are in turn routed both to the TM and external collectors. An *Heron Tracker (HT)* is a gateway for cluster-wide information about topologies.

An HI follows a two-threaded design with one thread responsible for executing the logic programmed as a spout or bolt (*i.e.* Execution), and another thread for communicating with other components and carrying out data movement in and out of the HI (*i.e.* Gateway). The two threads communicate with one another via three unidirectional queues, of which two are used by the Gateway to send/receive tuples to/from the Execution thread, and another is employed by the Execution thread to export collected performance metrics.

2.3.3 Apache S4

The Simple Scalable Streaming System (S4) [102] is a distributed DSPE that uses the actor model for managing concurrency. Processing Elements (PEs) perform computation and exchange events, where each PE can handle data events and either emit new events or publish results.

S4 can use commodity cluster hardware and employs a decentralised and symmetric runtime architecture comprising Processing Nodes (PNs) that are homogeneous concerning functionality. As depicted in Figure 2.9, a PN is a machine that hosts a container of PEs that receive events, execute user-specified functions over the data, and use the communication layer to dispatch and emit new events. ZooKeeper [17] provides features used for coordination between PNs.

When developing a PE, a developer must specify its functionality and the type of events it can consume. While most PEs can only handle events with given keyed attribute values, S4 provides a keyless PE used by its input layer to handle all events that it receives. PNs route events using a hash function of their keyed attribute values. Following receipt of an event, a listener passes it to the processing element container that in turn delivers it to the appropriate PEs.

2.3.4 Apache Flink

Flink offers a common runtime for DSP and batch processing applications [12]. Applications are structured as arbitrary DAGs, where special cycles are enabled via iteration constructs. Flink works with the notion of *streams* onto which *transformations* are performed. A stream is an intermediate result, whereas a transformation is an operation that takes one or more streams as input, and computes one or multiple streams. During execution, a Flink application is mapped to

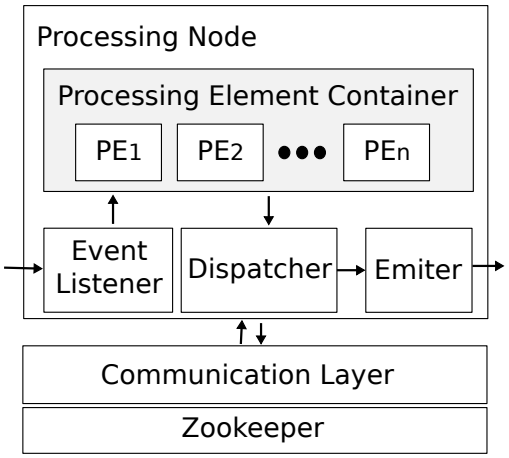

Figure 2.9: A processing node in S4 [102].

a *streaming workflow* that starts with one or more *sources*, comprises *transformation operators*, and ends with one or multiple *sinks*. Although there is often a mapping of one transformation to one dataflow operator, under certain cases, a transformation can result in multiple operators. Flink also provides APIs for iterative graph processing, such as Gelly [13].

The parallelism of Flink applications is determined by the degree of parallelism of streams and individual operators. Streams can be divided into *stream partitions* whereas operators are split into *subtasks*. Operator subtasks are executed independently from one another in different threads that may be allocated to different containers or machines.

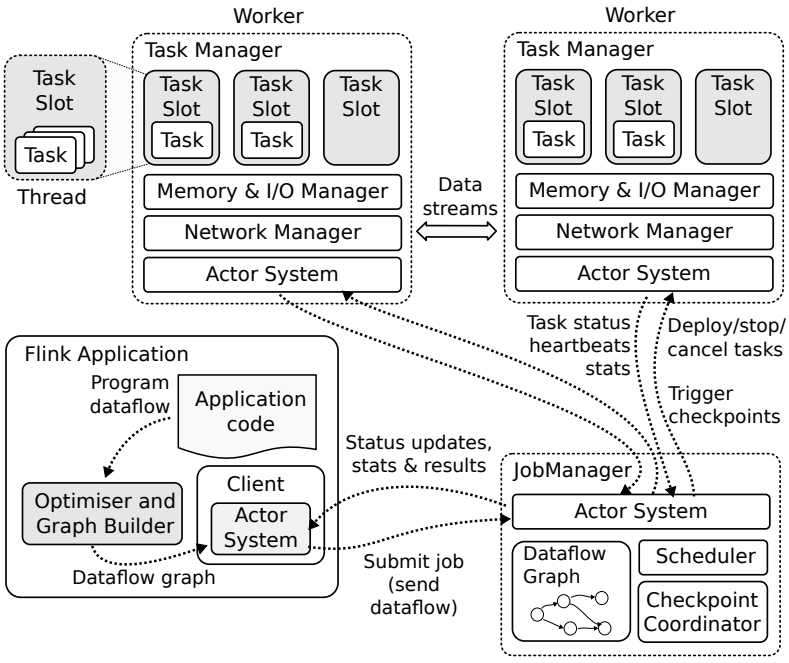

Figure 2.10: Apache Flink's execution model [12].

Flink's execution model (Figure 2.10) comprises two types of processes, namely a master also called the *JobManager* and workers termed as *TaskManagers*. The JobManager is respon-

sible for coordinating the scheduling tasks, checkpoints, failure recovery, among other functions. TaskManagers execute subtasks of a Flink dataflow. They also buffer and exchange data streams. A user can submit an application using the Flink client, which prepares and sends the dataflow to a JobManager.

Similar to Storm, a Flink worker is a JVM process that can execute one or more subtasks in separate threads. The worker also uses the concept of slots to configure how many execution threads can be created. Unlike Storm, Flink implements its memory management mechanism that enables a fair share of memory that is dedicated to each slot.

2.3.5 Spark Streaming

Apache Spark is a cluster computing solution that extends the MapReduce model to support other types of computations such as interactive queries and DSP [151]. Designed to cover a variety of workloads, Spark introduces an abstraction called Resilient Distributed Datasets (RDDs) that enables running computations in memory in a fault-tolerant manner. RDDs, which are immutable and partitioned collections of records, provide a programming interface for performing operations, such as map, filter and join, over multiple data items. For fault-tolerance purposes, Spark records all transformations carried out to build a dataset, thus forming the so-called *lineage graph*.

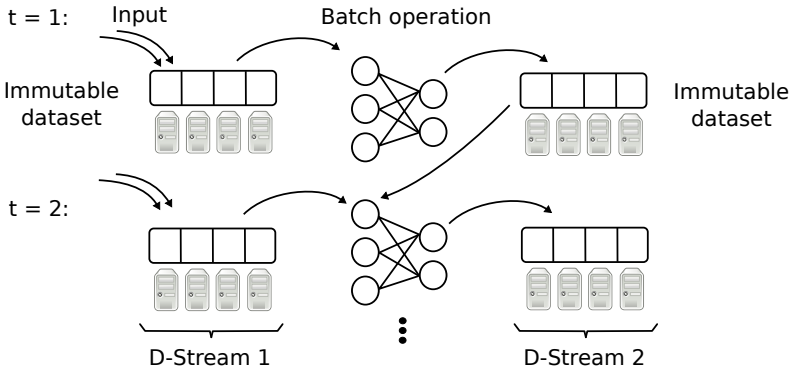

Figure 2.11: D-Stream processing model [152].

Under the traditional stream processing approach based on a graph of continuous operators that process tuples as they arrive, it is arguably difficult to achieve fault tolerance and handle stragglers. As application state is often kept by multiple operators, fault tolerance is achieved either by replicating sections of the processing graph or via upstream backup. The former demands synchronisation of operators via a protocol such as Flux [126] or other transactional protocols [142], whereas the latter, when a node fails, requires parents to replay previously sent messages to rebuild the state.

To handle faults and stragglers more efficiently, Zaharia *et al.* [152] proposed D-Streams, a discretised DSP based on Spark Streaming. As depicted in Figure 2.11, D-Streams follows a micro-batch approach that organises DSP as batch computations carried out periodically over small time windows. During a short time interval, D-Streams stores the received data, which the cluster resources then use as input dataset for performing parallel computations once the interval elapses. These computations produce new datasets that represent an intermediate state or computation outputs. The intermediate state consists of RDDs that D-Streams processes along with the datasets stored during the next interval. In addition to providing a strong

unification with batch processing, this model stores the state in memory as RDDs [151] that D-Streams can deterministically recompute.

2.3.6 Other Solutions

System S, a precursor to IBM Streams[2], is a middleware that organises applications as DAGs of operators and that supports distributed processing of both structured and unstructured data streams. Stream Processing Language (SPL) offers a language and engine for composing distributed and parallel data-flow graphs and a toolkit for building generic operators [77]. It provides language constructs and compiler optimisations that utilise the performance of the Stream Processing Core (SPC) [7]. SPC is a system for designing and deploying DSP DAGs that support both relational operators and user-defined operators. It places operators on containers that consist of processes running on cluster nodes. The SPC data fabric provides the communication substrate implemented on top of a collection of distributed servers.

Esc [124] is another DSPE that also follows the data-flow scheme where programs are DAGs whose vertices represent operations performed on the received data and edges are the composition of operators. The Esc system, which uses the actor model for concurrency, comprises a system and multiple machine processes responsible for executing workers.

Other systems, such as TimeStream [111], use a DAG abstraction for structuring an application as a graph of operators that execute user-defined functions. Employing a graph abstraction is not exclusive to DSP. Other big data processing frameworks [118] also provide high-level APIs that enable developers to specify computations as a DAG. The deployment of such computations is performed by engines using resource management systems such as Apache YARN.

Google's MillWheel [2] also employs a data flow abstraction in which users specify a graph of transformations, or computations, that are performed on input data to produce output data. MillWheel applications run on a dynamic set of hosts where each computation can run on one or more machines. A master node manages load distribution and balancing by dividing each computation into a set of key intervals. Resource utilisation is continuously measured to determine increased pressure, in which case intervals are moved, split, or merged.

The Efficient, Lightweight, Flexible (ELF) DSP system [80] uses a decentralised architecture with 'in-situ' data access where each job extracts data directly from a Web server, placing it in compressed buffer trees for local parsing and temporary storage. The data is subsequently aggregated using shared reducer trees mapped to a set of worker processes executed by agents structured as an overlay built using Pastry Distributed Hash Table (DHT). ELF attempts to overcome some of the limitations of existing solutions that require data movement across machines and where the data must be somewhat stale before it arrives at the DSP system.

Apache Edgent [11], SensorBee [125], and Apache Nifi [15] are lightweight DSPEs designed to be employed in gateways and small footprint edge devices enabling local, real-time, analytics on the continuous streams of data coming from sensors or any device. Working in conjunction with centralised analytic systems, they provide efficient and timely analytics across the whole infrastructure ecosystem.

[2]IBM has rebranded its DSP solution a few times over the years. Although some papers mention System S and InfoSphere Streams, hereafter we employ simply *IBM Streams* to refer to IBM's DSP solution.

2.4 Managed Cloud Systems

This section describes public cloud solutions for processing streaming data and presents details on how elasticity features are made available to developers and end users. The section primarily identifies prominent technological solutions for processing of streaming data and highlights their main features.

2.4.1 Amazon Web Services (AWS) Kinesis

A streaming data service can use Firehose for delivering data to AWS services such as Amazon Redshift, Amazon Simple Storage Service (S3), or Amazon Elasticsearch Service (ES). It works with data producers or agents that send data to Firehose, which in turn delivers the data to the user-specified destination or service. When choosing S3 as the destination, Firehose copies the data to an S3 bucket. Under Redshift, Firehose first copies the data to an S3 bucket before notifying Redshift. Firehose can also deliver the streaming data to an ES cluster.

Firehose works with the notion of *delivery streams* to which *data producers* or agents can send data *records* of up to 1000 KB in size. Firehose buffers incoming data up to a *buffer size* or for a given *buffer interval* in seconds before it delivers the data to the destination service. Integration with the Amazon CloudWatch [4] enables monitoring the number of bytes transferred, the number of records, the success rate of operations, time taken to perform certain operations on delivery streams, among others. AWS enforces certain limits on the rate of bytes, records and number of operations per delivery stream, as well as streams per region and AWS account.

Amazon Kinesis Streams is a service that enables continuous data intake and processing for several types of applications such as data analytics and reporting, infrastructure log processing, and complex event processing. Under Kinesis Streams *producers* continuously push data to Streams, which is then processed by *consumers*. A *stream* is an ordered sequence of *data records* that are distributed into *shards*. A Kinesis Streams *application* is a consumer of a stream that runs on Amazon Elastic Compute Cloud (EC2). A shard has a fixed data capacity regarding reading operations and the amount of data read per second. The total capacity of a stream is the aggregate capacity of all of its shards. Integration with Amazon CloudWatch allows for monitoring the performance of the available streams. A user can adjust the capacity of a stream by *resharding* it. Two operations are allowed for respectively increasing or decreasing available capacity, namely splitting an existing shard or merging two shards.

2.4.2 Google Dataflow

Google Cloud Dataflow [61] is a programming model and managed service for developing and executing a variety of data processing patterns such as Extract, Transform, and Load (ETL) tasks, batch processing, and continuous computing.

Dataflow's programming model enables a developer to specify a data processing job that is executed by the Cloud Dataflow runner service. A data processing job is specified as a *Pipeline* that consists of a directed graph of steps or *Transforms*. A transform takes one or more *PCollection*'s – that represent data sets in the pipeline – as input, performs the user-provided processing function on the elements of the PCollection and produces an output PCollection. A PCollection can hold data of a fixed size, or an unbounded data set from a continuously updating source. For unbounded sources, Dataflow enables the concept of *Windowing* where elements of the PCollection are grouped according to their timestamps. A *Trigger* can be

specified to determine when to emit the aggregate results of each window. Data can be loaded into a Pipeline from various *I/O Sources* by using the Dataflow SDKs as well as written to output *Sinks* using the sink APIs. As of writing, the Dataflow SDKs are being open sourced under the Apache Beam incubator project [10].

The Cloud Dataflow managed service can be used to deploy and execute a pipeline. During deployment, the managed service creates an execution graph, and once deployed the pipeline becomes a Dataflow job. The Dataflow service manages services such as Google Compute Engine [63] and Google Cloud Storage [62] to run a job, allocating and releasing the necessary resources. The performance and execution details of the job are made available via the Monitoring Interface or using a command-line tool. The Dataflow service attempts to perform certain automatic job optimisations such as data partitioning and parallelisation of worker code, optimisations of aggregation operations or fusing transforms in the execution graph.

On-the-fly adjustment of resource allocation and data partitioning are also possible via Autoscaling and Dynamic Work Rebalancing. For bounded data in batch mode Dataflow chooses the number of VMs based on both the amount of work in each step of a pipeline and the current throughput. Although autoscaling can be used by any batch pipeline, as of writing autoscaling for streaming-mode is experimental and participation is restricted to invited developers. It is possible, however, to adjust the number of workers assigned to a streaming pipeline manually, which replaces a running job with a new job while preserving the state information.

2.4.3 Azure Stream Analytics

Azure Stream Analytics (ASA) enables real-time analysis of streaming data from several sources such as devices, sensors, websites, social media, applications, infrastructures, among other sources [25].

A job definition in ASA comprises data *inputs*, a *query*, and data *output*. Input is the data streaming source from which the job reads the data, a query transforms the received data, and the output is to where the job sends results. Stream Analytics provides integration with multiple services and can ingest streaming data from Azure Event Hubs and Azure IoT Hub, and historical data from Azure Blob service. It performs analytic computations that are specified in a declarative language; a T-SQL variant termed as Stream Analytics Query Language. Results from Stream Analytics can be written to several data sinks such as Azure Storage Blobs or Tables, Azure SQL DB, Event Hubs, Azure Service Queues, among other sinks. They can also be visualised or further processed using other tools deployed on Azure compute cloud. As of writing, Stream Analytics does not support UDFs for data transformation.

The allocation of processing power and resource capacity to a Stream Analytics job is performed considering Streaming Units (SUs) where an SU represents a blend of CPU capacity, memory, and read/write data rates. Certain query steps can be partitioned, and some SUs can be allocated to process data from each partition, hence increasing throughput. To enable partitioning the input data source must be partitioned and the query modified to read from a partitioned data source.

2.5 Elasticity in Data Stream Processing Systems

Cloud computing is a model under which organisations of all sizes can lease IT resources and services on-demand and pay as they go [18]. Resources allocated to customers are often Virtual Machines (VMs) or containers that share the underlying physical infrastructure, which allows

for workload consolidation that can hence lead to better system utilisation and energy efficiency [131]. Another important feature of clouds is *resource elasticity*, which enables organisations to change infrastructure capacity dynamically with the support of *auto-scaling operations*. This capability is essential in several settings as it helps service providers: to minimise the number of allocated resources and to deliver adequate Quality of Service (QoS) levels, usually synonymous with low response times.

In addition to deciding when to modify the system capacity, auto-scaling algorithms must identify adequate step-sizes (*i.e.* the number of resources by which the cloud should shrink and expand) during scale out/in operations in order to prevent resource wastage and unacceptable QoS [101]. An elastic system requires not only mechanisms that adjust service execution to current resource capacity – *e.g.* present horizontal scalability – but also an *auto-scaling policy* that defines when and by how much resource capacity is added or removed.

When considering solutions for managing elasticity of DSP, this section discusses the techniques and metrics employed for monitoring the performance of DSP systems and the actions carried out during auto-scaling operations. The actions performed during auto-scaling operations include, for instance adding/removing computing resources and adjusting the DSP application by changing the level of parallelism of certain processing operators, adjusting the processing graph, merging or splitting operators, (re)assigning operator to computing resources, among other issues.

Over time several types of applications have benefited from resource elasticity, a key feature of cloud computing [93]. As highlighted by Lorido-Botran *et al.*, elasticity in cloud environments is often accomplished via a *Monitoring, Analysis, Planning and Execution (MAPE)* process where:

1. application and system metrics are *monitored*;
2. the gathered information is *analysed* to assess current performance and utilisation, and optionally predict future load;
3. based on an auto-scaling policy an auto-scaler creates an elasticity *plan* on how to add or remove capacity; and
4. the plan is finally *executed*.

After analysing performance data, an auto-scaler may choose to adjust the number of resources (*e.g.* add or remove compute resources) available to running, newly submitted, applications. Managing elasticity of DSP applications often requires solving two inter-related problems: (i) allocating or releasing IT resources to match an application workload; and (ii) devising and performing actions to adjust the application to make use of the additional capacity or release previously allocated resources. The first problem, which consists in modifying the resource pool available for a DSP application, is termed here as *elastic resource management*. A decision made by a resource manager to add/remove resource capacity for a DSP application is referred to as *scale out/in plan*[3]. We refer to the actions taken to adjust an application during a scale out/in plan as *elasticity actions*.

Similarly to other services running in the cloud, elastic resource management for DSP applications can make use of two types of elasticity, namely *vertical* and *horizontal* (Figure 2.12), which have their impact on the kind of elastic actions for adapting an application. Vertical

[3]The term scale out/in is often employed in horizontal elasticity, but a plan can also be scale up/down when using vertical elasticity. For brevity, we use only scale out/in in the rest of the text

elasticity consists in allocating more resources such as CPU, memory and network capacity on a host that has previously been allocated to a given application. As described later, DSP can benefit from this type of elasticity by, for instance, increasing the instances of a given operator (*i.e.* operator *fission* [78]). Horizontal elasticity consists essentially in allocating additional computing nodes to host a running application.

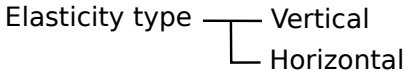

Figure 2.12: Types of elasticity used by elastic resource management.

To make use of additional resources and improve application performance, auto-scaling operations may require adjusting applications dynamically by, for example, performing optimisations in their execution graphs, or modifying intra-query parallelism by increasing the number of instances of certain operators, or (re)assigning operators. Previous work has discussed approaches on reconfiguration schemes to modify the placement of DSP operators dynamically to adjust an application to current resource conditions or provide fault-tolerance [88]. The literature on DSP often employs the term *elastic* to convey *operator placement schemes* that enable applications to deliver steady performance as their workload increases, not necessarily exploring the types of elasticity mentioned above.

Although the execution of scale out/in plans presents similarities with other application scenarios (*e.g.* adding/removing resources from a resource pool), adjusting a DSP system and applications dynamically to make use of the newly available capacity or release unused resources is not a trivial task. The enforcement of scale out/in plans faces multiple challenges. Horizontal elasticity often requires adapting the graph of processing elements and protocols, exporting and saving operator state for replication, fault tolerance and migration. As highlighted by Sattler and Beier [122], performing parallel processing is often difficult in the case of window- or sequence-based operators including CEP operators due to the amount of state they keep. Elastic operations, such as adding nodes or removing unused capacity, may require at least re-routing the data, changing the manner an incoming dataflow is split among parallel processing elements, among other issues. Such adjustments are costly to perform, particularly if processing elements maintain state. As DSP queries are often treated as long running that cannot be restarted without incurring a loss of data, the initial operator placement (also called task assignment), where processing elements are deployed on available computing resources becomes more critical than in other systems.

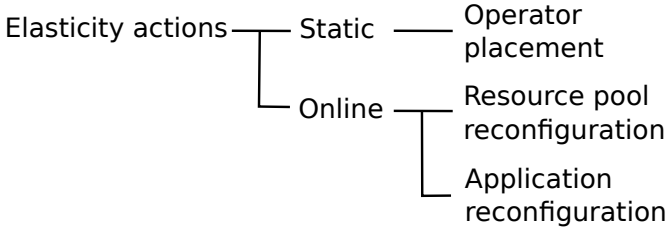

Figure 2.13: Elasticity actions for data stream processing engines.

Given how important the initial operator placement is to guarantee the elasticity of DSP systems, we classify elasticity actions into two main categories, namely *static* and *online* as depicted in Figure 2.13. When considering the operator DAG based solutions, static techniques

comprise optimisations made to modify the original graph (*i.e.* the logical plan) to improve task parallelism and operator placement, optimise data transfers, among other goals [78]. Previous work provided a survey of various static techniques [88]. Online approaches comprise both actions to modify the pool of available resources and dynamic optimisations carried out to adjust applications dynamically to utilise newly allocated resources. The next sections provide more details on how existing solutions address challenges in these categories with a focus on online techniques.

2.5.1 Static Techniques

A review of strategies for placing processing operators in early distributed DSP systems has been presented in previous work [88]. Several approaches for optimising the initial task assignment or scheduling exploit intra-query parallelism by ensuring that certain operators can scale horizontally to support larger numbers of incoming tuples, thus achieving greater throughput.

R-Storm [107] handles the problem of task assignment in Apache Storm by providing custom resource-aware scheduling schemes. Under the considered approach, each task in a Storm topology has soft CPU and bandwidth requirements and a hard memory requirement. The available cluster nodes, on the other hand, have budgets for CPU, bandwidth and memory. While considering the throughput contribution of a data sink, given by the rate of tuples it is processing, R-Storm aims to assign tasks to a set of nodes that increases overall throughput, maximises resource utilisation, and respects resource budgets. The assignment scenario results is a quadratic multiple 3-dimensional knapsack problem. After reviewing existing solutions with several variants of knapsack problems, the authors concluded that existing methods are computationally expensive for distributed DSP scenarios. They proposed scheduling algorithms that view a task as a vector of resource requirements and nodes as vectors of resource budgets. The algorithm uses the Euclidean distance between a task vector and node vectors to select a node to execute a task. It also uses heuristics that attempt to place tasks that communicate in proximity to one another, that respect hard constraints, and that minimise resource waste.

Pietzuch *et al.* [108] create a Stream-Based Overlay Network (SBON) between a DSPE and the physical network. SBON manages operator placement while taking into account network latency. The system architecture uses an adaptive optimisation technique that creates a multi-dimensional Euclidean space, termed as the *cost space*, over which the placement is projected. Optimisation techniques such as spring relaxation are used to compute operator placement using this mathematical space. A proposed scheme maps a solution obtained using the cost space onto physical nodes.

The scheme proposed by Zhou *et al.* also [155] for the initial operator placement attempts to minimise the communication cost whereas the dynamic approach considers load balancing of scheduled tasks among available resources. The initial placement schemes group operators of a query tree into query fragments and try to minimise the number of compute nodes to which they are assigned. Ahmad and Çetintemel [1] also proposed algorithms for the initial placement of operators while minimising the bandwidth utilised in the network, even though it is assumed that the algorithms could be applied periodically.

Cardellini *et al.* [35] introduce an integer programming formulation that takes into account resource heterogeneity for the Optimal Distributed Stream Processing Problem (ODP). They propose an extension to Apache Storm to incorporate an ODP-based scheduler, which estimates network latency via a network coordination system built using the Vivaldi algorithm [44]. It has been shown, however, that assigning DSP operators to VMs and placing them across multiple

geographically distributed cloud while minimising the overall inter cloud communication cost, can often be classified as an NP-Hard problem [64] or even NP-Complete [138]. Over time, however, cost-aware heuristics have been proposed for assigning DSP operators to VMs placed across multiple clouds [40, 64].

2.5.2 Online Techniques

Systems for providing elastic DSP generally comprise two key elements:

- a subsystem that monitors how the DSP system is utilising the available resources (*e.g.* use of CPU, memory and network resources) [51] and/or other service-level metrics (*e.g.* number of tuples processed over time, tail end-to-end latency [74], critical paths [139]) and tries to identify bottleneck operators; and
- a scaling policy that determines when scale out/in plans should be performed [91].

As mentioned earlier, in addition to adding/removing resources, a scale out/in plan is backed by mechanisms to adjust the query graph to make efficient use of the updated resource pool. Proposed mechanisms consist of, for instance, increasing operator parallelism; rewriting the query graph based on certain patterns that are empirically proven to improve performance and rewriting rules specified by the end user; and migrating operators to less utilised resources. The mechanisms can be employed on centralised (*e.g.*, cluster and cloud) or highly distributed (*e.g.*, cloud-edge) infrastructure. This section provides a non-exhaustive list of work regarding approaches to operator placement and application reconfiguration on centralised and highly distributed infrastructure.

Centralised Infrastructure

Most solutions are application and workload agnostic – *i.e.* do not attempt to model application behaviour or detect changes in the incoming workload [86] – and offer methods to: (i) optimise the initial scheduling, when processing tasks are assigned to and deployed onto available resources; and/or (ii) reschedule processing tasks dynamically to take advantage of an updated resource pool. Operators are treated as black boxes and (re)scheduling and elastic decisions are often taken considering a performance metric. Certain solutions that are not application-agnostic attempt to identify workload busts and behaviours by considering characteristics of the incoming data.

Sattler and Beier [122] argue that distributing query nodes or operators can improve reliability *"by introducing redundancy, and increasing performance and/or scalability by load distribution"*. They identify operator patterns – *e.g.* simple standby, check-pointing, hot standby, stream partitioning and pipelining – for building rules for restructuring the physical plan of an application graph, which can increase fault tolerance and achieve elasticity. They advocate that re-writings should be performed when a task becomes a bottleneck; *i.e.* it cannot keep up with the rate of incoming tuples. An existing method is used to scan the execution graph and find critical paths based on monitoring information gathered during query execution [139].

While dynamically adjusting queries with stateless operators can be difficult, modifying a graph of stateful operators to increase intra-query parallelism is more complex. As stated by Fernandez *et al.* [51], during adjustment, operator *"state must be partitioned correctly across a larger set of VMs"*. Fernandez *et al.* hence propose a solution to manage operator state, which

they integrate into a DSPE to provide *scale out* features. The solution offers primitives to export operator state as a set of tuples, which is periodically check-pointed by the processing system. An operator keeps state regarding its processing, buffer contents, and routing table. During a scale out operation, the key space of the tuples that an operator handles is repartitioned, and its processing state is split across the new operators. The system measures CPU utilisation periodically to detect bottleneck operators. If multiple measurements are above a given threshold, then the scale-out coordinator increases the operator parallelism.

Previous work has also attempted to improve the assignment of tasks and executors to available resources in Storm and to reassign them dynamically at runtime according to resource usage conditions. T-Storm [143] (*i.e.* Traffic-aware Storm), for instance, aims to reduce inter-process and inter-node communication, which is shown to degrade performance under certain workloads. T-Storm monitors workload and traffic load during runtime. It provides a scheduler that generates a task schedule periodically, and a custom Storm scheduler that fetches the schedule and executes it by assigning executors accordingly. Aniello *et al.* provide a similar approach, with two custom Storm schedulers, one for offline static task assignment and another for dynamic scheduling [8]. Performance monitoring components are also introduced, and the proposed schedulers aim to reduce inter-process and inter-node communication.

Lohrmann *et al.* [91] introduce policies that use application or system performance metrics such as CPU utilisation thresholds, the rate of tuples processed per operator, and tail end-to-end latency. They propose a strategy to provide latency guarantees in DSP systems that execute heady UDF data flows while aiming to minimise resource utilisation. The reactive strategy (*i.e.* *ScaleReactively*) aims to enforce latency requirements under varying load conditions without permanently overprovisioning resource capacity. The proposed solution assumes homogeneous cluster nodes, effective load balancing of elements executing UDFs, and elastically scalable UDFs. The system architecture comprises elements for monitoring the latency incurred by operators in a job sequence. The reactive strategy uses two techniques, namely *Rebalance* and *ResolveBottlenecks*. The former adjusts the parallelism of bottleneck operators whereas the latter, as the name implies, resolves bottlenecks by scaling out so that the first technique can be applied again at later time.

The Esc DSP system [124] comprises several components for task scheduling, performance monitoring, management of a resource pool to/from which machines are added/released, as well as application adaptation decisions. A processing element process executes UDFs and contains a manager and multiple workers, which serve respectively as a gateway for the element itself and for executing multiple instances of the UDF. The PE manager employs a function for balancing the load among workers. Each worker contains a buffer or queue and an operator. The autonomic manager of the system process monitors the load of machines and the length of the worker processes. For adaptation purposes, the autonomic manager can add/remove machines, replace the load balancing function of a PE manager and spawn/kill new workers, kill the PE manager and its workers altogether. The proposed elastic policies are based on load thresholds that, when exceeded, trigger the execution of actions such as attaching new machines.

StreamCloud (SC) [65] provides multiple cloud parallelisation techniques for splitting DSP queries that it assigns to independent subclusters of computing resources. According to the chosen technique, the number of resulting subqueries depends on the number of stateful operators that the original query contains. A subquery comprises a stateful operator and all intermediate stateless operators until another stateful operator or a data sink. SC also introduces *buckets* that receive output tuples from a subcluster. Bucket-Instance Maps (BIMs) control the distribution of buckets to downstream subclusters, which may be dynamically modified by Taneja et. al.s

(LBs). A load balancer is an operator that distributes tuples from a subquery to downstream subqueries. To manage elasticity, SC employs a resource management architecture that monitors CPU utilisation and, if the utilisation is out of pre-determined lower or upper thresholds, it can: adjusts the system to rebalance the load; or provision or releases resources.

Heinze *et al.* [74] attempt to model the spikes in a query's end-to-end latency when moving operators across machines, while trying to reduce the number of latency violations. Their target system, FUGU, considers two classes of scaling decisions, namely mandatory, which are operator movements to avoid overload; and optional, such as releasing an unused host during light load. FUGU employs the Flux protocol for migrating DSP operators [126]. Algorithms are proposed for scale out/in operations as well as operator placement. The scale-out solution extends the subset sum algorithm, where subsets of operators whose total load is below a pre-established threshold are considered to remain in a host. To pick a final set, the algorithm takes into consideration the latency spikes caused by moving the operators that are not in the set. For scale-in, FUGU releases a host with minimum latency spike. The operator placement is an incremental bin packing problem, where bins are nodes with CPU capacity, and items are operators with CPU load as weight. Memory and network are second-level constraints that prevent placing operators on overloaded hosts. A solution based on the *FirstFit* decreasing heuristic is provided.

Gedik *et al.* [57] tackle the challenge of auto-parallelising distributed DSPEs in general while focusing on IBM Streams. As defined by Gedik *et al.* [57], *"auto-parallelisation involves locating regions in the application's data flow graph that can be replicated at run-time to apply data partitioning, in order to achieve scale."* Their work proposes an elastic auto-parallelisation approach that handles stateful operators and general purpose applications. It also provides a control algorithm that uses metrics such as the blocking time at the splitter and throughput to determine how many parallel channels provide the best throughput. Data splitting for a parallel region can be performed in a round-robin manner if the region is stateless, or using a hash-based scheme otherwise.

Also considering IBM Streams, Tang and Gedik [135] address task and pipeline parallelism by determining points of a data flow graph where adding additional threads can level out the resource utilisation and improve throughput. They consider an execution model that comprises a set of threads, where each thread executes a pipeline whose length extends from a starting operator port to a data sink or the port of another thread's first operator. They use the notion of utility to model the goodness of including a new thread and propose an optimisation algorithm find and evaluating parallelisation options. Gedik *et al.* [56] propose a solution for IBM Streams exploiting pipeline parallelism and data parallelism simultaneously. They propose a technique that segments a chain-like data flow graph into regions according to whether the operators they contain can be replicated or not. For the parallelisable regions, replicated pipelines are created preceded and followed by, respectively split and merge operators.

Wu and Tan [142] discuss technical challenges that may require a redesign of distributed DSP systems, such as maintaining large amounts of state, workload fluctuation and multi-tenant resource sharing. They introduce *ChronoStream*, a system to support elasticity and high availability in latency-sensitive stream computing. To facilitate elasticity and operator migration, *ChronoStream* divides the application-level state into a collection of *computation slices* that are periodically check-pointed and replicated to multiple specified computing nodes using locality-sensitive techniques. In the case of component failure or workload redistribution, it reconstructs and reschedules slice computation. Unlike D-Streams, *ChronoStream* provides techniques for tracking the progress of computation for each slice to reduce the overhead of reconstructing if information about the lineage graph is lost from memory.

STream processing ELAsticity (Stela) is a system capable of optimising throughput after a scaling out/in operation and minimising the interruption to computation while the operation is being performed [144]. It uses Expected Throughput Percentage (ETP), which is a per-operator performance metric defined as the *"final throughput that would be affected if the operator's processing speed were changed"*. While evaluation results demonstrate that ETP performs well as a post-scaling performance estimate, the work considers stateless operators whose migration can be performed without copying large amounts of application-related data. Stela is implemented as an extension to Storm's scheduler. Scale out/in operations are user-specified and are utilised to determine which operators are given more resources or which operators lose previously allocated resources.

Hidalgo *et al.* [75] employ operator fission to achieve elasticity by creating a processing graph that increases or decreases the number of processing operators to improve performance and resource utilisation. They introduce two algorithms to determine the state of an operator, namely a *short-term* algorithm that evaluates load over short periods to detect traffic peaks; and (ii) a *long-term* algorithm that finds traffic patterns. The short-term algorithm compares the actual load of an operator against upper and lower thresholds. The long-term algorithm uses a Markov chain based on operator history to evaluate state transitions over the analysed samples to define the matrix transition. The algorithm estimates for the next time-window the probability that an operator reaches one of the three possible states (*i.e.* overloaded, underloaded, stable).

Li *et al.* [89] introduce greedy techniques for operator placement and migration in DSP systems based on Deep Reinforcement Learning for enabling model-free control in DSP systems avoiding issues of high complexity of model-based approaches (*e.g.* queueing theory). The proposed methods learn from run-time metrics without employing any mathematically solvable system model. One approach combines *Deep Reinforcement Learning* and *Deep Q Network*, but it has issues related to the action space requiring methods to reduce it. Restricting the action space can result in limited exploration of the actions and thus resulting in a sub-optimal approach with poor solutions. To solve the aforementioned issue, the authors present an *Actor-critic-based* method where the actor and critic network can be pre-trained by historical transitions samples, the training includes the reduction in the action space applying k-Nearest Neighbors for considering only actions with high Q-values.

Highly Distributed Infrastructure

In recent past, researchers and practitioners have also exploited the use of containers and lightweight resource virtualisation to perform migration of DSP operators. Pahl and Lee [106] review container technology as means to tackle elasticity in highly distributed environments comprising edge and cloud computing resources. Both containers and virtualisation technologies are useful when adjusting resource capacity during scale out/in operations, but containers are more lightweight, portable and provide more agility and flexibility when testing and deploying applications.

To support operator placement and migration in Mobile Complex Event Processing (MCEP) systems, Ottenwälder *et al.* [105] present techniques that exploit system characteristics and predict mobility patterns for planning operator-state migration in advance. The envisioned infrastructure comprises a federation of distributed brokers whose hierarchy comprises a combination of cloud and fog resources. Mobile nodes connect to the nearest broker, and each operator along with its state are kept in their own virtual machine. The problem tackled consists of finding a sequence of placements and migrations for an application graph so that the network utilisation

is minimised and the end-to-end latency requirements are met. The system performs an incremental placement where, a placement decision is enforced if its migration costs can be amortised by the gain of the next placement decision. A migration plan is dynamically updated for each operator and a *time-graph model* is used for selecting migration targets and for negotiating the plans with dependent operators to find the minimum cost plans for each operator and reserve resources accordingly. The link load created by events is estimated considering the most recent traffic measurements, while latency is computed via regular ping messages or using Vivaldi coordinates [44].

Salahuddin *et al.* [120] propose a dynamic resource allocation technique for Vehicular Ad hoc Networks. The authors argue that some companies can leverage computing resources of Vehicular Ad hoc Networks and thus provide a pay-as-you go model (*i.e.* Vehicular Cloud) for resource usage. The provisioning problem in *Vehicular Cloud* is modelled as Markov Decision Process (MDP) and solved using MATLAB. The solution allows for operator placement and migration seeking to minimise the cost of resource provisioning, the end-to-end application time and the overhead of dynamic resource provisioning.

Taneja *et al.* [134] offers a scheduling policy for application configuration on cloud and edge computing resources. The solution covers static application characteristics that consider user-defined information such as event emission rate of sensors, data processing rate of operators, among other application parameters. The policy organises computing resources and application operators in two distinct vectors, and then sorts both in ascending order of their CPU capacity and CPU requirement respectively. Then the policy iterates the operators' vector, and at each iteration, gets the computing resource on the middle of the computational vector for placing the operator. During the placement decision, the policy evaluates whether the computing resource meets or not the CPU, memory, and bandwidth requirements. For constraining scenarios, the policy identifies a computing resource which is capable of supporting the requirements. Otherwise, the computing vector is reorganised according to the residual capacity.

Mai *et al.* [94] support operator placement and application reconfiguration in DSP systems by considering either cloud and edge. The proposed solution employs a combination of Reinforcement Learning (RL) and *Neural Network*. The neural network is trained to minimise the end-to-end processing time. Similarly, Russo *et al.* [117] propose a hierarchical solution for controlling the elasticity of a DSP application on a cloud-edge infrastructure. The authors model the problem as multi-objective problem and thus employ a *full backup model-based RL*.

Sajjad and Danniswara [119] introduce a stream processing solution, *i.e.* SpanEdge, that uses central and edge data centres. SpanEdge follows a master-worker architecture with *hub* and *spoke* workers, where a *hub-worker* is hosted at a central data centre and a *spoke-worker* at an edge data centre. SpanEdge also enables global and local tasks, and its scheduler attempts to place local tasks near the edges and global tasks at central data centres to minimise the impact of the latency of Wide Area Network (WAN) links interconnecting the data centres.

Mehdipour *et al.* [95] introduce a hierarchical architecture for processing streamlined data using edge and cloud resources. They focus on minimising communication requirements between edge and cloud when processing data from IoTs devices. Shen *et al.* [127] advocate the use of Cisco's Connected Streaming Analytics (CSA) for conceiving an architecture for handling DSP queries for IoT applications by exploiting cloud and edge computing resources. CSA provides a query language for continuous queries over streams.

Geelytics [42] is a system tailored for IoT environments that comprise multiple geographically distributed data producers, result consumers, and computing resources that can be hosted either on the cloud or at the network edges. Geelytics follows a master-worker architecture

with a publish/subscribe service. Similarly to other data stream processing systems, Geelytics structures applications as DAGs of operators. Unlike other systems, however, it enables scoped tasks, where a user specifies the scope granularity of each task comprising the processing graph. The scope granularity of tasks and data-consumer scoped subscriptions are used to devise the execution plan and deploy the resulting tasks according to the geographical location of data producers.

Moira [52] is a goal-oriented framework built on top of Apache Flink for dynamically optimising resource allocation. The user must specify the application DAG and the weights of three parameters (*i.e.* throughput, latency and the monetary cost). Moira constantly monitors the environment where the application is deployed, and provides the monitored data to an optimiser method which employs Linear Least Squares for exploring the underlying relations among the performance metrics. The method provides the new reconfiguration plan. Similarly, ELYSIUM [92] profiles the application workload and hence produces accurated estimations. It employs a Q-Learning algorithm to take decisions that allow it to be reactive or proactive.

2.6 Open Issues and Positioning

Most distributed DSP systems have been traditionally designed for running locally to explore cluster of homogeneous computing resources or virtually unlimited computing resources of a single cloud service provider as presented in Section 2.3. Section 2.3 also introduced some lightweight systems conceived to deploy DSP applications on a constrained device at the edge of the Internet. None of the presented approaches can deploy or manage DSP applications across multiple cloud server providers and multiple constrained devices. On one hand, heavyweight DSP systems such as Apache Storm, Apache Flink and Apache Spark extrapolate the limited capacities of edge devices. On the other hand, lightweight systems like Apache Edgent and Apache Nifi cover only local deployments avoiding geo-distributed infrastructures.

More recently, architectural models have emerged for more distributed environments spanning multiple clouds or for exploiting the edges of the Internet (*i.e.*, edge [82, 121]). Section 2.4 presented some management systems for managing DSP applications, but they do not support dynamic deployments across the available resources because they do not have scheduling policies for handling the heterogeneity of computing resources or the high overhead imposed by the communication. Existing work aims to use the Internet edges by trying to place certain DSP elements on micro data centres (*i.e.*, cloudlets [123]) closer to where the data is generated [34], transferring events to the cloud in batches [137], or by exploiting mobile devices in the edge for DSP [99]. Proposed architecture aims to place data analysis tasks at the edge of the Internet in order to reduce the amount of data transferred from sources to the cloud, improve the end-to-end latency, or offload certain analyses from the cloud [39].

Task scheduling considering hybrid scenarios has been investigated in other domains, such as mobile clouds [54] and heterogeneous memory [53]. For stream processing, Benoit *et al.* [27] show that scheduling linear chains of processing operators onto a cluster of heterogeneous hardware is an NP-Hard problem, whereas placement of virtual computing resources and network flows onto hybrid infrastructure has also been investigated in other contexts [116]. Table 2.1 summarises a selected number of solutions that aim to provide elastic DSP. The table details the infrastructure targeted by the solutions (*i.e.* cluster, cloud, edge); the types of operators considered (*i.e.* stateless, stateful); the metrics monitored and taken into account when planning a scale out/in operation; the type of elasticity envisioned (*i.e.* vertical or horizontal); and the elasticity actions performed during the execution of a scale out/in operation.

Table 2.1: Static and online techniques for elastic data stream processing.

Solution	Infrastructure	Operator type	Metrics for Elasticity	Elasticity	
				Type	Actions
Sun *et al.* [131]	cloud	stateful	Resource use (energy consumption) and System metrics (response time)	N/A	operator placement
SBON [108]	multi-cloud	stateless	System metrics (response time)	N/A	operator placement and migration
R-Storm [107]	cloud	stateful	Resource use (CPU, memory, bandwidth) and System metrics (response)	N/A	operator placement
Fernandez *et al.* [51]	cloud	stateful	Resource use (CPU)	horizontal	operator state check-pointing, fission
T-Storm [143]	cluster	stateless	Resource use (CPU, inter-executor traffic load)	N/A	operator migration, topology rebalance
Adaptive Storm [8]	cluster	stateful	Resource use (CPU, inter-node traffic)	N/A	operator placement and migration
Nephele SPE [91]	cluster	stateless	System metrics (task and channel latency)	vertical	data batching, operator fission
Esc [124]	cloud	stateless[1]	Resource use (machine load), system metrics (queue lengths)	horizontal	replace load balancing functions dynamically, operator fission
StreamCloud (SC) [65]	cluster or private cloud[2]	stateful	Resource use (CPU)	horizontal	query splitting and placement, compiler for query parallelisation
FUGU [74]	cloud	stateful	Resource use (CPU, network and memory consumption)	horizontal	operator migration, query placement
Gedik *et al.* [57]	cluster	stateless and partitioned stateful	System metrics (congestion index, throughput)	vertical	operator fission, state check-pointing, operator migration

ChronoStream [142]	cloud	stateful	N/A	vertical and horizontal[3]	operator state check-pointing, replication, migration, parallelism
Stela [144]	cloud	stateless	System metrics (impacted throughput)	horizontal[3]	operator fission and migration
MCEP [105]	edge + cloud	stateful	System metrics (load on event streams, inter-operator latency)	N/A	operator placement and migration
Li *et al.* [89]	cloud	stateful	System metric (average end-to-end latency)	N/A	operator migration
Salahuddin *et al.* [120]	edge + cloud	stateless	System metrics (cost and overhead of resource provisioning and end-to-end latency)	horizontal and vertical	operator placement and migration
Mai *et al.* [94]	edge + cloud	stateless	System metric (end-to-end latency)	horizontal	operator migration
Taneja *et al.* [134]	edge + cloud	stateless	System metric (end-to-end latency)	N/A	operator placement
SpanEdge [119]	edge + cloud	stateless	System metric (end-to-end latency)	N/A	operator placement and migration
Geelytics [42]	edge + cloud	stateless	Resource use (network) and System metric (end-to-end latency)	N/A	operator placement
Moira [52]	edge + cloud	stateful	Resource use (network, CPU) and System metric (end-to-end latency, cost)	N/A	operator placement and migration
ELYSIUM [92]	cloud	stateful	Resource use (CPU)	horizontal and vertical	operator placement and migration
Russo *et al.* [117]	edge + cloud	stateful	System metric (monetary costs)	horizontal and vertical	operator fission, operator migration

[1] ESC experiments consider only stateless operators.
[2] Nodes must be pre-configured with StreamCloud.
[3] Execution of scale out/in operations are user-specified, not triggered by the system.

This thesis investigates mechanisms for application (re)configuration. The proposed methods seek to achieve elastic DSP systems. A target scenario is IoT, which poses challenges on how systems manipulate large volumes of data and manage and achieve scalability. The current state of DSP systems oversimplifies applications and infrastructure neglecting key characteristics and emerging requirements. For instance, existing work considers all data sinks to be located in the cloud, with no feedback loop to actuators located at the edge [34, 103]. Hence, there is a lack of solutions covering scenarios involving smart cities, precision agriculture, and smart homes comprising various heterogeneous sensors and actuators, as well as, time-constraint applications that may contain actuators often placed close to where data is collected.

The conventional approach for implementing DSP applications is to send data from all sources to the cloud for processing and data analytics. As shown in Table 2.1, there are several solutions [51, 107, 131, 143] exploring application (re)configuration on the cloud or on clusters. However, using only the cloud has limitations that impact the end-to-end application latency, network congestion, storage cost, and privacy. Moreover, cloud providers charge for computing, networking, storage resources, and messages exchanged between the edge and the cloud, making the conventional approach expensive while limiting the potential impact of IoT.

Edge services are emerging close to the data sources and can provide potential data-processing capabilities. Therefore, the edge devices can be leveraged to complement the computing capabilities of the cloud and reduce the overall latency and bandwidth requirements. Using cloud-edge infrastructure also allows for exploring different models to minimise the cost of performing data analytics.

The use of the cloud-edge infrastructure poses the following challenges:

- Deciding how to split DSP applications among the edge and cloud resources is difficult due to the operators' heterogeneity;
- Exploring heterogeneous infrastructure for deploying dataflow applications has proved to be NP-hard [27]; and
- Moving operators from cloud to edge devices is challenging due to the devices' limitations with respect to memory, CPU, and often network bandwidth [20].

Addressing the challenges above allows for shorter end-to-end application latency, a reduction in edge to cloud data transfers, costs, and can ensure efficient use of edge and cloud resources which remain relevant nowadays [72]. Many proposed solutions are oblivious to operator patterns and behaviours, edge device limitations and heterogeneous computing resources. For instance, previous work [42, 94, 119, 120, 134] neglects or does not present properly operator states in their models. Some solutions [52, 105, 117] employ stateful operators, but ignore the required memory for storing states when considering cloud-edge infrastructure even being a constraint of the devices. Also some solutions for application reconfiguration [94, 120] focus on stateless operators when deciding how to reorganise the application neglecting the time and bandwidth requirements for the operator migration. Moreover, current models [52, 92, 120] for application (re)configuration forget most of the existing operator patterns (*e.g.*, selectivity and operator data transformation), which have a direct impact over the communication overhead.

This thesis proposes a model for application (re)configuration covering heterogeneous operator behaviours and computing resources covering the aforementioned challenges and the current issues of the state-of-the-art. The model takes into account constraints such as memory and bandwidth, which many methods are oblivious as they are designed to meet cloud requirements

or are over-simplified. The model explores smart scenarios that comprehend on-line data processing architecture that incorporates, for instance, a messaging system based on IoT Hubs. The model introduces a holistic approach to the DSP application (re)configuration problem, building a flexible system capable of handling multiple QoS metrics.

This thesis employs the proposed model and offers strategies and algorithms for managing DSP application (re)configuration on cloud-edge infrastructure. The solutions explore techniques such as Series-Parallel-Decomposable Graphs (SPDG) from graph theory to establish the application configuration, and Monte-Carlo Tree Search (MCTS) based algorithms and Q-learning of RL to investigate reconfiguration. Current models for employing RL to the application (re)configuration problem oversimplify the operator behaviours [117, 120] or cover a single performance metric [92]. Previous work [92, 117] exacerbates the problem since Q-Learning requires maintaining a lookup table. The proposed reconfiguration solution extends the state-of-the-art modelling a MCTS algorithm as a multi-optimisation problem and proposing enhancements to the algorithm.

2.7 Conclusion

Both academia and industry have shown interest in DSP elasticity, as well as (re)configuring DSP applications to satisfy QoS requirements with minimal resource cost. These topics have received extensive attention in the literature – a work that has paved the way to self-adaptive deployment by proposing techniques such as elastic resource scaling, and dynamic operator scheduling.

In this chapter, we presented a comprehensive taxonomy and survey on DSP elasticity and application (re)configuration. Following the taxonomy, we discussed existing work in detail and compared the strengths and weaknesses of different methods. As a result, we identified open issues in DSP elasticity and application (re)configuration on an emerging scenario, *i.e.*, cloud-edge infrastructure. Among the various issues involving the subject, we opt for investigating how to reconfigure a DSP application among available computing resources while meeting QoS metrics of IoT scenarios. In the next chapter, we present the first step of our investigation, which is a mathematical model of the application (re)configuration problem on cloud-edge infrastructure.

Chapter 3

Modelling Data Stream Processing Using Queueing Theory

Contents

3.1 Introduction

As discussed in Section 2.3, Data Stream Processing Engines (DSPEs) often use simple policies to (re)configure operator tasks onto available compute resources. The schedulers of these systems also consider homogeneous resources, commonly assigning tasks to resources in a round-robin fashion while ignoring intricacies of operators and resources. This often results in poor performance and load imbalances when deploying Data Stream Processing (DSP) applications on heterogeneous and highly-distributed infrastructure. Current work proposes placement strategies considering user intervention [119] and many models do not support memory and communication constraints [79].

There is also a lack of solutions on application reconfiguration where the entire application life-cycle is taken into account, during which the load can change, devices can fail, among other issues that can occur [36]. Furthermore, existing work on application (re)configuration offers solutions attempting to optimise Quality of Service (QoS) metrics designed for clusters or clouds which are not highly distributed infrastructure with heterogeneous computing resources [51]. Likewise, other efforts offer solutions where the behaviour of a DSP application deployment on heterogeneous infrastructure is non-stochastic. Such works propose solutions using only linear approaches [35]. Existing solutions often design the system seeking to optimise one QoS

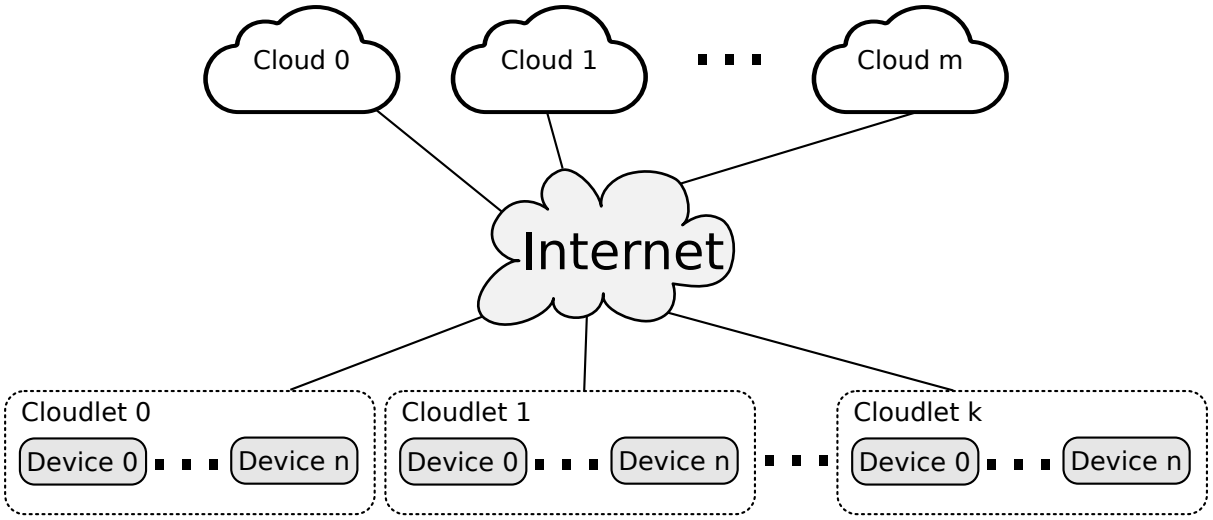

Figure 3.1: Overview of cloud-edge infrastructure.

metric, hence neglecting important user-requirements and a holistic view of the DSP deployment environment only achievable by considering several metrics [65].

This chapter addresses *how* to model an application (re)configuration on a highly distributed infrastructure by including aspects and characteristics which are frequently omitted in previous work. The model covers operator behaviours as presented in Section 2.2.2, such as *selectivity*, *operator data transformation pattern* and *operator state*. Furthermore, the model includes *splitters* and *mergers* from the *data parallelism*. Although splitters commonly perform data distribution across data streams in a round-robin fashion, our model contemplates various schemes by employing probabilities to data streams. The model also explores the impact of QoS metrics on application (re)configuration problem and proposes objective functions meeting one or multiple QoS metrics at a time.

3.2 Data Stream Processing Infrastructure and Architecture

Gedeon et al. [55] discuss how real-life infrastructure such as cellular base stations, routers, and street lamps can be used to host cloudlets [81]. The authors highlight recent investment and emerging projects including Humble Lamppostproject[1], SM!GHT[2], Edgemicro[3], LinkNYC[4], and DOT[5], which demonstrate that cloud-edge infrastructure will become even more accessible in near future.

Figure 3.1 depicts the cloud-edge infrastructure for hosting DSP systems considered in this thesis, which comprises cloud server providers and stakeholders that own or operate cloudlets. The cloud servers and cloudlets are interconnected by the Internet. Each cloud server represents a service supplied by a cloud service provider (*e.g.*, Amazon EC2, Microsoft Azure) irrespective of the number of servers that the provider uses. The internal network of a cloud provider is simplified as it generally presents very low latency, hence not having a great impact on end-

[1]https://eu-smartcities.eu/initiatives/78/description
[2]https://smight.com/en/
[3]https://www.edgemicro.com/
[4]https://www.link.nyc/
[5]https://www1.nyc.gov/html/dot/html/infrastructure/streetlights.shtml

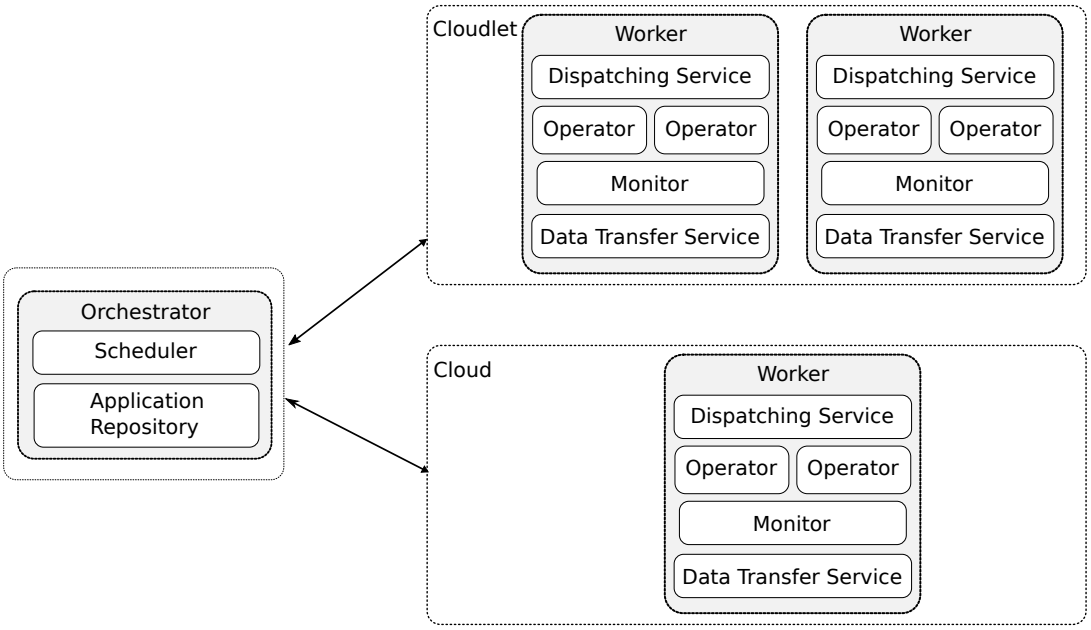

Figure 3.2: Architecture for scheduling DSP application across cloud and edge resources.

to-end application latency. In contrast, each cloudlet has multiple and heterogeneous devices connected to a LAN. The internal communication of a cloudlet often adopts technologies such as LTE and other wireless equipment that increase the latency significantly [81]. For this reason, the internal links and edge devices in a cloudlet are analysed individually. With respect to computing resource capacities, this thesis considers two constraints:

- *memory*, edge devices have limited features in this sense, and frequently DSP operators store data in memory which can be more than the device's capacity; and

- *CPU*, which impacts the volume of processed data and can also be a constraint on edge devices.

Similarly to computing resource capacities, this thesis focuses on impacting metrics of network capacities. For instance, it considers the bandwidth which can be limited and network latency, which raises the application latency significantly.

Regarding how data is generated and consumed, this thesis assumes a scenario where sensors are geographically distributed with data sources and data sinks either on the cloud or on cloudlets. We focus on system architecture capable of managing operators across cloud and edge devices considering each resource as multi-tenant. A *scheduler*, responsible for system management, leverages performance metrics on application runtime and infrastructure statistics. The envisioned architecture for deploying DSP applications depicted in Figure 3.2 is based on a master/work approach and consists of mainly two components:

- **Orchestrator**: Performs (re)configuration guided by the *scheduler* hosted in an *orchestrator* node using performance metrics collected my monitoring modules. The module also has an *application repository* service that stores information about application topology and operator code.

- **Worker**: Hosts *operators* that execute functions over the data streams, or stores data after processing. A worker also comprises a *dispatching service* in charge of managing internal connections across operators and distributing arrival data to operators; a *monitor* that obtains performance metrics of operators and compute resources; and a *data transfer service* that manages intercommunication between resources.

A user submits to the orchestrator all the operators' executable code encapsulated in containers, along with the application topology and QoS requirements which the orchestrator stores in the repository. Initially, the orchestrator profiles the application using techniques such as those offered by Kaur et al. [83]. Software Defined Network solutions [35] or discovery algorithms such as Vivaldi [44] can be used to determine and maintain the network topology information. At last, the scheduler employs data from the application, user requirements, and infrastructure to decide *how* to split the application operators dynamically and on which computing resources to place them. The scheduling happens in phases:

- **Initial operator placement** (application configuration), which refers to the initial application deployment. This deployment, which happens after the user submits the application, runs policies for establishing in which computing resource a given operator must be hosted considering application dataflow and infrastructure information.

- **Application reconfiguration**, the process of reorganising or migrating operators across available computing resources. This process follows the Monitoring, Analysis, Planning and Execution (MAPE) loop. The scheduler *monitors* the application performance metrics. The QoS requirements are *analysed* when the execution reaches a given deadline or all application paths have generated enough data to assist in establishing statistics; whichever comes last. Based on the analysis, the scheduler *plans* the operator reconfiguration. At last, the scheduler *executes* the reconfiguration plan.

We consider a single Orchestrator hosted on the cloud. The existing availability guarantees in cloud services allow this thesis to focus on solutions to application (re)configuration, whereas cloud and edge failures are left for future work.

During the initial application configuration, the operators code stored in the application repository is copied following the scheduler decisions. At reconfiguration, the operators are migrated from their current location to new resources decided by the scheduler. Each container is created to meet the CPU and memory required by an operator, and is isolated. A computing resource only hosts containers that do not exceed its memory and CPU capacity.

The aforementioned cloud and edge devices scenario is considered because it reduces the amount of data transferred at different application phases, avoiding network constraints that might exist along a path from edge to the cloud. Moreover, we are interested in evaluating the use of DSP application operators traditionally employed for cluster/cloud-based DSP solutions in more decentralised environments comprising cloud and edge resources.

3.3 Modelling a DSP System

The next section describes a Queueing Theory model covering the DSP application and infrastructure characteristics and requirements to (re)configure applications on cloud-edge infrastructure. Table 3.1 summarises the notation used throughout this chapter and the rest of this thesis.

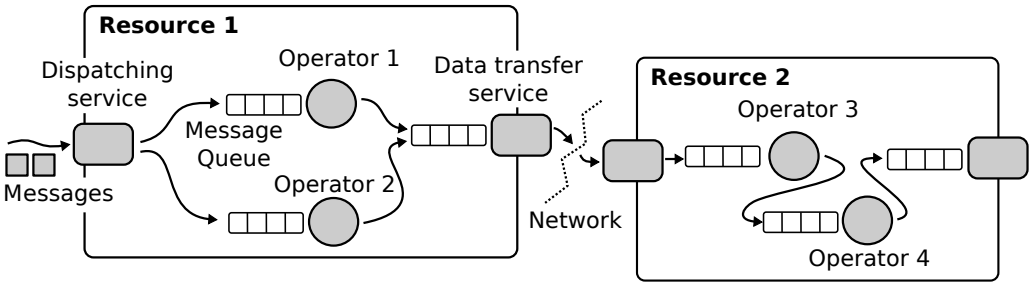

Figure 3.3: Example of four operators and their respective queues placed on two resources.

3.3.1 Infrastructure Model

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) [6] 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.

3.3.2 Application Model

A DSP application is a graph $\mathcal{G} = (\mathcal{O}, \mathcal{E})$ of *operators* $\mathcal{O}$ comprising *data sources* $\mathcal{O}^{src}$, *data sinks* $\mathcal{O}^{out}$ where data is stored or published, and *transformations* $\mathcal{O}^{trn}$ 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 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.*, operator data transformation pattern), 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 ρ 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 (λ_i^{out}) 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 depends on the number of measurements that it takes from a sensor or another monitored device. The output event rate of a data source operator $k \in \mathcal{O}^{src}$ depends on the number of measurements it takes from a sensor or another monitored device. We can then recursively compute the input and output event rates for downstream operators j as follows:

[6] The unit MIPS allows for establishing the computing resource capabilities and the required computing time for messages. In our scenario, sensors collect data periodically, and each one follows a pattern of content. The operator function is constant and message contents have minor variations. The MIPS can be applied due to messages require approximately the same number of instructions and do not have a substantial variance in the processing time. However, the proposed model can be extended to handle FLOPS or even other metrics.

Table 3.1: 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 in MIPS and bytes of resource k
$lat_{k \leftrightarrow l}$,$bdw_{k \leftrightarrow l}$	Latency and bandwidth in seconds and bps of bidirectional link $k \leftrightarrow l$
$\mathcal{O}$	Set of data source ∪ transformation ∪ data sink operators
$\mathcal{O}^{src}$, $\mathcal{O}^{trn}$, $\mathcal{O}^{out}$	Set of data sources, transformations and data sink 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
ψ_i^o	Selectivity of operator i
ω_i^o	Data compression rate of operator i
$e_{i \to j}^{\rho}$	Event stream with probability ρ 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
$\mathcal{M}$	(Re)configuration plan
M	Set of QoS metrics

$$\lambda_i^{in} = \lambda_k^{out} \qquad \forall s_{k\to i} \in \mathcal{E}, k \in \mathcal{O}^{src} \tag{3.1}$$

$$\lambda_j^{in} = \sum_{s_{i\to j}\in\mathcal{E}} \lambda_i^{in} \times \psi_{o_i} \times \rho_{s_{i\to j}} \qquad \forall i \in \mathcal{O}, i \notin \mathcal{O}^{src} \tag{3.2}$$

$$\lambda_j^{out} = \lambda_j^{in} \times \psi_{o_j} \qquad \forall j \in \mathcal{O}, j \notin \mathcal{O}^{out} \tag{3.3}$$

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 operator data transformation pattern (*i.e.*, ω_i^o). In other words:

$$\varsigma_i^{in} = \varsigma_k^{out} \qquad \forall s_{k\to i} \in \mathcal{E}, k \in \mathcal{O}^{src} \tag{3.4}$$

$$\varsigma_j^{in} = \varsigma_i^{in} \times \omega_{o_i} \qquad \forall i, j \in \mathcal{O}, \forall s_{i\to j} \in \mathcal{E}, i, j \notin \mathcal{O}^{src} \tag{3.5}$$

$$\varsigma_j^{out} = \varsigma_j^{in} \times \omega_{o_j} \qquad \forall j \in \mathcal{O}, j \notin \mathcal{O}^{out} \tag{3.6}$$

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 3.3. The queueing system model for communication and processing services was chosen due to its wide use in DSPEs. As presented in Chapter 2, frameworks often run operators in containers, and the communication between them is via brokers. After an operator processes a message, the message pushed to the queue(s) of the downstream operator(s). The Dispatching Service and Data Transfer Service mimics the Stream Manager in Apache Heron or the Supervisor in Apache Storm, which controls the execution of containers. The local manager was decoupled in our case because of the geo-distributed nature of cloud-edge infrastructure. The Dispatching Service manages the communication between operators locally while the Data Transfer Service handles the external ones. Operators in the same computing resource write directly to their downstream operator queues while in external communications message brokers bring guarantees for the message delivery.

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. This guarantees the time order of events; an important requirement in many data stream processing applications. Both operators and the communication service follow a widely employed M/M/1 model [22, 36, 59] for their queues which allows for estimating the waiting and service times for computation and communication. The used queue model is one of the most widely researched models in the classic literature and it is capable of capturing randomness in arrival and service times. Arrivals occur according to a time-homogeneous Poisson process with a constant rate – event bursts are unconsidered because we assume that sensors are wired and collect data periodically –, and the service rate has an exponential distribution with a constant mean time –we assume a homogeneous processing time for computing each message. 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:

$$stime_{\langle i,k\rangle} = \frac{1}{\mu_{\langle i,k\rangle} - \lambda_i^{in}} + \frac{ws_i^o}{\lambda_i^{in}} \tag{3.7}$$

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:

$$ctime_{\langle i,k\rangle\langle j,l\rangle} = \frac{1}{\left(\frac{bdw_{k\leftrightarrow l}}{\varsigma_i^{out}}\right) - \lambda_j^{in}} + lat_{k\leftrightarrow l} \tag{3.8}$$

3.3.3 Infrastructure and Application Constraints

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

$$\lambda_i^{in} < \mu_{\langle i,k\rangle} \qquad \forall i \in \mathcal{O}, \forall k \in \mathcal{R} | \mathbb{1}_{\langle i,k\rangle} = 1 \tag{3.9}$$

$$\lambda_i^{out} < \left(\frac{bdw_{k\leftrightarrow l}}{\varsigma_i^{out}}\right) \qquad \forall i \in \mathcal{O}, \forall k \leftrightarrow l \in \mathcal{L} | \mathbb{1}_{\langle i,k\rangle} = 1 \tag{3.10}$$

The CPU and memory requirements of operators on each host are ensured by constraints 3.11 and 3.12:

$$\sum_{i\in\mathcal{O}} \mathbb{1}_{\langle i,k\rangle} \times \lambda_i^{in} \leq cpu_k^r \qquad \forall k \in \mathcal{R} \tag{3.11}$$

$$\sum_{i\in\mathcal{O}} \mathbb{1}_{\langle i,k\rangle} \times (mem_i^o + ws_i^o \times \varsigma_i^{in}) \leq mem_k^r \qquad \forall k \in \mathcal{R} \tag{3.12}$$

Data requirements of streams placed on links are guaranteed by:

$$\sum_{\substack{s_{i\rightarrow j}\in\mathcal{E} \\ k\leftrightarrow l\in\mathcal{L}}} \mathbb{1}_{\langle i\rightarrow j, k\leftrightarrow l\rangle} \times \varsigma_i^{out} \leq bwd_{k\leftrightarrow l} \qquad \forall k \leftrightarrow l \in \mathcal{L} \tag{3.13}$$

Constraints 3.14 and 3.15 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:

$$\sum_{k\in\mathcal{R}} \mathbb{1}_{\langle i,k\rangle} = 1 \qquad \forall i \in \mathcal{O} \tag{3.14}$$

$$\sum_{k\leftrightarrow l\in\mathcal{L}} \mathbb{1}_{\langle i\rightarrow j, k\leftrightarrow l\rangle} = 1 \qquad \forall s_{i\rightarrow j} \in \mathcal{E} \tag{3.15}$$

3.3.4 Quality of Service Metrics

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

Aggregate End-to-End Application 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:

$$p_i = o_0, o_1, \ldots, o_k, o_{k+1}, \ldots, o_{n-1}, o_n \tag{3.16}$$

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:

$$\begin{aligned} L_{p_i} = & \sum_{j \in p_i, k \in \mathcal{R}} \mathbb{1}_{\langle j,k \rangle} \times stime_{\langle j,k \rangle} \\ & + \sum_{l \in \mathcal{R}} \mathbb{1}_{\langle j \to j+1, k \leftrightarrow l \rangle} \times ctime_{\langle j,k \rangle \langle j+1,l \rangle} \end{aligned} \tag{3.17}$$

The aggregate end-to-end latency (*i.e.*, end-to-end latency) AL is therefore given by:

$$AL = \sum_{p_i \in \mathcal{P}} L_{p_i} \tag{3.18}$$

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 $\mathcal{L}_w$:

$$W = \sum_{\substack{s_{i \to j} \in \mathcal{E} \\ k \leftrightarrow l \in \mathcal{L}_w}} \mathbb{1}_{\langle i \to j, k \leftrightarrow l \rangle} \times \varsigma_i^{out} \tag{3.19}$$

Monetary Cost of Communication

The monetary cost of event exchange is based on the main elements[7] of Internet of Things (IoT) services of two major Cloud-edge players, namely Amazon IoT Core[8] and Microsoft Azure IoT Hub[9]. 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:

$$\begin{aligned} C^m = \sum_{\substack{i \in \mathcal{O}, j \in \mathcal{O} \\ k \in \mathcal{R}_e, l \in \mathcal{R}_c}} & \big(\mathbb{1}_{\langle i \to j, k \leftrightarrow l \rangle} \times \mathbb{1}_{\langle j,l \rangle} \times \mathbb{1}_{\langle i,k \rangle} \times \lambda_j^{in} + \\ & \mathbb{1}_{\langle j \to i, k \leftrightarrow l \rangle} \times \mathbb{1}_{\langle i,k \rangle} \times \mathbb{1}_{\langle j,l \rangle} \times \lambda_j^{out} \big) \end{aligned} \tag{3.20}$$

[7]For simplicity, we consider mainly two costs in IoT Hubs, namely connections and events.
[8]AWS IoT Core - https://aws.amazon.com/iot-core/pricing/
[9]Microsoft Azure IoT Hub - https://azure.microsoft.com/en-us/pricing/details/iot-hub/

The number of connections between edge and cloud is:

$$C^c = \sum_{\substack{i\in\mathcal{O},j\in\mathcal{O}\\ k\in\mathcal{R}_e,l\in\mathcal{R}_c}} \big(\mathbb{1}_{\langle i\to j,k\leftrightarrow l\rangle} \times \mathbb{1}_{\langle j,l\rangle} \times \mathbb{1}_{\langle i,k\rangle} + \mathbb{1}_{\langle j\to i,k\leftrightarrow l\rangle} \times \mathbb{1}_{\langle i,k\rangle} \times \mathbb{1}_{\langle j,l\rangle}\big) \tag{3.21}$$

Hence the total cost is given by:

$$C = C^c \times price\_connections + C^m \times price\_events \tag{3.22}$$

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

Reconfiguration Overhead

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 configuration and conveniently reassign operators to available resources. Similarly to solving the DSP application configuration problem, addressing reconfiguration consists of accommodating the application operators onto the available resources in order to optimise one or multiple QoS metrics. 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:

$$D = \max_{\substack{i\in\mathcal{O}\\ k\in\mathcal{R}}} [\mathbb{1}_{\langle i,k\rangle} \times (T^i_{code} + T^i_{state})] \tag{3.23}$$

3.3.5 Single-Objective versus Multi-Objective (Re)configuration

As DSP applications are generally latency-sensitive, this thesis initially provides solutions that minimise the *aggregate end-to-end application latency* or end-to-end application latency. When optimising for a single metric, the problem of placing a distributed DSP application consists of finding a mapping that minimises the aggregate end-to-end application latency of all application paths and that respects the resource and network constraints. In other words, find the mapping that minimises AL:

$$\min AL \tag{3.24}$$

When considering multiple metrics, the goal is to find a plan $\mathcal{M} : \mathcal{O} \rightarrow \mathcal{R}$, $\mathcal{E} \rightarrow \mathcal{L}$ where operators are (re)reconfigured 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 [148] 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:

$$agg\_cost = \sum_{z \in W} w_z \times m_z, \tag{3.25}$$

The agg_cost is flexible as it allows a user to specify the importance of each QoS metric by assigning weights. The multi-objective approach also allows for adding new QoS metrics without complex changes in the model.

3.4 Conclusion

In this chapter, we modelled DSP applications and infrastructure topology by employing Queueing Theory. The chapter presented the target DSP architecture, which comprises a cloud-edge infrastructure managed and controlled by a cloud-hosted orchestrator. The orchestrator takes decisions based on user-defined and performance metrics. The metrics are applied to a model that accounts for application behaviours such as selectivity, operator data transformation pattern, and operator state. The model also considers splitters and mergers from the data parallelism that are often neglected in previous work.

According to previous work on configuring DSP applications, our model address certain weaknesses in IoT scenarios. For instance, it is structured to work with architectures designed for highly distributed infrastructure and can handle either single or multiple QoS metrics. The model's features led us to formulate and investigate the application (re)configuration in two phases. In the first phase, we investigate the application configuration by considering a static IoT scenario where the user and system yield information to decision-making. The second phase consists of incorporating the dynamicity of the DSP application life-cycle where the application operators must be reorganised on the available resources to maintain the application performance. In the next chapter, we examine the first phase, where we introduce strategies to perform the DSP application configuration dynamically, looking at optimising a single and simultaneous QoS metrics.

Chapter 4

Strategies for Data Stream Processing Placement

Contents

4.1 Introduction

The deployment of Data Stream Processing (DSP) applications onto heterogeneous infrastructure has proved to be NP-hard [27]. Moreover, moving operators from cloud to edge devices is challenging due to limited capabilities of edge devices [20]. Existing work proposes architecture that places certain stream processing elements on micro data centers located closer to where the data is generated [32] or employs mobile devices for DSP [49, 99]. To simplify the placement problem, communication is often neglected [42], but it could be an important issue in highly distributed infrastructure [81]. Likewise, the operator behaviour and requirements are oversimplified using static splitting decisions, such as those proposed by Sajjad *et al.* [119].

This chapter introduces a set of strategies to place operators onto cloud and edge infrastructure while considering characteristics of resources and meeting application requirements. We consider analytics applications with multiple geographically distributed sources and sinks. In particular, we decompose the application graph by identifying behaviours such as *splitters* and *mergers* and then dynamically create a candidate placement for the operators considering resources from a cloud-edge infrastructure. Based on the candidate placement, we propose strategies to address the application configuration problem. The problem is addressed as two distinct optimisation schemes:

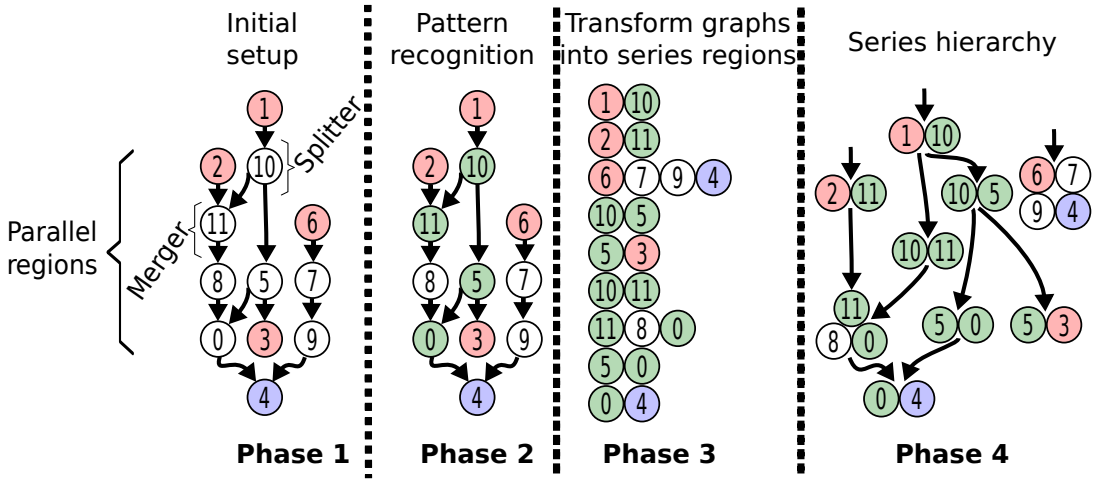

Figure 4.1: Method for finding the dataflow split points, where red means placed on edge, blue represents placed on cloud, and green delimits splitters and mergers.

- Single-objective: The proposed solutions account only for the end-to-end application latency. We execute comprehensive simulations covering multiple application settings, and our approach demonstrates an improvement of up to 50% compared to state-of-the art strategies.
- Multi-objective: We use one of the strategies proposed for single-objective and expand it to cover WAN traffic and the monetary cost of communication. We implemented it in a real-life framework. Results show a reduction of over 38% in the end-to-end application latency. Meanwhile, the approach reduces the data transfer by at least 38%, and saves up to 50% in messaging costs.

4.2 Strategies Considering End-to-End Application Latency

This section explains how patterns in the DSP application graphs are identified and then introduces strategies that employ these patterns to devise placement decisions. The strategies focus on identifying operator placements that minimise the end-to-end application latency.

4.2.1 Finding Application Patterns

As depicted in Figure 4.1, a dataflow can comprise multiple patterns such as (i) splitters, 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 [103]; (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) mergers, which merge the outcome of parallel regions.

The strategies consider Series-Parallel-Decomposable Graphs (SPDG) and related techniques to identify graph regions that present these patterns [48]. The operator patterns refer to the final destination of the messages, either cloudlets or to the cloud. This information is used to build a hierarchy of downstream and upstream relations between regions and assist on placing operators across cloud and edge resources. The streams in the graph paths that separate the

operators are hereafter called the *split points*. Figure 4.1 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:

1. the method starts with sources and sinks whose placements are predefined by the user;
2. split points are discovered (green circles) as well as sinks that correspond to actuators that can be placed on the edge;
3. the branches between the existing patterns (green, red and blue circles) are transformed into series regions; and
4. a hierarchy following the dependencies between regions is created.

Algorithm 1 describes the function *GetRegions* used to identify the patterns and obtain the series regions. Operators are grouped in series regions according to their stream connections and message destination. The series regions are used in the operator placement decisions. First, the function adds two virtual vertices to the graph, one named *virt_src* connected to all data sources and another named *virt_sink* to which all sinks are connected (line 2-4). The virtual vertices allow for recognising all paths between sources and sinks. Breadth-first search traversal algorithm [107] is used to discover the application paths. 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). At last, 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).

4.2.2 Operator Placement Strategies

Data must be handled in short delays to meet the stringent requirements of DSP. For this reason, we propose strategies which rely on the *end-to-end application latency* metric. The proposed solutions exploit the resulted hierarchy set from *GetRegions* to sort operators according to their upstream operators, *i.e.*, the operator 11 shown in Figure 4.1 must be placed after operator 1, 2 and 10. This approach establishes an operator ordering (*i.e.*, deployment sequence), which allows for applying our model linearly. For instance, the model requires the output data rate from the previous operators to estimate the requirements of a given operator, and through the deployment sequence it is possible to estimate these requirements sequentially, which allows us to propose two strategies:

- **Response Time Rate (RTR)**, which estimates the *end-to-end application latency* of all available computing resources; and
- **Response Time Rate with Region Patterns (RTR+RP)**, which uses the hierarchy to split the application graph across edge and cloud, optimising only the response time on the edge.

As presented earlier, the end-to-end application latency of an operator in a path comprises the time taken to transfer data and the time to compute an event. As an operator can be in multiple paths, the response time rate corresponds to the total time taken to transfer data from multiple paths rather than evaluating each path individually.

Algorithm 1: Algorithm to detect splitters and mergers.

```
1  Function GetRegions(G = (O, S), O^src, O^out)
2      O ← O ∪ virt_src ∪ virt_sink
3      S ← S ∪ s_{virt_src→o}, ∀o ∈ O^src
4      S ← S ∪ s_{o→virt_sink}, ∀o ∈ O^out
5      for p ∈ GetAllPaths(G, virt_src, virt_sink) do
6          for o ∈ p do
7              temp ← temp ∪ {o}, ∀o ∉ {virt_src, virt_sink}
8              ups ← |⟨∗, o⟩ ⊂ S|, downs ← |⟨o, ∗⟩ ⊂ S|
9              if ups > 1 or downs > 1 and o ∉ {virt_src, virt_sink} then
10                 regions ← regions ∪ temp, temp ← {o}
11     Delete duplicate regions
12     for src_series ∈ regions do
13         for dst_series ∈ regions do
14             if src_series ≠ dst_series then
15                 if src_series[|src_series| − 1] = dst_series[0] then
16                     hierarchy ← hierarchy ∪ {src_series, dst_series}
17     return hierarchy
```

Response Time Rate (RTR)

RTR is a greedy strategy that places operators incrementally by evaluating the end-to-end application latency of paths while respecting the resource constraints presented in Section 3.3.5.

RTR calculates the response time for each operator by considering the previous mappings, resource capabilities, and operator requirements. The approach initially organises the deployment sequence by employing a breadth-first search traversal algorithm [107] to give priority to upstream operators. Each operator of the deployment sequence has its response time estimated for non-constrained computational resources (Algorithm 2). After that, the resources are sorted in ascending manner by their response times. The host with the shortest response time is picked, and the host's residual capabilities are updated.

Response Time Rate with Region Patterns (RTR+RP)

RTR+RP is a strategy that handles complex dataflows that contain multiple paths from sources to sinks. It explores the operator patterns (split points) and the sink placement (cloud or edge) respecting the environment constraints (Section 3.3.3). Based on the region hierarchy (Figure 4.2), the operators are classified and allocated. Operator 5, for instance, was reallocated since the edge does not respect the resource constraints. RTR+RP aims to allocate operators across edge and cloud meeting the response time rate only for operators located in the edge, in contrast to the RTR strategy that evaluates the response time rate for all operators.

RTR+RP defines the deployment sequence similar to RTR, but it builds upon the classification of operators considering the served sink infrastructure (candidate infrastructure).

The classification is:

- *cloud-only* when an operator only serves sinks placed on the cloud; and

Algorithm 2: Calculating the computational response times.

Function *EstimateResponseTimes*$(\mathcal{N} = (\mathcal{R}, \mathcal{L}), \mathcal{G} = (\mathcal{O}, \mathcal{S}), o)$
  for $child \in \langle o, * \rangle \subset \mathcal{S}$ **do**
    $upstreams \leftarrow \langle child, r \rangle, \forall r \in \mathcal{R}$ ***and*** $mo_{\langle child,r \rangle} = 1$
  for $r \in \mathcal{R}$ **do**
    $comm \leftarrow 0$
    for $mapping \in upstreams$ **do**
      if `GetHost`$(mapping) \neq r$ **then**
        $com \leftarrow comm + ctime_{\langle mapping \rangle \langle o,r \rangle}$
    if `MeetConstraints` **then**
      $rt \leftarrow rt \cup \langle r, stime_{\langle o,r \rangle} + comm \rangle$
  return rt

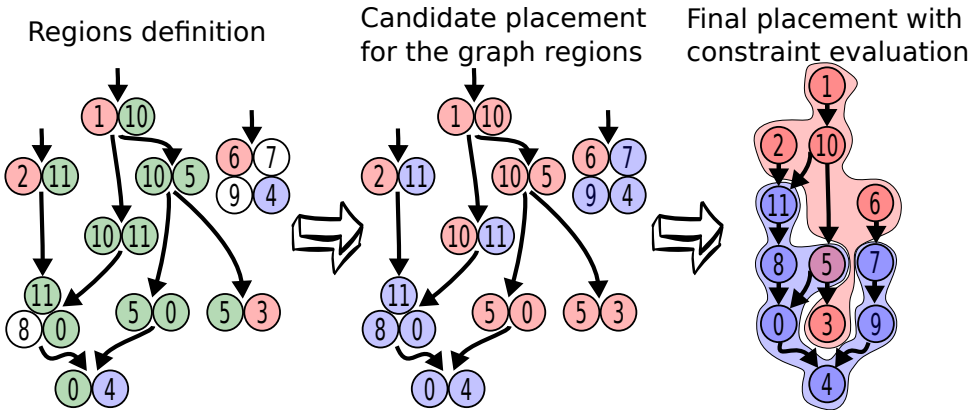

Figure 4.2: Blue circles are operator candidates to be deployed on cloud whereas red circles are candidates for edge. The right-hand graph shows the final deployment.

- *edge* when an operator shares paths with sinks located at the edge.

Each operator on the deployment sequence has its candidate infrastructure evaluated. Edge candidates have their response time estimated for non-constrained edge devices where the device with the shortest response time is picked. In contrast, cloud candidates do not have their response time estimated. Hence, the cloud hosts its operator candidates and those that do not meet the constraints on the edge. At last, after the operator mapping, the resources have their residual capabilities updated.

4.2.3 Experimental Setup and Performance Evaluation

This section first describes the experimental setup, performance metrics and then discusses experimental results on how the strategies impact the end-to-end application latency.

Experimental Setup

We built a framework atop OMNET++[1] discrete event simulator to model and simulate distributed DSP applications. A computational resource is an entity with CPU, memory and bandwidth capabilities whereas operators comprise waiting queues and transformation operations that pose demands in terms of CPU, memory and bandwidth.

We model our computing resources as:

- Raspberry PI's 2 (RPi) (*i.e.*, 4,74 Millions of Instructions per Second (MIPS)[2] at 1 GHz and 1 GB of RAM), which are considered to edge devices; and
- AMD RYZEN 7 1800x (*i.e.*, 304,51 MIPS[3] at 3.6 GHz and 1 TB of memory), which are considered as cloud servers.

The infrastructure comprises two cloudlets with edge computing devices (*Cloudlet 1* and *Cloudlet 2*) and a *Cloud*. Each cloudlet has 20 RPi's, whereas the cloud consists of 2 servers. A gateway interfaces each cloudlet's LAN and the Internet [68]. The LAN has a latency drawn from a uniform distribution between 0.015 and 0.8 ms and a bandwidth of 100 Mbps. The WAN has latency drawn uniformly between 65 and 85 ms, and bandwidth of 1 Gbps. These values reflect measurements carried out in previous work considering this type of environment [81].

As DSP applications exist in multiple domains with diverse topologies such as face recognition, speech recognition, weather sensing. Sensors and actuators ingest a variety of events in the system. We aim to capture this diversity by modelling and simulating two scenarios namely microbenchmarks and complex applications with various application workloads.

Microbenchmarks: As in previous work [81], we first perform a controlled evaluation using 10 bytes, 50 KB, and 200 KB message sizes which corresponds to text, pictures/objects, and voice records data types. Each application, depicted in Figure 4.3, has three input event rates as presented in Table 4.2, a set of CPU requirements according to the message sizes as presented in Table 4.1 and a configuration of splitter/merger operators to explore the path sizes.

The operators selectivity and operator transformation pattern rates[4] use 100, 75, 50 and 25% as parameters. Sources ingest messages from sensors while sinks act as actuators on cloudlets and databases/message brokers on cloud.

[1] OMNET++ http://www.omnetpp.org/.

[2] https://hackaday.com/2015/02/05/benchmarking-the-raspberry-pi-2/

[3] https://reddit.com/r/BOINC/comments/5xog5v/boinc_performance_on_amd_ryzen

[4] If the operator transformation pattern rate is equal to 100%, it means a stable system. Otherwise, it refers

Table 4.1: CPU requirements following message sizes.

Message size	CPU requirement in Instructions per Second (IPS)
10 bytes	3.7952
50 KB	18,976
200 KB	75,904

Table 4.2: Input event rate.

App.	10 bytes	50 KB	200 KB
App1	124999, 624999, 1249999	24, 124, 249	6, 31, 62
App2	124999, 374999, 624999	24, 74, 124	6, 19, 31
App3	124999, 218749, 300000	24, 43, 62	6, 10, 15
App4	124999, 137499, 150000	24, 27, 30	6, 7

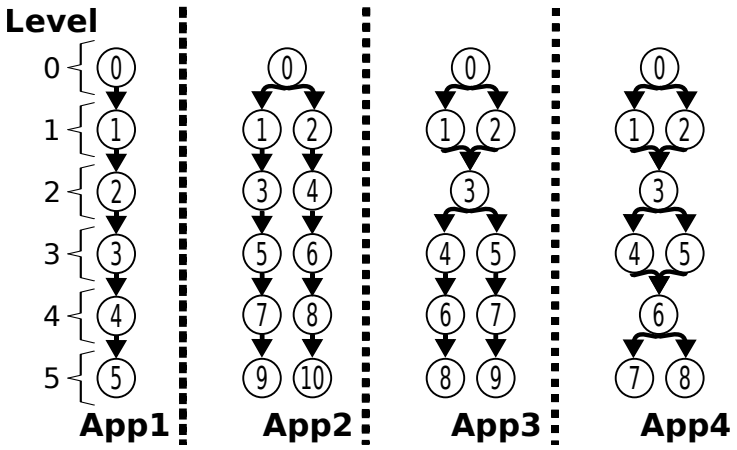

Figure 4.3: Six-hop applications.

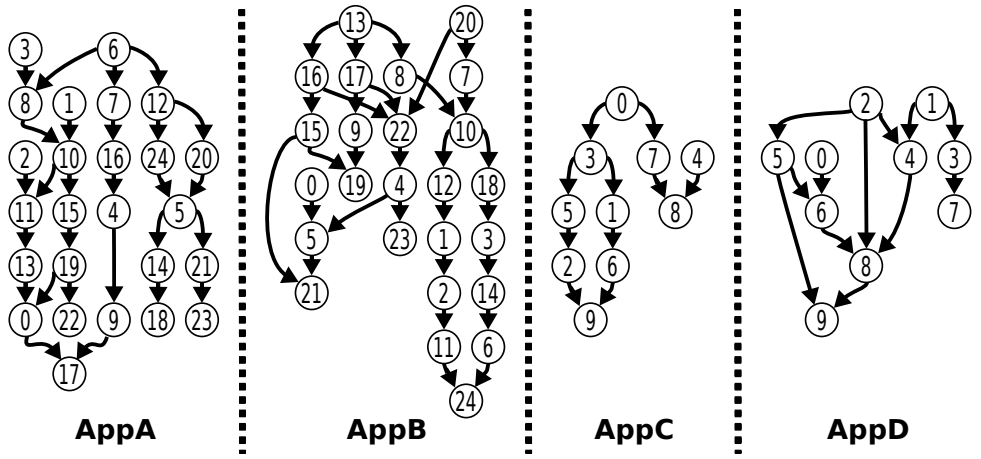

Figure 4.4: Complex applications.

Complex Applications: This scenario presents multiple operator behaviours and larger numbers of operators. We crafted the application graphs presented in Figure 4.4 using a Python library[5] and varying the parameters of the operators using a uniform distribution with the ranges presented in Table 4.3. The cloudlets host the sink and source placements, except for the sink on the critical path, which will be hosted on the cloud. This is the typical behaviour of Internet of Things (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 at the cloud. We generated 1160 graphs randomly applying multiple selectivities, operator transformation pattern rates, sink and source locations, input event sizes and rates, memory, and CPU requirements. Inspired on the size and operator behaviours of RIoTBench [129] applications, a Realtime IoT Benchmark suite, we created two sets of applications, namely:

- *large* (AppA and AppB) containing 25 operators; and
- *small* (AppC and AppD) holding 10 operators.

Table 4.3: Operator attributes.

Parameter	Value
cpu (MIPS)	1-100
Operator transformation pattern rate	10%-100%
mem (bytes)	100-7500
Input event size	100-2500
Selectivity	10%-100%
Input event rate	1000-10000

Metrics: The main performance metric is the *aggregate end-to-end application latency*, which is the time events are generated to the time they are processed by the sinks. To demonstrate the gains obtained by our approach, we compared the proposed strategies against:

to a compression of 25%, 50% and 75% when considering 75, 50 and 25% rates, respectively.

[5]https://gist.github.com/bwbaugh/4602818

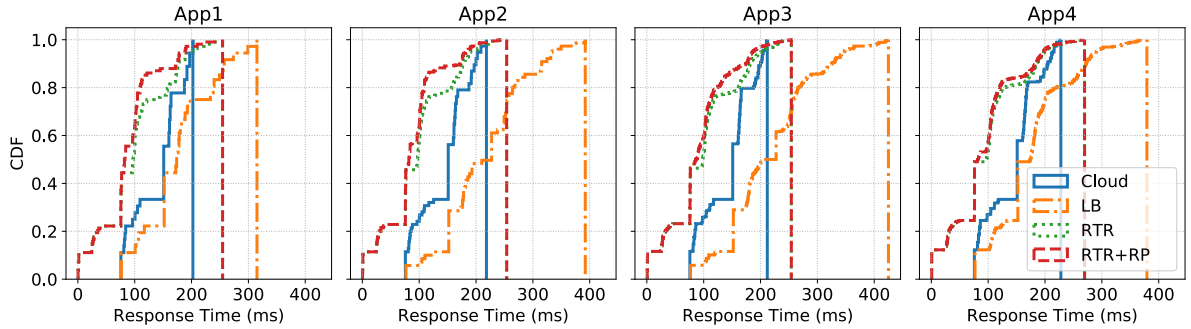

Figure 4.5: CDF of end-to-end application latency for microbenchmarks.

- *Cloud*, a traditional approach which deploys all operators in the cloud, apart from operators provided in the initial placement; and
- *LB* [134] from the state-of-the-art, which iterates a vector containing the application operators, gets the middle host of the computational vector and evaluates CPU, memory, and bandwidth constraints to obtain the application configuration.

Evaluation of End-to-End Application Latency

Figure 4.5 summarises the response times for all microbenchmarks. For App1 we carried out 432 experiments considering 4 selectivities, 4 operator transformation patterns, 3 input event rates, 3 sink locations and 3 input event sizes. Each experiment ran for 300 seconds in simulation time. RTR and RTR+RP have shown to be over 95% more efficient than Cloud approach and LB. Initially, LB had its performance comparable to Cloud, but LB lost performance afterward due to its specific modeling (*i.e.*, health care, and latency-critical gaming) and method (computational ordering).

Cloud achieved 5% better results (when the blue line crosses the red line at ≈200 ms) when handling voice records (200 KB), selectivity, and data compression rate equal to 1 (without reducing the size of the messages and discarding events) and when the sink was placed on Cloudlet 2 and the source was located on Cloudlet 1 (traverse WAN). For the scenario mentioned before, the operators were CPU-intensive where Cloudlet 1 or 2 can host only one operator per edge device at a time, which increases the communication costs. Moreover, RTR+RP outperformed RTR for sinks placed on cloud, mainly without message discarding and no reduction on message sizes. Even further, to investigate the impacts generated by the split points, we launched App2, App3, and App4 and observed a gradual performance loss (decreasing on the distance between green and red line - ≈100 ms) according to the position between the split points and sinks, and the location of sinks. When sinks and sources require events to traverse the WAN and there is a low number of hops between the split point and sink, the proposed strategies cannot define a reasonable dataflow split because of the assumption to prioritise the sinks on the edge.

The complex application scenario investigates the outcomes for generic and multiple path applications using various dataflow configurations. We launched each experiment during 60 seconds of simulation time, and the sources and sinks were distributed uniformly and randomly across the infrastructure, except for operator 17 on AppA, operator 24 on AppB, operator 9 on AppC, and operator 9 on AppD placed in cloud due to the critical path. Figure 4.6 shows the CDF of response times. Even under large applications RTR+RP was able to reduce the

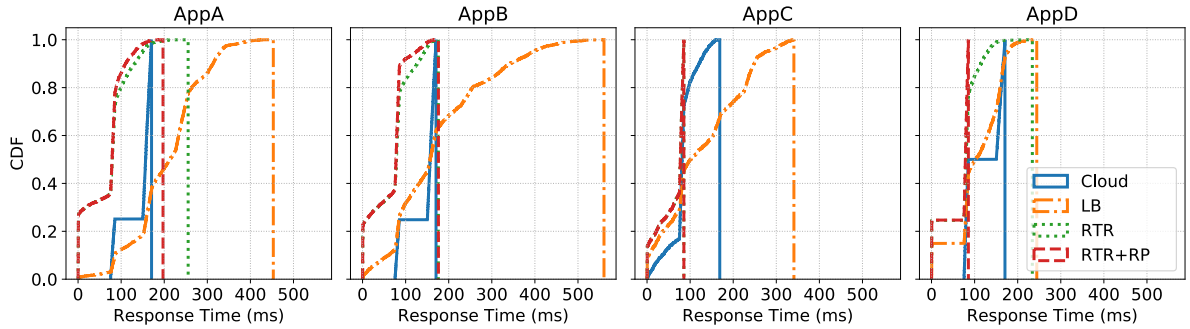

Figure 4.6: CDF of response times for complex applications.

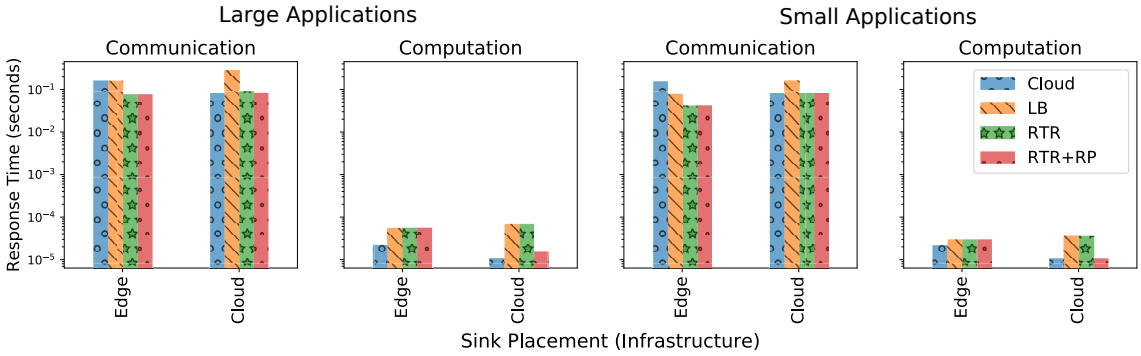

Figure 4.7: Communication and computation time for sinks placed on cloud and cloudlets.

response time by applying the region pattern identifications and recursively discovering the operator dependencies with a given sink placement onto dataflows with various paths.

Our strategies outperformed Cloud in over 6% and 50% under small and large applications, respectively. Cloud poses high communication overhead when the sink is located on cloudlets due to messages having to traverse the Internet. Similarly, we improve the response times in over 23% (small) and 57% (large applications) compared to the LB approach. This occurs because LB does not estimate the communication overhead and assumes a shorter response time on cloudlets.

Figure 4.7 shows the communication latency which comprehends the total time to transfer a message between the resources, and the computation that corresponds to the total time to compute all operators. The communication cost for sinks placed on cloudlets at Cloud approach was about 160 ms, and RTR+RP was 76 ms. Our solution outperformed Cloud in up to 52% by putting operators closer to cloudlet sinks, but sinks on the cloud. RTR+RP had a slight performance loss of 3%. Hence, our approach is effective in reducing the communication cost, and, by doing so, it compensates the edge limitations and reaches good results in minimising the total response time.

RTR outperforms LB and Cloud because when planning the operator placement, it sequentially estimates the operator response times applying our proposed model (Chapter 3) while Cloud and LB analyse only the resource capabilities when establishing the operator placement. In contrast, RTR+RP focuses on IoT scenarios where there exist feedback-loop relative to actuators. Actuators require short delays for message processing. RTR+RP splits the application operators by regions according to the infrastructure destination of the messages, and then gives

priority to cloudlet destinations when estimating the response time because actuators are generally placed on cloudlets. Doing so, RTR+RP improves the edge devices utilisation by placing operators that send messages to cloulets and avoiding to waste their capabilities feeding data sinks placed on the cloud (*i.e.*, do not require low response time). As the application dataflows contain several actuators, much of the operators are placed on cloudlets closer to data sources or sinks neglecting the high communication overhead imposed by Internet links.

The proposed strategies allow for reducing the end-to-end application latency, but in many scenarios there are other QoS metrics or user-defined requirements that impact the performance. In addition, although a simulated environment provides a controlled scenario, it cannot capture all existing complexities of a distributed DSP architecture. For these reasons, in the next section, we explore the implementation of our approach in a real-life Data Stream Processing Engine (DSPE) where we expand RTR+RP strategy by including new QoS metrics.

4.3 Strategy Using Multiple Quality of Service Metrics

This section describes a case study that requires a holistic view of the DSP environment where multiple Quality of Service (QoS) metrics must be considered. We consider a geo-distributed DSP system that requires a stack of services that must be scalable. The services must exchange data among themselves to guarantee availability. However, this communication often occurs through the Internet, which poses challenges regarding delays and the cost of exchanging data among elements of the service stack. To address the issues above, we extend the RTR+RP strategy to take into account the WAN traffic and the monetary cost of communication. We select it because it achieved the best results when considering the end-to-end application latency. The strategy was implemented in R-Pulsar, a real-life DSP framework allowing the scheduler to compute operator placements covering multiple QoS metrics presented in Section 3.3.5.

4.3.1 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 [31]. OODA is a decision-making cycle to process data streaming from sensors in real time, becoming an essential design characteristic for DSP applications.

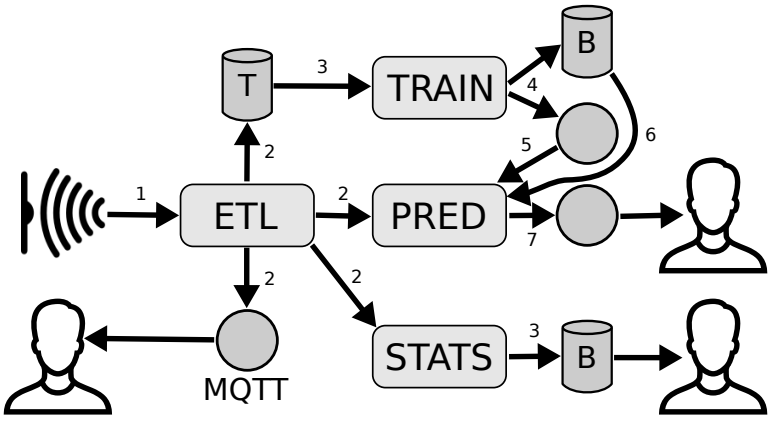

Figure 4.8: RIoTBench IoT high-level logical interactions among sensors, applications and users.

Anshu *et al.* [129] offer a suite of DSP applications that follows the closed-loop OODA cycle. The applications are based on common IoT patterns for data pre-processing, statistical summarisation, and predictive analytics. These are coupled with workloads sourced from real IoT observations. A high-level overview of the logical interaction of the DSP applications is depicted in Figure 4.8.

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.

Statistical Summarisation (STATS) performs higher order aggregation and plotting operations, and stores the generated plots into the cloud, from where webpages can load the visualisation 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.

4.3.2 The DSPE and a Multi-Objective Strategy

This section presents the R-Pulsar framework, its components, and how a multi-objective strategy was implemented on it.

R-Pulsar Framework

R-Pulsar is a lightweight data analytics software stack for collecting, processing, and analysing data at the edge and/or at the cloud. R-Pulsar has been extended to provide developers with the ability to decide how to split the application operators across edge and the cloud resources, 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 [114]. 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.

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 Network (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. The services of this layer are used by *Scheduler* presented in Chapter 3 for providing insights and then helping in the operator placement.

- **R-Pulsar Plan Finder:** This layer computes an optimised operator placement plan. It uses a three-step approach for calculating the sub-optimal operator placement plan for deploying dataflows between the edge and the cloud. We describe later 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, operator assignment, and operator reassignment in case of failure. These services are found in both Workers and Orchestrator (see Chapter 3). Each Worker is responsible for executing and monitoring its operators. Data gathered from the operator executions is provided to the Orchestrator, which assists in determining the operator reassignments and defining whether the system is running properly.

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/worker clusters such as Apache Storm [136] in the sense that R-Pulsar master and worker node roles are assigned dynamically every time a dataflow is deployed.

The master RP's primary responsibility is to manage, coordinate, and monitor a dataflow running on the R-Pulsar P2P network, including dataflow deployment, operator assignment, and operator 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. The master RP receives the role of Orchestrator presented in Chapter 3. R-Pulsar [113] creates a location-aware overlay network where computing nodes are organised in quadtree following their latency for avoiding the constant update of the routing tables. Each quadtree is a tree data structure in which each internal node has at maximum four children and represents a 2D-bounded box covering a part of the space to index, using a root node to cover the entire area. A new RP is added to the system by determining which quadrant the RP point occupies, and inserting it to the quadtree from the root node to the appropriate leaf node. Every time the quadtree splits, the system creates four new P2P rings. The master RP is in charge of manning the quadtree structure, and dictates when to split P2P structure. Any time the overlay network is splitted, the master RP randomly elects one of the RP nodes of the subdivision to be the master node of that region.

Deploying a topology to the R-Pulsar P2P network involves submitting the pre-packaged dataflow file along with topology configuration, which will be stored in the Application Repository in the Orchestrator. Then the information will be routed to the responsible RP using the content-based interactions [114]. 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 application operators stored in the Application Repository to the worker RPs. Each worker node is responsible for creating, starting, and stopping worker operators assigned to that node. Worker RPs are also responsible for once the master RP has died to perform a master RP election.

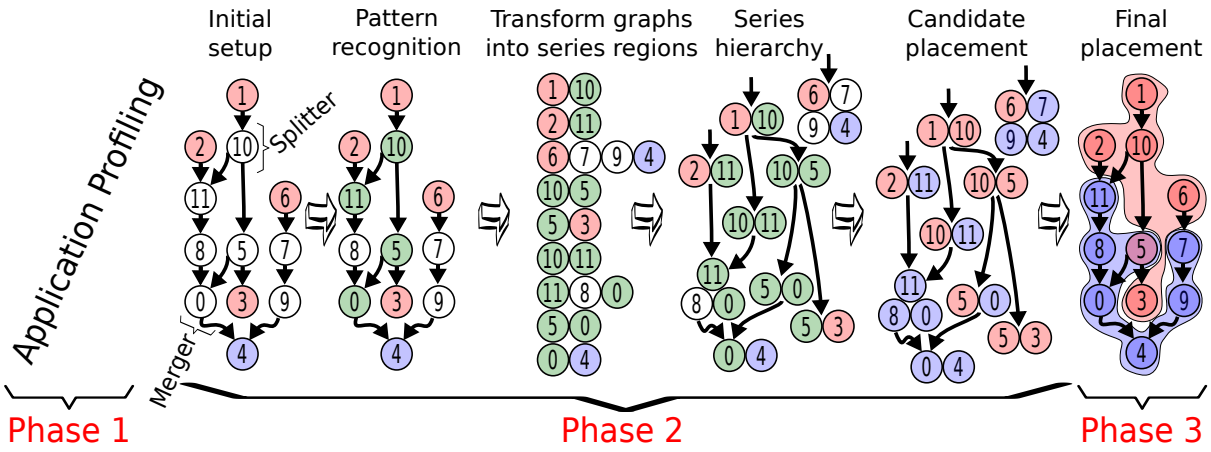

Figure 4.9: Phases to determine the final placement using split points, where red means placed on edge, blue represents placed on cloud, and green delimits splitters and mergers.

The master RP also tracks the status of all worker nodes and the operators 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 operators 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, operator 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 operators remain healthy.

R-Pulsar does not have a scheduler to establish operator placement dynamically on edge-cloud infrastructure. Then, the next section describes a RTR+RP version, which considers multiple QoS metrics and also introduces how the interaction between the framework and the configuration strategy happens.

Response Time Rate with Region Patterns with Multiple QoS Metrics

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 minimise the *agg_cost* (Section 3.3.5) by splitting the IoT application across edge and cloud by considering priorities of operators according to the infrastructure to which the sinks are assigned. As depicted in Figure 4.9, the operator placement strategy comprises three phases:

Phase 1 – Application Profiling: By using the infrastructure controller layer, the worker RPs and the master RPs continuously collect statistics [83] 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 message;
- Memory to run the operator;
- Arrival message size; and

- Outcome message size;

This information is used to establish the selectivity, operator transformation pattern, as well as, the CPU and memory requirements.

Phase 2 – Candidate Placement: The candidate placement is established using phases 1, 2, 3, and 4 presented in Section 4.2.1. After that, the operators are organised following whether their data flows to sinks placed on the edge or the cloud. For operators that send data only to the cloud, their target infrastructure is the cloud, otherwise, it is the edge.

Phase 3 – Final Placement: Once phase 2 has completed and the profiling phase has established the requirements of the different operators, an operator placement strategy is created and deployed. The strategy reduces the combinatorial space by estimating only once the computation and communication overheads (Section 3.3.2) to operators targeted to cloud (Phase 2). Otherwise, operators to edge (edge candidate placements) have their overheads estimated for all edge devices evaluating their constraints (Section 3.3.3). The strategy gives high priority to edge since cloud sinks often store 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. Along with the overhead estimations, the strategy greedily uses the edge candidate placements for sequentially estimating the *agg_cost* and at each iteration, it picks the device with the minimal value to assign the operator.

4.3.3 Experimental Setup and Performance 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.

Experimental Setup

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

- We used an experimental edge testbed inspired by Hu *et al.* [81] that consists of 13 Raspberry Pis; 5 Raspberry Pis model 3 (4x ARM Cortex-A53 1.2 GHz, 1 GB of RAM and 10/100 Ethernet), and 8 Raspberry Pis model 2 (4x ARM Cortex-A7 900 MHz, 1 GB of RAM and 100 Ethernet).
- For the cloud we used the Chameleon cloud [38] 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 [68] (the Internet) for connecting to the cloud. The LAN has a latency of 0.523 ms and a bandwidth of 15 Mbits/sec. The WAN has latency of 66.75 ms, and bandwidth of 87.0 Mbits/sec.

All the tests are evaluated using the ETL dataflow. The ETL dataflow is an implementation of the ETL RIoTBench topology and 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[6] which consists of transmitting data each minute from sensors in 7

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

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).

Metrics: The performance metrics consist of *end-to-end application latency*, *WAN traffic*, and *monetary cost for communication*. We compare our strategy against the following strategies:

- *Cloud* and *LB* [134] as presented in Section 4.2.3; and
- *Random* which 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.

Evaluation of End-to-end Application Latency

The conducted experiment evaluates the end-to-end application latency using Equation 3.25 presented in Section 3.3.5, 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 minimising the end-to-end application latency and compare the R-Pulsar operator placement approach. In addition, three failures were manually injected to showcase the dynamicity and flexibility to recover from node failures (see Figure 4.10). 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).

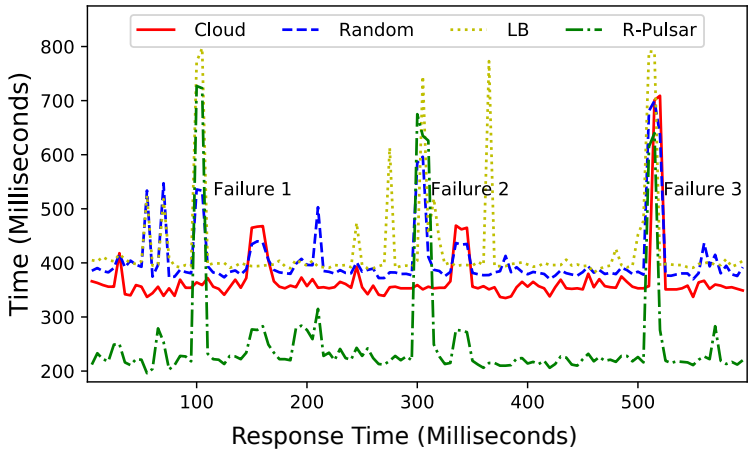

Figure 4.10: End-to-end application latency optimisation with 3 self injected failures affecting edge and cloud resources, while comparing R-Pulsar with Cloud, Random and LB approaches.

Figure 4.10 shows that on average messages 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 4.10 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 application latency.

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

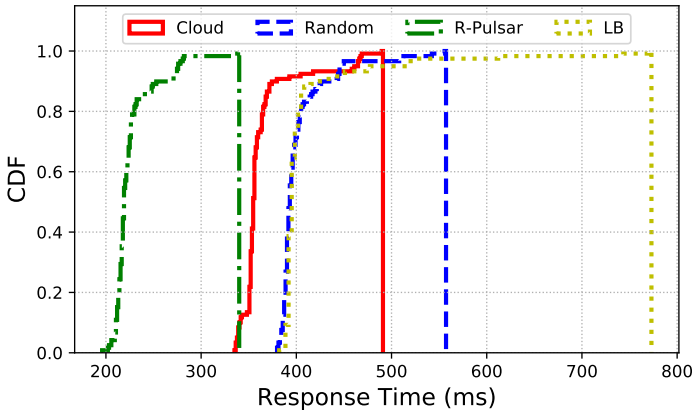

Figure 4.11: End-to-end application latency optimisation cumulative distribution function (CDF) comparison with Cloud, Random and LB approaches.

Figure 4.11 shows that when R-Pulsar operator placement approach is used 80% of the messages see a reduction in the end-to-end application latency by 44% compared to the LB and Random approaches and 38% compared to the Cloud.

Evaluation of Data Transfer Rate

The Data transfer rate consists of the sum of all message sizes that traverse a WAN link per second. The values for Equation 3.25 presented in Section 3.3.5 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 minimising the transfer rate between the edge and the cloud and compare the results with R-Pulsar operator placement approach. Minimising the transfer rate between the edge and the cloud is a critical point in order to achieve real-time analytics.

Figure 4.12 shows that 80% of the time R-Pulsar reduces the transfer rate between the edge and the cloud on average by 35% when compared to the LB approach. It reduces the data transfer rate by 45% when compared to the Cloud and Random.

This next experiment aims to evaluate the efficiency of minimising 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 4.13 shows that the R-Pulsar operator placement approach can also optimise 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 approach.

Evaluation of Messaging Cost

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 3.25 presented in Section 3.3.5 with w_c equal

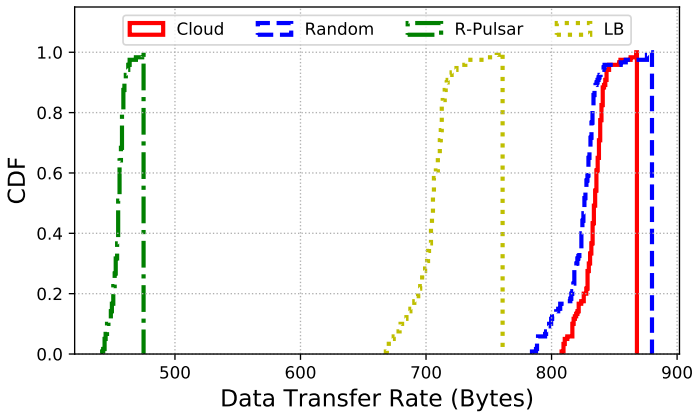

Figure 4.12: End-to-end data transfer rate optimisation cumulative distribution function (CDF) comparison with Cloud, Random and LB approaches.

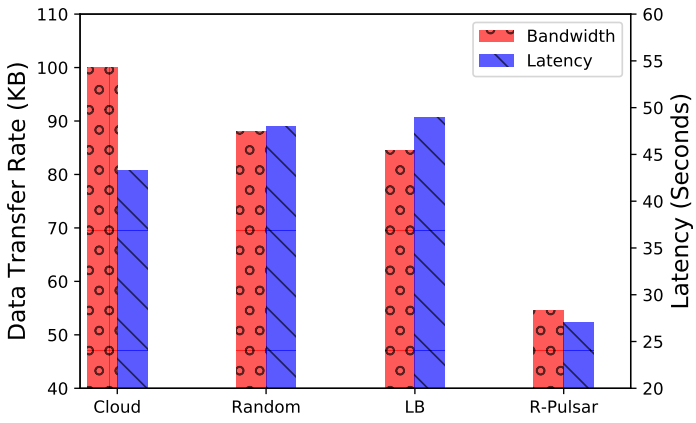

Figure 4.13: Multi optimisation evaluation, end-to-end application latency and data transfer rate comparison with Cloud, Random, and LB approaches.

to 1, and w_l and w_w equal to 0. The goal of this optimisation is to reduce the number of messages that reach the cloud servers.

Table 4.4: 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 4.4 depicts two IoT cost models. The first cost model comes from the Microsoft Azure IoT Hub [98]. 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 comes from the Amazon IoT Core [23] 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.

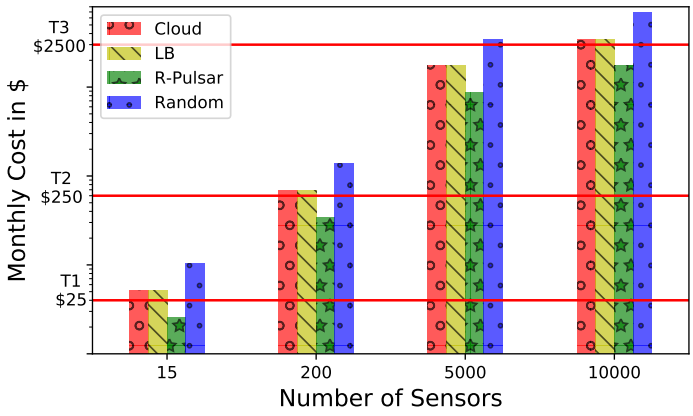

Figure 4.14: Messaging cost savings evaluation based on the Microsoft Azure IoT Hub pricing model, for four different setups.

Figure 4.14 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 behaviour happens with a medium (200 sensors) and extra large (10,000 sensors) setups.

Figure 4.15 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 in all four

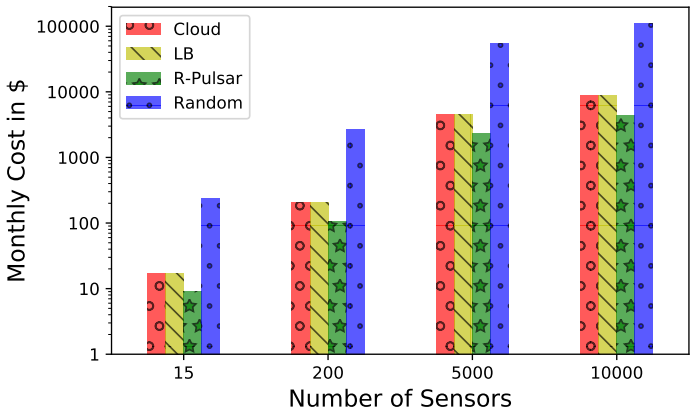

Figure 4.15: Messaging cost savings evaluation based on the Amazon IoT pricing model, for four different setups.

setups (15, 200, 5,000 and 10,000 sensors). In addition, our system obtains a 97% savings when compared to the Random approach in all four different setups.

4.4 Conclusion

In this chapter, we described two strategies to minimise an application's response time by splitting its graph dynamically and distributing its operators across cloud and edge resources. Our solutions were evaluated considering key aspects to identify application behaviours. The RTR strategy estimates the response time of each operator in all computational resources while the RTR+RP strategy splits the dataflow using region patterns and then calculates the response time only for operators that are candidates to be deployed on the edge. We simulated the strategies' behaviour and compared them against the state-of-the-art. The results showed that our strategies are capable of achieving 50% better response time than Cloud deployment when applications have multiple splitters and mergers.

This chapter also described R-Pulsar, a framework for solving the operator placement problem in IoT scenarios. The RTR+RP strategy was extended by considering data transfer rates and messaging costs. The strategy was implemented as a programming model for specifying how DSP applications will be split across the edge and the cloud. Evaluation of this application configuration was performed using an experimental testbed comprising edge and cloud resources on which we deployed and executed real-world DSP applications. The RTR+RP extension was evaluated against three other strategies from the literature, showing the ability of the strategy to efficiently place the computations across the available computing resources. Results showed that the system was capable of reducing the end-to-end latency in over 38%, the data transfer rate in up to 38% and save the communication costs in over 50%.

While this chapter addressed the application configuration, during an application's life cycle things can change; changes that can require an application to be reconfigured. In the next chapter, we address the application reconfiguration as a Reinforcement Learning (RL) problem. We introduce a Markov Decision Process (MDP) model and apply it to a proposed algorithm and baseline RL algorithms.

Chapter 5

Reinforcement Learning Algorithms for Reconfiguring Data Stream Processing Applications

Contents

5.1 Introduction

After an initial placement, operators may need to be reconfigured due to variable load conditions or device failures. The solution search space for the application reconfiguration can be enormous depending on the number of operators, streams, resources, and network links. Existing work has investigated Reinforcement Learning (RL) and Monte-Carlo Tree Search (MCTS) to tackle problems with large search spaces and states. For instance, some solutions explore how to improve Quality of Service (QoS) metrics in Data Stream Processing (DSP) scheduling [94, 104] while others focus on DSP elasticity in a general manner [117].

In this chapter, we model the environment for application reconfiguration as a Markov Decision Process (MDP), and employ RL algorithms considering either single and multiple QoS metrics. The proposed RL algorithms with the MDP framework makes part of the *scheduler* on the Orchestrator component. The algorithms interact with the *monitor* components on the Workers to collect performance metrics and gather information for assisting in their decisions. Once the RL algorithms have enough data to establish statistics, they analise the QoS metrics, plan the application reconfiguration and then communicate the migrations to the Dispatching Service. The Dispatching Service is in charge of triggering the migrations providing the right location of the operator and transferring it by the Data Transfer Service. While running the migrations the whole application dataflow is paused, the operators are migrated in parallel, and the produced events are stored in data sources until all the migrations are finished. Our proposed RL algorithm minimises the end-to-end application latency, and results show that it is capable of achieving similar or better latency improvement while requiring fewer operators to be migrated when compared to baseline RL algorithms. We also implement a multi-objective approach considering metrics such as WAN traffic, reconfiguration overhead, the monetary cost of communication, and end-to-end application latency. Results demonstrate that when compared to state-of-the-art approaches on operator placement, our version of RL algorithms reduces in over 50% the target QoS metrics.

5.2 Background on Reinforcement Learning

The concept of learning through interacting with an environment is probably the main idea when we think about the nature of learning. For instance, much of our knowledge is based on experiences resulting from interactions. Whether we are learning to drive a car or developing a conversation, we are aware of the feedback of our environment concerning what we do, and we try to influence the environment through our behaviour.

Similarly, computers can learn from experience and improve methods and algorithms which are well explored on Artificial Intelligence. *Reinforcement Learning (RL)* accounts specially for goal-directed learning by interacting with the environment. Modeling a problem considering an RL approach involves learning on how to map situations to actions in order to optimise a given numerical reward signal. Essentially, the RL system is a *closed-loop* problem since taken actions influence the later inputs.

One of the challenges considering RL is the trade-off between exploration and exploitation. Often the Reinforcement Learning agent prefers actions that it has tried in the past which resulted in a more effective reward. However to discover such actions, the agent has to try actions that have not been tried before. The agent has to exploit what it already knows to be good actions to maximise the reward, but it also has to explore new actions to learn making better selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The RL agent must vary actions and progressively favor those which result in a better reward.

The RL environment is generally modelled as an MDP framework. The MDP provides a decision-making framework where an RL agent makes decisions by interacting with a simulated environment over a number of steps. An MDP is composed of the following:

- *States*, which correspond to the set of environment states;
- *Actions*, to be taken in the environment;

- *Reward function*, which defines the reward derived from taking a given action in a given state; and
- *Transition dynamic*, which represents the probability of taking an action in a given state.

The goal of solving an MDP is to determine the mapping from states to actions (*i.e.*, policy), which maximises the reward. When the transition model and reward function are available, dynamic programming easily solves this task. Otherwise, the concept of an iterative approach remains the backbone of most RL algorithms. These algorithms apply greedy approaches based on Monte-Carlo (MC) and/or Temporal Difference (TD) in order to keep track of state transitions when evaluating the application using the MDP framework and by applying mathematical approaches to balance exploration of new solutions and exploitation of good and well-known ones. This chapter focuses on MCTS, Temporal-Difference Tree Search (TDTS), and Q-Learning algorithms, which are derived from MC and TD to investigate application reconfiguration.

5.2.1 Monte-Carlo Tree Search

MC methods sample sequences of states, actions, and rewards by interacting with an environment. Using only current experience from the environment to learn is striking because it requires no prior knowledge of the transition dynamic or the policy for determining actions from given states, but can still attain optimal behaviour. One powerful approach is learning by simulated experience. Although a model is required, it must provide sample transitions, avoiding the entire probability distributions of the transition dynamics required for dynamic programming. MC methods help for solving the RL problem using averaging sample returns.

One algorithm derived from MC is MCTS which applies heuristic search methods to establish the best available action in a given situation. It obtains knowledge by simulating the given problem and thus requires at least its sample model, which is simpler than discovering the complete transition model of a given task. The MCTS incrementally builds a search tree which is led by the most promising direction using an exploratory action-selection policy. The MCTS algorithm consists of a closed-loop, and when increasing the number of iterations the algorithm brings more precise solutions. The algorithm builds a search tree using the results of the iterations [58]. Formally, each node $n(s)$ of the search tree $\mathcal{T}$ represents a state s that has been seen during simulation. A node/state maintains a count $N(s)$ with the number of times it was visited, an action value $Q(s, a)$ for each action $a \in A(s)$ and a count $N(s, a)$ with the number of times the action was picked. An iteration usually consists of four consecutive phases:

- *selection* which represents the election of actions already memorised in the tree using a *tree policy* or employing a *default policy* for non memorised actions;
- *expansion* which uses the selected action to add a new node into the tree;
- *playout* which employs the MDP model to determine a new state and reward using the current state of the environment and the selected action; and
- *backpropagation* which backpropagates the result from the playout to feedback up the tree.

A simulation or episode starts at the root state s_0 and is divided into two phases. For each episode, the estimated value function can be updated with an incremental mean. When a state s_t is found in the search tree, a *tree policy* is employed to select an action. Otherwise in the second phase a default policy continues the simulation until a terminal state. The simplest policy is

greedy that selects $max_a Q(s,a)$ in the first phase and random actions during the second phase. MCTS attempts to approximate value functions from experience.

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(λ) change the MCTS behaviour to obtain the action-value $Q(s,a)$. Each algorithm mainly varies the Tree Policy and backpropagation – different approaches to feedback up the obtained results of the iteration in tree – methods by employing Upper Confidence Bound (UCB) and/or Temporal Difference.

MCTS-UCT: MCTS with Upper Confidence Bounds for Trees (UCT) is an algorithm [132] 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 [132] 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:

$$UCT(s,a) = Q(s,a) + C\sqrt{\frac{2lnN(s)}{N(s,a)}} \tag{5.1}$$

$$a^* = max_a Q(s,a) \tag{5.2}$$

5.2.2 Temporal Difference Tree Search

TD learning combines MC and dynamic programming ideas. Similar to MC, TD can use raw experience to learn without knowing the transition model or the policy. While MCTS needs to wait until an episode ends to update the node and action values, the basic implementation of TD, *i.e.*, $TD(0)$, waits until the next step and updates the values after transitioning to a new state s_t and receiving the reward $R(s_t)$. Since TD bases its update on an existing estimate, it is said to be a *bootstrapping* method. A TD variant that unifies TD and MCTS and allows for specifying the number of future states on which estimates are evaluated is $TD(\lambda)$ where a large value for λ will eventually result in MCTS behaviour.

TDTS-Sarsa(λ): is a TD method that combines Sarsa(λ) and UCT algorithms. The general Sarsa derives its name from how the policy evaluation algorithm is structured, where a state-action (S,A) pair yields a reward R and takes the execution to a new state, S', at which the policy picks action A', and the value of this transition is evaluated to $Q(S',A')$. Sarsa(λ) considers m steps of experience [140].

5.2.3 Q-Learning

Q-learning algorithm has an agent that tries to learn an optimal state-action transition policy based on state-action rewards that it receives by interacting with the environment [132]. The agent computes the return, the so called Q-values, of state-actions so that it picks actions that maximise the reward. With Q-values computed the value of state s is:

$$Q(s) = max_a Q(s, a) \tag{5.3}$$

Action values are updated when transitioning from state s to s' as follows:

$$Q(s, a) = Q(s, a) + \alpha[R(s) + \gamma max_{a'} Q(s', a') - Q(s, a)] \tag{5.4}$$

where α is the learning rate and γ is the discount factor [132].

All mentioned RL algorithms often employ a model to simulate the interaction with the environment. Hence, in the next section, we describe an MDP model for application reconfiguration.

5.3 Modeling DSP Application Reconfiguration as an MDP

The application reconfiguration is structured as an 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 Chapter 3 for evaluating the impact of operator migrations.

An MDP provides a decision-making framework where an agent makes decisions by interacting with a simulated environment over a number of steps. As described earlier, an MDP 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 5.1 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 using breadth-first search [107] 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 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 3.3.3).

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 $\mathbb{1}_{\langle i,a \rangle} = 1$; where $\mathbb{1}_{\langle i,a \rangle} = 1$ indicates that operator i is placed on resource a. 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 5.1, which depicts the MDP-simulated deployment of the example given in Figure 3.3, the state s_t has index d_t

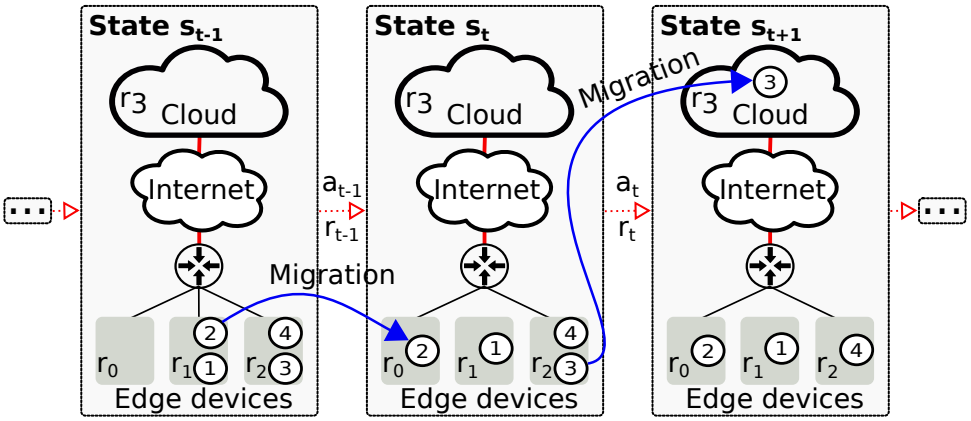

Figure 5.1: Example of MDP-based operator reconfiguration.

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 *cost* of the new state using the model of Chapter 3. The cost function varies depending on the selected approach, whether it is single or multi-objective.

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 $cost = -1$ indicating constraint violations, or the *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 $cost_{s_0}$ of the original mapping and the $cost_{s_{t+1}}$ of the state s_{t+1}. In other words:

$$R(s) = cost_{s_0} - cost_{s_{t+1}} \tag{5.5}$$

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. By considering that, we propose two approaches as follows:

1. Single-objective: we extend MCTS-UCT to consider all UCT when deciding despite looking at the UCT in hierarchical form. The new version of the algorithm, called MCTS-Best-UCT, covers the single-objective approach presented in Section 3.3.5, and we called it as

MCTS-Best-UCT. Together with the algorithm, we introduce a domain optimisation to improve the action searching when moving from one state to another.

2. Multi-objective: we employ known RL algorithms along with the multi-objective function presented in Section 3.3.5.

5.4 Single-Objective Reinforcement Learning Algorithm

This section presents a domain optimisation to sort operators to be migrated, and an algorithm to devise reconfiguration plans.

5.4.1 Building a Deployment Hierarchy

In some IoT scenarios, DSP applications have data sinks placed on the cloud that store data or send data to external application while data sinks on edge feed actuators or alert data sinks, which are time-sensitive. The search space of the application reconfiguration can be enormous due to the number of available computing resources and application operators. When addressing only the Aggregate End-to-End Latency (AL) metric into the optimisation problem, we can assume that operators that flow data only to data sinks on the cloud have less priority than operators that send data to sinks on the edge. From this a priori knowledge, we propose a domain optimisation to sort operators when evaluating the application reconfiguration. The approach increases the chance that operators with greater impact on AL are evaluated first.

The operators are sorted using the candidate infrastructure of Response Time Rate with Region Patterns (RTR+RP) algorithm presented in Section 4.2, building a Deployment Hierarchy (DH) to be exploited by the MDP. Cloud candidates in the RL algorithm iteration do not move from the cloud until the algorithm explores all combinations of edge candidates on available computing resources. We also ignore sources and sinks as their placement is user-defined and do not change.

5.4.2 Traditional MCTS-UCT

MCTS, depicted in Algorithm 3, is a search mechanism consisting of running a number of simulations to build a search tree with the results [58]. 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(λ).

Algorithm 3: The MCTS algorithm.

```
1  Function MCTS(s_0)
2      T ← n(s_0)
3      while within computational budget do
4          episode ← GenerateEpisode (T, s_0)
5          ExpandTree (T, episode)
6          Backup (T, episode)
7      return n(s) ∈ T with the best reward
8  Function GenerateEpisode(T, s_0)
9      episode ← {}
10     s ← s_0
11     while s is not terminal do
12         if n(s) ∈ T then
13             a ← TreePolicy (s)
14         else
15             a ← DefaultPolicy (s)
16         (s, a, r, s') ← SimulateTransition (s, a)
17         append (s, a, r, s') to episode
18         s ← s'
19     return episode
20 Procedure ExpandTree(T, episode)
21     (s, a, r, s') ← first tuple where n(s') ∉ T ∧ s' ∈ episode
22     T ← T ∪ n(s')
23     Q(s', a) ← 0
24     N(s', a) ← 0
25 Procedure Backup(T, episode)
26     R ← r from episode(|episode| − 1)
27     for i = Length(episode) down to 1 do
28         (s, a, r, s') ← episode(i)
29         UpdateTreeValues (T, s, R)
```

5.4.3 MCTS-Best-UCT

The algorithm extends MCTS-UCT by storing the UCB value at each node and enabling a search for the best UCT node. The MDP reconfiguration model allows for making any node terminal as it contains a valid stream and operator mapping. Algorithm 4 depicts the $MCTSBestUCT$ function that receives the current mapping (s_0), initialises the search mechanism and builds the tree. $MCTSBestUCT$ is similar to MCTS-UCT, but with different function behaviours. $TreePolicy$ (line 9) returns the node with highest UCT value, whereas MCTS-UCT starts the search at the root and estimates the UCT values up to a new state $n' \notin \mathcal{T}$. $DefaultPolicy$ (lines 15-17) expands and simulates ($f(s(n), a)$) for the new node (n') taking a random action from the input node. *Backup* (lines 19-23) includes and updates the UCB value for each node as well as the $N(s)$ count and Q value.

Algorithm 4: The MCTS-Best-UCT algorithm.

1 **Function** *MCTSBestUCT(s_0)*
2 create root node n_0 with state s_0
3 **while** *within computational budget*:
4 $n \leftarrow$`TreePolicy()`
5 $n', \Delta \leftarrow$`DefaultPolicy(`n`)`
6 `Backup(`n', Δ`)`
7 **return** `BestChild()`
8 **Function** *TreePolicy(n)*
9 **return** `BestChild()`
10 **Function** *Expand(n)*
11 choose $a \in$ untried actions from $A(s(n))$ at random
12 add a new child n' to n with $s(n') = f(s(n), a)$ and $a(n') = a$
13 **return** n'
14 **Function** *DefaultPolicy(n)*
15 $n' \leftarrow$`Expand(`n`)`
16 $\Delta \leftarrow R(s(n'))$
17 **return** n', Δ
18 **Function** *Backup(n, Δ)*
19 **while** n *is not null*:
20 $N(n) \leftarrow N(n) + 1$
21 $Q(n) \leftarrow Q(n) + \Delta$
22 $UCT(n) = \frac{Q(s,a)}{N(s,a)} + C\sqrt{\frac{2lnN(s)}{N(s,a)}}$
23 $n \leftarrow$ parent of n
24 **Function** *BestChild()*
25 **return** $\underset{n' \in \mathcal{T}}{\operatorname{argmax}}\, UCT(n')$

5.4.4 Experimental Setup and Performance Evaluation

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

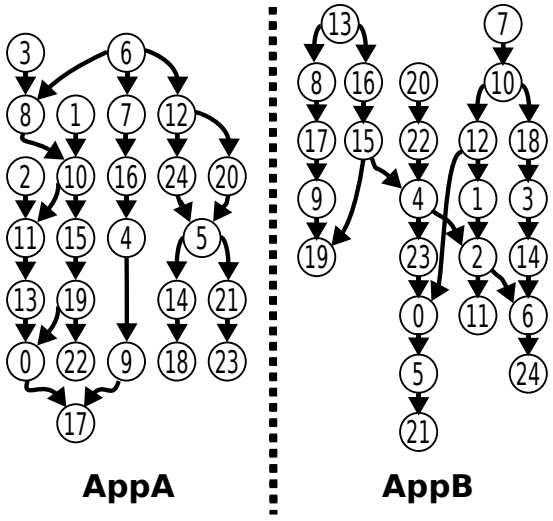

Figure 5.2: Evaluated applications.

Experimental Setup

The infrastructure is the same as in Section 4.2. That is, 2 cloudlets with 20 Raspberry PI's 2 each and a cloud with two servers[1].

We consider two dataflows, shown as AppA and AppB in Figure 5.2, with multiple data paths. The edge hosts sources (3, 6 for AppA and 7, 13 for AppB) and sinks (18, 23 for AppA and 11, 19, 24 for AppB), except for the sink on the critical path (17 for AppA and 21 for AppB), which is hosted on the cloud.

As performance metrics we consider:

- **latency improvement**, which represents the best percentage of latency improvement at a given number of simulations of the MCTS algorithms and Q-Learning;
- **number of operator migrations**, which refers to migrations needed by the devised plan to achieve the latency improvement; and
- the **minimal AL** (in ms), which is achieved when the simulations finish.

Evaluation of End-to-End Application Latency

We evaluate the algorithms under three scenarios:

- **Scenario 1**: Deploying the whole DSP application wherein the MCTS algorithms, Q-Learning and MCTS-Best-UCT receive the Cloud placement and run a budget of 10000 simulations to devise a reconfiguration.
- **Scenario 2**: Applying DH and conducting the experiments as above.
- **Scenario 3**: Evaluating the minimal AL against state-of-the art algorithms (Cloud, Taneja's [134], RTR and RTR-RP [130]) and the baseline approaches in MCTS, Q-Learning and MCTS-Best-UCT.

[1]All evaluations use the built-in-house framework, network infrastructure, Python library for crafting the applications and operator behaviours described in Section 4.2.3.

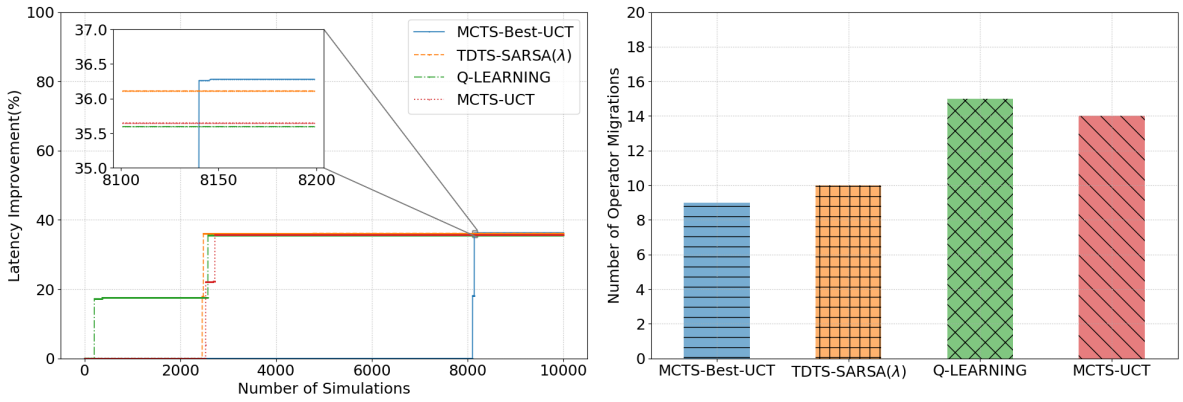

Figure 5.3: Latency improvement and operator migrations for AppA without DH.

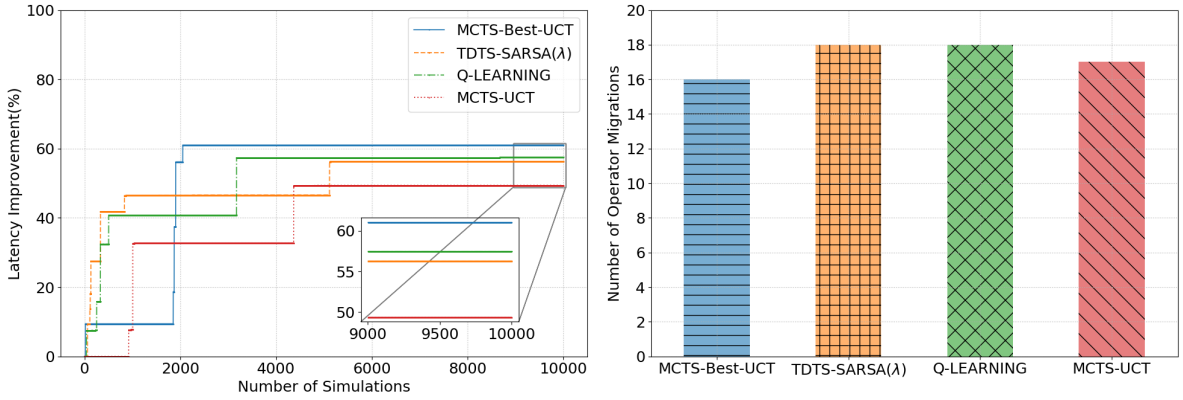

Figure 5.4: Latency improvement and operator migrations for AppB without DH.

Scenario 1: Figure 5.3 shows the results on latency improvement and number of migrations for AppA. MCTS-Best-UCT needs more simulations to start improving the latency, but at 8140 simulations it outperforms the other approaches until the budget finishes. Although it achieves less than 1% of latency improvement compared to TDTS-SARSA(λ) and $\approx$ 2% compared to the remaining approaches, as the right-hand graph shows, it requires less migrations to achieve the latency improvement (10% less migrations than TDTS-SARSA(λ), and 35% less compared to Q-Learning and MCTS-UCB). This is because MCTS-Best-UCT maintains a global view of UCB values and at each simulation expands the tree from the node with the best UCB.

Figure 5.4 presents the results for AppB. MCTS-Best-UCT reaches latency improvement with less simulations outperforming Q-Learning, MCTS-UCT and TDTS-SARSA(λ) by over 3%, 11% and 4%, respectively. As shown on the right-hand figure, MCTS-Best-UCT requires 16 operator migrations while Q-Learning, MCTS-UCT and TDTS-SARSA(λ) perform 18, 17 and 18, respectively.

Scenario 2: Figure 5.5 shows the latency improvement and number of migrations for AppA. DH prioritises reassigning operators that process and forward events to sinks placed on the edge and does not reassign operators that forward events only to the cloud. As the left-hand figure shows, the number of simulations to find a good solution decreases. MCTS-Best-UCT requires $\approx$ 8000 simulations without DH and $\approx$ 1800 with DH. MCTS-Best-UCT achieves the best latency

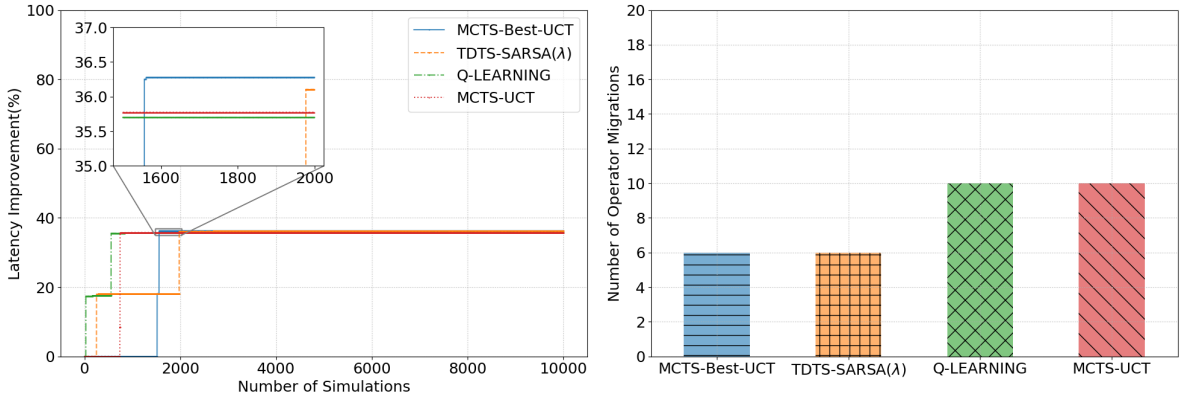

Figure 5.5: Latency improvement and operator migrations for AppA with DH.

and the least number of migrations, down from 9 without DH to 6 with DH.

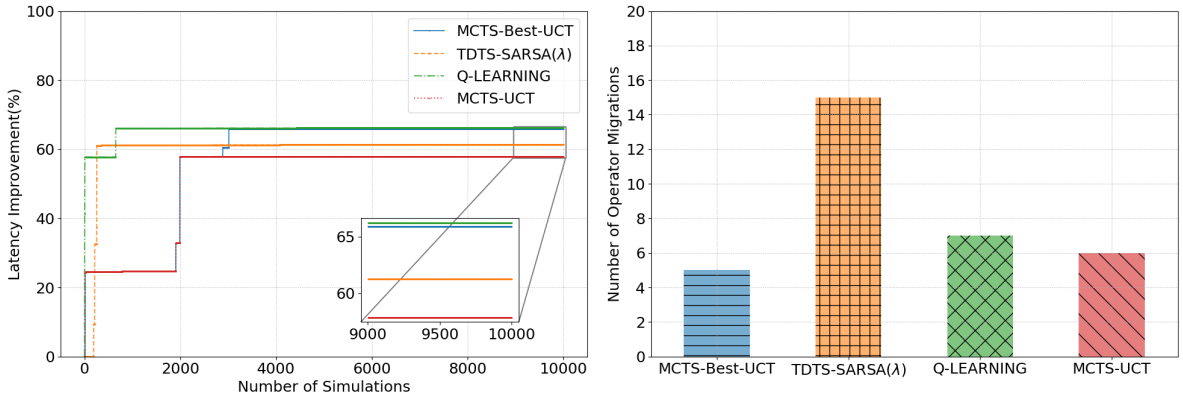

Figure 5.6: Latency improvement and operator migrations for AppB with DH.

Figure 5.6 shows the latency improvement and number of migrations for AppB. With DH the number of migrations decreases by ≈ 31% compared to Scenario 1 and the latency improvement increases by ≈ 7%. MCTS-Best-UCT outperforms MCTS-UCT and TDTS-SARSA(λ) by over 8% and 5%, respectively. Considering Q-Learning, MCTS-Best-UCT has a slight loss in the latency improvement of less than 0.5%, but its is acceptable because Q-Learning is more costly to maintain and search into a lookup table with all possible system states. On the other hand, MCTS-Best-UCT requires only 5 migrations to optimise the end-to-end application latency.

Scenario 3: Figure 5.7 summarises the results on minimal AL with and without DH. The RTR and Cloud were evaluated without applying DH because they consider the whole application dataflow. On the other hand, RTR-RP was evaluated considering DH as the algorithm optimises the number of operators to be assigned. The results show that the RL approaches can improve and provide more stable operator reconfigurations regarding AL than the state-of-the-art. The reason is that RL algorithms balance exploration and exploitation of solutions. Also, considering AL, MCTS-Best-UCT outperformed Taneja's algorithm and Cloud by over 48%, and RTR by over 20% without DH. With DH our proposed algorithm outperformed RTR+RP by over 5%. When compared to the other RL algorithms on average MCTS-Best-UCT reduces the end-to-end

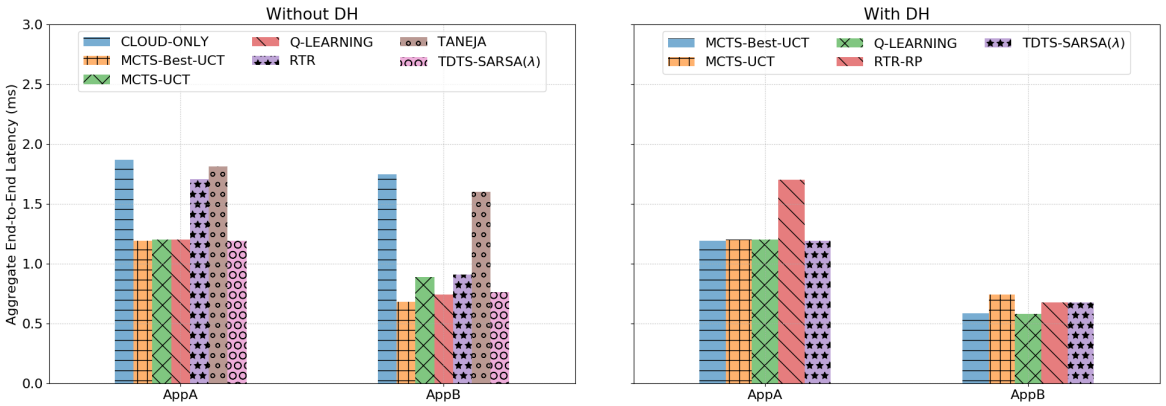

Figure 5.7: Minimal AL of state-of-the-art against RL approaches with/without DH.

application latency by ≈4%.

Our algorithm demonstrated that it improves the AL metric with fewer migrations in application reconfiguration when compared to baseline RL algorithms. When considering the DH, results showed that it achieves better results when considering the same number of iterations. During the evaluation of RL algorithms, we identified their potential in solving large space problems when compared to application configuration solutions. The next section, we advance in the application reconfiguration problem extending existing RL algorithms to handle multiple QoS metrics.

5.5 Multi-Objective Reinforcement Learning Algorithms

We extend MCTS-based algorithms such as MCTS-UCT and TDTS-Sarsa(λ) to address the operator reconfiguration problem by considering multiple metrics such as WAN traffic, reconfiguration overhead, the monetary cost of communication and the end-to-end application latency.

5.5.1 Reinforcement Learning Algorithms

The Q-Learning algorithm has shown to be inefficient when considering a single-objective approach, as presented in Section 5.4. Q-Learning also incurs a high overhead due to maintaining the Q-table; the application reconfiguration problem has a big range of states and actions. These issues led us to work with TDTS and MCTS. In this section, we explore MCTS-UCT and TDTS to identify the application reconfiguration considering a multi-objective approach.

In our version of MCTS, $DefaultPolicy$ method of Algorithm 3 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 $Backup$ method of the algorithm increments the number of node visits $N(s,a)$ and adds the reward from the last tuple to $Q(s,a)$.

5.5.2 Experimental Setup and Performance Evaluation

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

Experimental Setup

The infrastructure is the same as in Section 4.2. That is, 2 cloudlets with 20 Raspberry PI's 2 each and a cloud with two servers[2].

We consider 11 application graphs with single and multiple data paths. Each application has at least 20% of stateful operators.

We use the following QoS requirements as performance metrics:

- **aggregate end-to-end latency**, which is the difference from the time when events are generated to the time they are processed by the sinks;
- **monetary cost**, which represents the cost in dollars by using the Microsoft Azure IoT Hub Pricing policy;
- **WAN traffic**, which corresponds the amount of data transferred inter cloudlet, among cloudlets and cloud communication; and
- **reconfiguration overhead**, which is the maximum time to reassign operators and states across the infrastructure.

The RL algorithms are compared against a traditional deployment approach (Cloud) and a solution from the state-of-the-art that performs cloud-edge placement [134] (LB). Also, we show the benefits and the behaviours of RL algorithms by varying the metric weights.

Performance Evaluation

The performance evaluation was conducted by giving a budget of 10,000 iterations to the RL algorithms and we analyse them under three aspects. First, we compare the RL algorithms against traditional and the state-of-the-art solutions that are oblivious to multi-objective optimisations. Second, we demonstrate the non-negligible impact of the reconfiguration overhead. At last, we show the behaviour of RL algorithms when applying multiple weights to the QoS metrics.

RL algorithms versus traditional and state-of-the-art deployment: Figure 5.8 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, LB and RL algorithms). The left-hand graph in Figure 5.8 demonstrates that RL algorithms can achieve $\approx$ 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 5.8 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 5.9 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

[2]All evaluations use the built-in-house framework, network infrastructure, Python library for crafting the applications and operator behaviorus described in Section 4.2.3.

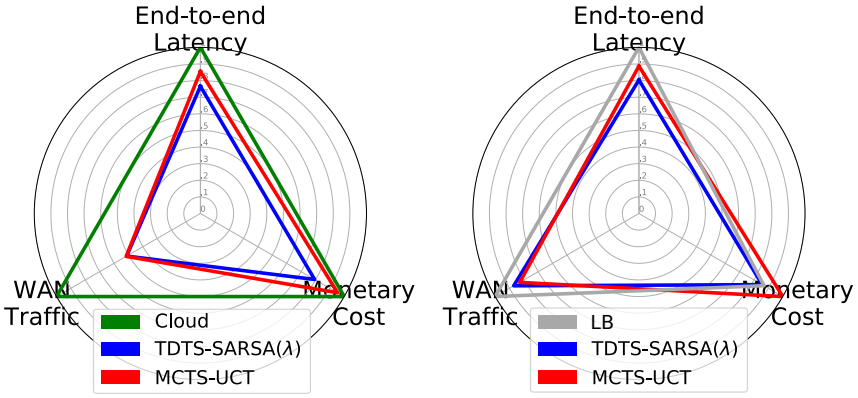

Figure 5.8: Cloud and LB, with end-to-end latency weight = 1.

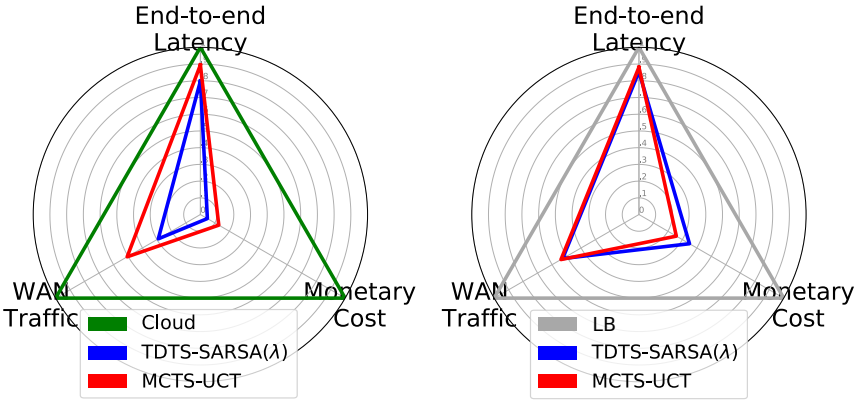

Figure 5.9: Cloud and LB with weights for end-to-end latency, monetary cost and WAN traffic equal to 0.33.

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%.

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 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.

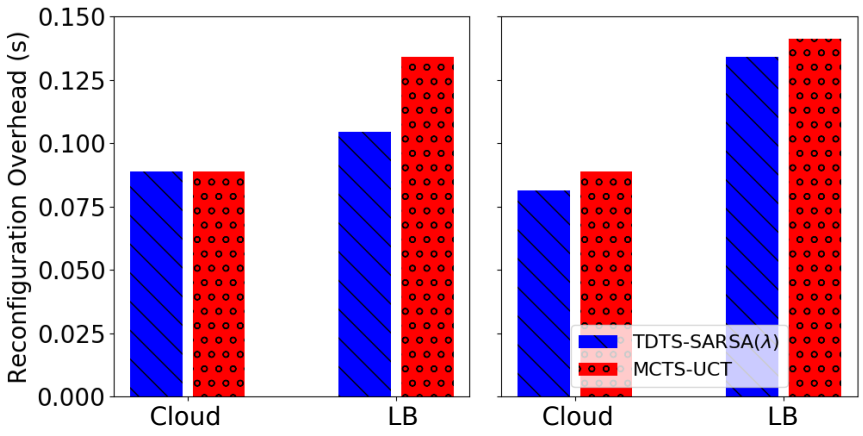

Figure 5.10: Reconfiguration overhead of Cloud and LB. On the left-hand graph the end-to-end latency weight is 1; the right-hand graph shows results of equal weights to end-to-end latency, monetary cost and WAN traffic.

Figure 5.10 presents the reconfiguration overhead for the weighted scenarios of the previous execution. The RL algorithms achieved lower reconfiguration overhead in over 40% when starting from the Cloud approach. As LB 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.* [130]. On the one hand, LB 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, the RL algorithms have to move operators just from the cloud to the edge.

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 5.5.2, but normalised by the maximum observed values from all experiments (two previous scenarios). Hereafter, the term *base values* is used as a reference to the end-to-end latency, monetary cost and WAN traffic obtained from LB and Cloud operator deployments. For the reconfiguration overhead we consider the maximum value from RL algorithms when oblivious to this metric. Figure 5.11 summarises the results of equal weight of 0.25 to all metrics when starting with a deployment using either Cloud or LB. The RL algorithms reduce in over 45% the end-to-end latency, monetary cost and WAN traffic while achieving $\approx$ 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 5.12 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 $\approx$ 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 *agg_cost* presented in Section 3.3.5, and the optimistic approach in the face of uncertainty of UCT-based algorithms – TDTS-Sarsa(λ) and MCTS-

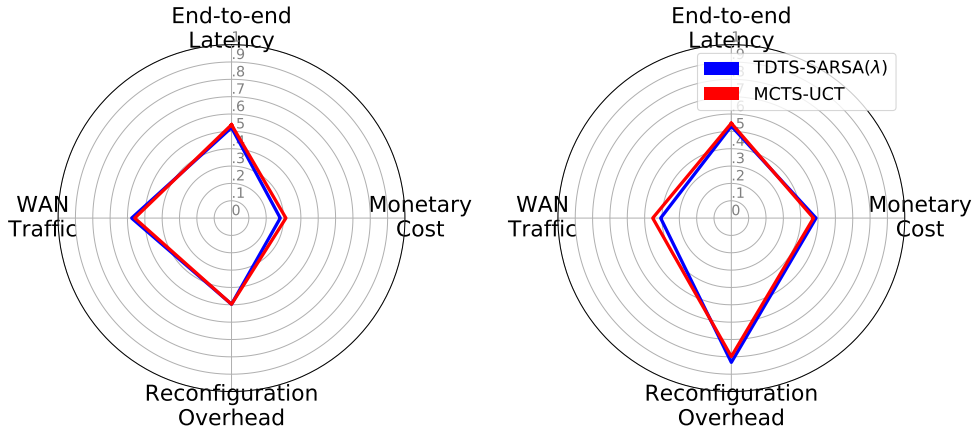

Figure 5.11: Cloud and LB with 0.25 weight for end-to-end latency, monetary cost, WAN traffic and reconfiguration overhead.

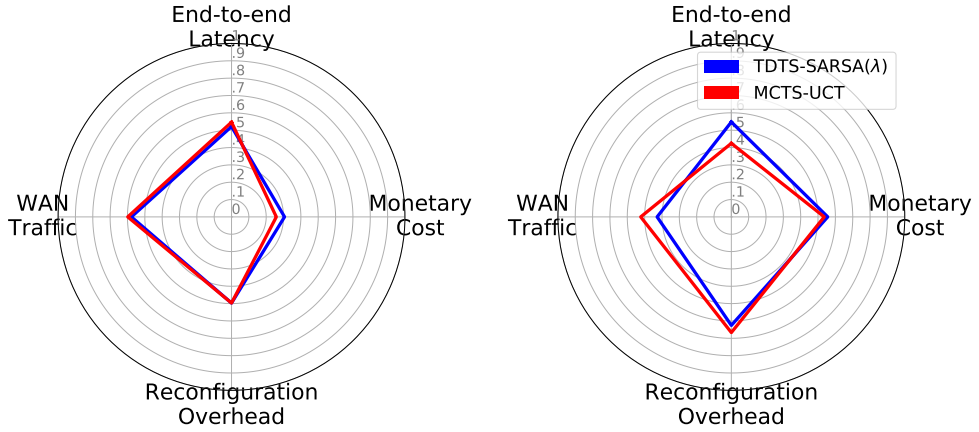

Figure 5.12: Cloud and LB with weights 0.7, 0.1, 0.1, and 0.1 for end-to-end latency, monetary cost, WAN traffic and reconfiguration overhead, respectively.

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 5.13 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 or LB) dictate the performance of the RL algorithms. In this sense, the best configuration was

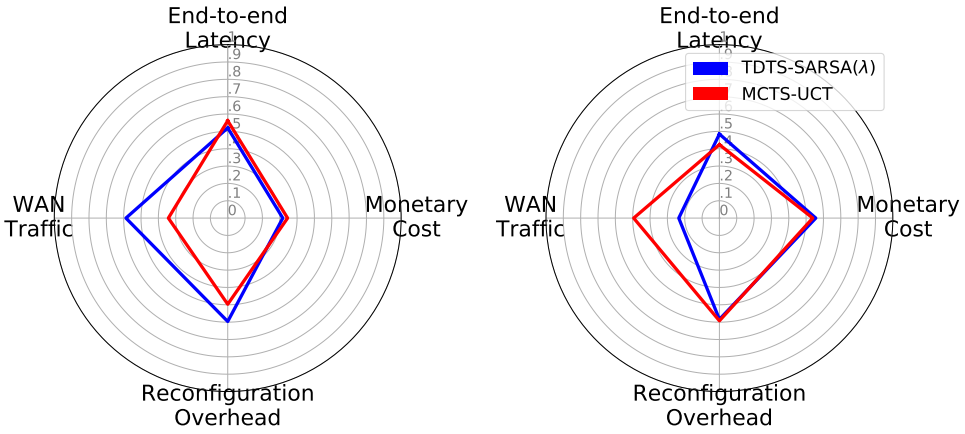

Figure 5.13: Cloud and LB with weights 0.4, 0.2, 0.2, and 0.2 for end-to-end latency, monetary cost, WAN traffic and reconfiguration overhead, respectively.

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(λ) requires a fine tune of its parameters (reward discount and eligibility trace decay rates) as presented in Section 5.2. 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.

5.6 Conclusion

This introduced an MDP model and employed MCTS-UCT and TDTS-Sarsa(λ) algorithms to devise application reconfiguration plans. We proposed an algorithm to minimise the aggregate end-to-end latency while reducing the number of required iterations to create an episode. We also introduced a domain optimisation for RL algorithms in order to reduce the action search space. The method applies a domain knowledge avoiding to reconfiguring operator that only process and forward events to sinks on the cloud. We compared the performance of the state-of-the-art on MCTS algorithms and Q-Learning algorithms against the proposed method and algorithm. The results showed that our approach is capable of achieving similar or better latency improvement while requiring fewer operators to be migrated.

We also extended RL algorithms to tackle multi-objective problem optimisation by considering end-to-end latency, WAN traffic, monetary cost, and reconfiguration overhead. We considered an IoT scenario for evaluating our approach that comprises several DSP applications and a cloud-edge infrastructure. Our version of RL algorithms was evaluated and compared against state-of-the-art solutions. The results showed that our MDP model and multi-objective approach enable RL algorithms to have a holistic view of the application reconfiguration and allow for reducing all target QoS metrics by over 50%.

Chapter 6

Conclusions and Future Directions

Contents

6.1 Discussion

This thesis addressed the challenge of (re)configuring Data Stream Processing (DSP) applications on infrastructure combining cloud and edge computing resources. We focused on mechanisms that can improve the application performance. Solutions for DSP application (re)configuration need to meet QoS requirements such as end-to-end application latency, monetary cost, WAN traffic, reconfiguration overhead, among others.

We investigated existing work on DSP elasticity, and (re)configuration, and enumerated several characteristics of existing systems such as their architectural views, data management and structure, operational models, operator behaviours and scheduler organisation. This investigation revealed:

- An extensive literature on DSP application elasticity, and (re)configuration on the cloud;
- Some efforts on modelling the (re)configuration of DSP applications; and
- Attempts to improve the execution of DSP applications by exploring the edge computing.

The investigation also showed a lack of mechanisms that build on these efforts, that enable placing DSP applications on highly distributed infrastructure, and that meet either a single or multiple Quality of Service (QoS) requirements simultaneously. These lessons led to a proposed set of mechanisms for (re)configuring DSP applications on highly distributed infrastructure. The mechanisms were conceived using Queueing Theory that allows for estimating: the number of

messages waiting to be handled, the time for processing and transferring messages, the required amount of memory for storing states and waiting messages, the arrival rate and departure rate of messages, the required amount of computation to process the arriving number of messages, and so on. The model viewed the DSP system as multi-layered architecture, which is required to execute applications on highly distributed infrastructure. The model also considered existing DSP operator behaviours such as operator selectivity, changes in the volume of data across operators, and the communication overhead in a geographically distributed infrastructure.

The complexity of (re)configuring DSP applications on heterogeneous computing resources inspired the conception of mechanisms based on single and multi-objective optimisation. Instead of hosting all operators required by a DSP solution in the cloud itself, we considered a more decentralised approach where the application is decoupled and placed across multiple geographical locations. Our effort aims to improve resource utilisation and data movement, which can be achieved by deploying DSP operators along a physical path by considering data source and sink locations. In this kind of infrastructure, there are variables such as computational power, network bandwidth, network latency, application constraints, network topology, that have a significant impact on placement decisions and bring complexity to the problem. The problem is exacerbated when considering highly distributed infrastructure as assigning operators to heterogeneous resources has proven to be NP-hard [27]. The concept of exploring the edges allows the DSP applications to reduce QoS metrics such as volume of transferred data, end-to-end application latency and communication costs.

With respect to finding solutions for configuring DSP application, this thesis proposed strategies that rely on the end-to-end application latency for meeting the single objective optimisation. When considering multiple objective optimisation it extended one of the proposed strategies to include metrics as monetary cost and WAN traffic. The configuration strategies look at the DSP application as Series-Parallel-Decomposable Graphs (SPDG), where we investigated techniques for decomposing the graph and providing an operator ordering by giving priority to upstream operators when evaluating their placement. In addition, experiments evaluated the performance of the strategies by considering microbenchmarks and complex applications with multiple paths. Simulation results obtained by modelling the cloud-edge infrastructure and DSP application showed that it is possible to split the DSP operator graph across edge devices and cloud dynamically thus improving the end-to-end latency and respecting edge devices and network constraints. The benefits derive from how effective the approach reduces the communication overhead. By doing so, it compensates the limitations of edge devices and minimises the end-to-end application latency by over 50% when compared to Cloud only deployment.

We also extended one strategy to include new QoS metrics and implemented it in the R-PULSAR framework, which is a real-life Data Stream Processing Engine (DSPE) developed to deal with cloud-edge infrastructure. Results showed that our approach can save over 50% the communication costs, reduce in up 38% data transferred across computing resources, and minimise over 38% the end-to-end application latency. Our approach allows a user to specify the importance of QoS metric weights by employing a Simple Additive Weighting method where multiple metrics can be handled simultaneously.

This effort on configuring DSP applications motivated the investigation of mechanisms for another phase of the DSP application life-cycle, namely the application reconfiguration. DSP applications are long-running, and the load and infrastructure conditions can change after their initial placement, which can require their reconfiguration. Meanwhile, we observed how emerging techniques on Reinforcement Learning (RL) had proposed efficient mechanisms to address problems with large search spaces. As a result of our investigation, we applied RL mechanisms

to handle the application reconfiguration problem. We contributed with an Markov Decision Process (MDP) model, an RL algorithm based on MCTS-UCT, and a domain optimisation for improving the RL action search space.

The proposed algorithm and domain optimisation were employed to address the minimise the end-to-end application latency while reconfiguring a DSP application. Simulation results showed that domain optimisation allowed the RL algorithms to achieve better end-to-end application latency in fewer iterations, while our algorithm achieved good latency improvement with fewer operator migrations when compared to existing RL algorithms. We also extended existing RL algorithms to handle multi-objective optimisation. Simulation results revealed that our approach led to reducing the QoS metrics by over 50% when compared to cloud deployment.

Moreover, the solutions proposed in this thesis can be applied to address multiple societal issues. Improving the latency of applications and resource utilisation can prevent resource wastage and therefore reduce the CO_2 footprint of the employed computing infrastructure. Moreover, IoT is becoming key to optimise the management of industrial and operational infrastructure. By decoupling DSP applications and allowing their placement on cloud-edge infrastructure, we allow for a larger number of applications to be deployed, also enabling executing applications with more stringent requirements of near-real-time. This enables a new range of applications in fields like accessibility, education, sustainability, among other areas.

6.2 Future Directions

Nowadays, organisations often demand not only online processing of large amounts of streaming data, but also solutions that can perform computations on large data sets by using models such as MapReduce. As a result, big data processing solutions employed by large organisations exploit hybrid execution models (e.g, using batch and online execution) that can span multiple data centres. In addition to providing elasticity for computing and storage resources, ideally, a big data processing service should be able to allocate and release resources on demand. This section highlights some future directions in this area.

6.2.1 SDN and In-Transit Processing

Networks are becoming increasingly programmable and flexible by using several solutions such as Software Defined Network (SDN) [85] and Network Functions Virtualization (NFV), which can provide mechanisms required for allocating network capacity for certain dataflows both within and across data centres with certain computing operations been performed in-network. In-transit DSP can be carried out where certain processing elements, or operators, are placed along the network interconnecting data sources and the cloud. This approach raises security and resource management challenges. In scenarios such as Internet of Things (IoT), having components that perform processing along the path from data sources to the cloud can increase the number of hops susceptible to attacks.

Most of the existing work on DSP application (re)configuration considered network metrics such as latency and bandwidth while proposing decentralised algorithms, without taking into account that the network can be programmed and capacity allocated to certain network flows. The interplay between hybrid models and SDN as well as joint optimisation of DSP application (re)configuration and flow routing in edge computing can be better explored.

6.2.2 Investigation of Machine Learning Mechanisms

Emerging cognitive assistance scenarios [69] offer interesting use cases where machine learning models can be trained on the cloud and then be deployed on edge computing resources. This thesis applied RL algorithms with the goal of exploring edge computing resources, but focusing on how to split DSP applications dynamically across edge and cloud computing resources. We focused on applying MCTS-based algorithms and Q-Learning, but our model can be leveraged to implement a big range of ML algorithms for application (re)configuration problem using for example Trust Region Policy Optimisation (TRPO), Support Vector Regression (SVR), among others algorithms.

The solution offered in this thesis for sorting DSP operators improve the chance that operators that have a greater impact on latency are evaluated first. Our solution gives priority to given actions by considering whether an operator sends data to the cloud or when deciding operator migrations. The approach leads to solutions that improve the aggregate end-to-end latency metric. When considering monetary costs for communication and reconfiguration overhead, operators that send data to the cloud must have the same priority because they affect the metrics directly. There is hence a lack of mechanisms to dynamic explore the action search space, for example, Convolutional Neural Networks (CNNs) and Neural Networks (NNs) to create a network of action probabilities at $TreePolicy$. By training this network, the process of picking actions can be improved.

6.2.3 Programming Models for Hybrid and Highly Distributed Architecture

DSPEs that provide high-level programming abstractions have been introduced in recent past to ease the development and deployment of big data applications that use hybrid models [30, 61]. Platform bindings have been provided to deploy applications developed using these abstractions on the infrastructure provided by commercial public cloud providers and open source solutions.

Under the Apache Beam project [10], efforts have been made towards providing a unified SDK while enabling processing pipelines to be executed on distributed processing back-ends such as Apache Spark [151] and Apache Flink [12]. Beam is particularly useful for embarrassingly parallel applications. There is still a lack of unified SDKs that simplify application development covering the whole spectrum, from data collection at the Internet edges to processing at micro data centres (more closely located to the Internet edges) and data centres.

This thesis offered a programming model for specifying how DSP applications should be split across the edge and the cloud. However, this field should be better explored where high-level programming abstractions and bindings to platforms capable of deploying and managing resources under such highly distributed scenarios are desirable.

6.2.4 Simulation Frameworks for DSP Systems

Currently, there are simulators that can handle cloud-edge infrastructure like Omnet++ and others designed specifically for this type of infrastructure such as iFogSim [66] and RECAP [33]. However, they do not offer DSP toolkits, and users must implement support for DSP by themselves, which is time-consuming. Moreover, simulators do not support certain types of networks and protocols that comprise the fabric of edge infrastructure such as LTE, LoRaWAN, Zigbee, among others. Efforts should be made towards developing high-level APIs that facilitate the modelling and simulation of DSP applications and the comparison of scheduling mechanisms.

6.2.5 DSP System, and Failure and Energy Consumption Models

In this thesis, we addressed the problem of application (re)configuration. An extension to this work would consider scenarios where computing resources can fail during the execution of a DSP application, or when there exists energy constraints for edge devices. Nowadays, there is a lack of understanding on the type and frequency of failures and energy consumption of sensors and edge devices. Including these features in our solutions was out of the scope of this thesis. It requires a deep analysis of the cloud-edge infrastructure in order to build failure and energy consumption models. Hence, an effort is required to build models capturing the aforementioned behaviours. A call to arms is necessary towards devising energy consumption and failure models for data stream analytics on cloud-edge infrastructure. Such models could enable the development of more fault tolerant and sustainable solutions for resource management in highly distributed infrastructure.

6.2.6 Scalability

Market research constantly advocates the continuous and exponential growth of IoT and investments in this field[1]. Considering this and the current state of DSPEs, there is a lack of solutions that handle the ever-increasing number of edge devices. This thesis proposed a centralised management solution for DSP systems. Some solutions [45, 145, 146, 154] have explored blockchain-based technology to support distributed control systems and provide decentralised management. It is extremely important that attempts should be made towards decentralised resource management systems mainly considering aspects such as scalability, privacy, and security.

[1] According to Gartner Inc. the market for IoT is likely to have an estimated 20.4 billion connected sensors by 2020 [97]

Bibliography

[1] Yanif Ahmad and Uğur Çetintemel. "Network-aware Query Processing for Stream-based Applications." In: *13th International Conference on Very Large Data Bases – Volume 30*. VLDB '04. Canada: VLDB Endowment, 2004, pp. 456–467.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. "MillWheel: Fault-tolerant Stream Processing at Internet Scale." In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1033–1044. ISSN: 2150-8097. DOI: 10.14778/2536222.2536229. URL: http://dx.doi.org/10.14778/2536222.2536229.

[3] Sean T. Allen, Matthew Jankowski, and Peter Pathirana. *Storm Applied: Strategies for Real-time Event Processing*. First. Greenwich, USA: Manning Publications Co., 2015.

[4] *Amazon CloudWatch*. 2015. URL: https://aws.amazon.com/cloudwatch/.

[5] *Amazon EC2 Container Service*. 2015. URL: https://aws.amazon.com/ecs/.

[6] *Amazon Kinesis Firehose*. 2015. URL: https://aws.amazon.com/kinesis/firehose/.

[7] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. "SPC: A Distributed, Scalable Platform for Data Mining." In: *4th International Workshop on Data Mining Standards, Services and Platforms*. DMSSP '06. Philadelphia, USA: ACM, 2006, pp. 27–37.

[8] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. "Adaptive Online Scheduling in Storm." In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS '13. Arlington, Texas, USA: ACM, 2013, pp. 207–218. ISBN: 978-1-4503-1758-0. DOI: 10.1145/2488222.2488267. URL: http://doi.acm.org/10.1145/2488222.2488267.

[9] *Apache ActiveMQ*. 2016. URL: http://activemq.apache.org/.

[10] *Apache Beam*. 2016. URL: http://beam.incubator.apache.org/.

[11] *Apache Edgent*. 2017. URL: https://edgent.apache.org.

[12] *Apache Flink*. 2015. URL: http://flink.apache.org/.

[13] *Apache Flink – Iterative Graph Processing*. API Documentation. 2017. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/gelly/iterative_graph_processing.html.

[14] *Apache Kafka*. 2016. URL: http://kafka.apache.org/.

[15] *Apache Nifi*. 2019. URL: https://nifi.apache.org/index.html.

[16] *Apache Thrift*. 2016. URL: https://thrift.apache.org/.

[17] *Apache Zookeeper*. 2016. URL: http://zookeeper.apache.org/.

[18] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing*. Technical report UCB/EECS-2009-28. Berkeley, USA: Electrical Engineering and Computer Sciences, University of California at Berkeley, Feb. 2009.

[19] Marcos Dias de Assuncao, Rodrigo N. Calheiros, Silvia Bianchi, Marco A. S. Netto, and Rajkumar Buyya. "Big Data Computing and Clouds: Trends and Future Directions." In: *Journal of Parallel and Distributed Computing* 79–80.0 (2015), pp. 3–15. ISSN: 0743-7315.

[20] Marcos Dias de Assunção, Alexandre da Silva Veith, and Rajkumar Buyya. "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions." In: *Journal of Net. and Computer Applications* 103 (2018), pp. 1–17. ISSN: 1084-8045.

[21] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey." In: *Computer Net.* 54.15 (2010), pp. 2787–2805.

[22] G. S. Aujla, N. Kumar, A. Y. Zomaya, and R. Ranjan. "Optimal Decision Making for Big Data Processing at Edge-Cloud Environment: An SDN Perspective." In: *IEEE Transactions on Industrial Informatics* 14.2 (Feb. 2018), pp. 778–789. ISSN: 1551-3203. DOI: 10.1109/TII.2017.2738841.

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

[24] *Azure IoT Hub*. 2016. URL: https://azure.microsoft.com/en-us/services/iot-hub/.

[25] *Azure Stream Analytics*. 2015. URL: https://azure.microsoft.com/en-us/services/stream-analytics/.

[26] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. "Models and Issues in Data Stream Systems." In: *21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '02. Wisconsin: ACM, 2002, pp. 1–16. ISBN: 1-58113-507-6. DOI: 10.1145/543613.543615. URL: http://doi.acm.org/10.1145/543613.543615.

[27] Anne Benoit, Alexandru Dobrila, Jean-Marc Nicod, and Laurent Philippe. "Scheduling Linear Chain Streaming Applications on Heterogeneous Systems with Failures." In: *Future Gener. Comput. Syst.* 29.5 (July 2013), pp. 1140–1151. ISSN: 0167-739X. DOI: 10.1016/j.future.2012.12.015. URL: http://dx.doi.org/10.1016/j.future.2012.12.015.

[28] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, and Omer Rana. "The Internet of Things, Fog and Cloud continuum: Integration and challenges." In: *Internet of Things* 3-4 (2018), pp. 134–155. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2018.09.005. URL: http://www.sciencedirect.com/science/article/pii/S2542660518300635.

[29] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. "Apache Hadoop Goes Realtime at Facebook." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2011)*. Greece: ACM, 2011, pp. 1071–1080. ISBN: 978-1-4503-0661-4.

[30] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. "Summingbird: A Framework for Integrating Batch and Online MapReduce Computations." In: *Proceedings of the VLDB Endowment* 7.13 (Aug. 2014), pp. 1441–1451. ISSN: 2150-8097.

[31] Berndt Brehmer. "The Dynamic OODA Loop : Amalgamating Boyd ' s OODA Loop and the Cybernetic Approach to Command and Control ASSESSMENT , TOOLS AND METRICS." In: 2005.

[32] T. Buddhika and S. Pallickara. "NEPTUNE: Real Time Stream Processing for Internet of Things and Sensing Environments." In: *IEEE Int. Parallel and Distributed Proc. Symp.* May 2016, pp. 1143–1152.

[33] J. Byrne, S. Svorobej, A. Gourinovitch, D. M. Elango, P. Liston, P. J. Byrne, and T. Lynn. "RECAP simulator: Simulation of cloud/edge/fog computing scenarios." In: *2017 Winter Simulation Conference (WSC)*. Dec. 2017, pp. 4568–4569. DOI: 10.1109/WSC.2017.8248208.

[34] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. "Distributed QoS-aware Scheduling in Storm." In: *9th ACM Int. Conf. on Distributed Event-Based Systems*. DEBS '15. Oslo, Norway: ACM, 2015, pp. 344–347.

[35] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. "Optimal Operator Placement for Distributed Stream Processing Applications." In: *10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. California: ACM, 2016, pp. 69–80. ISBN: 978-1-4503-4021-2.

[36] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. "Optimal operator deployment and replication for elastic distributed data stream processing." In: *Concurrency and Computation: Practice and Experience* 30.9 (2018). e4334 cpe.4334, e4334. DOI: 10.1002/cpe.4334. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4334. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4334.

[37] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi. "Long-range communications in unlicensed bands: the rising stars in the IoT and smart city scenarios." In: *IEEE Wireless Communications* 23.5 (Oct. 2016), pp. 60–67. ISSN: 1536-1284. DOI: 10.1109/MWC.2016.7721743.

[38] *Chameleon Cloud. https://www.chameleoncloud.org/.*

[39] Samantha Chan. *Apache Quarks, Watson, and Streaming Analytics: Saving the World, One Smart Sprinkler at a Time*. Bluemix Blog. June 2016. URL: https://www.ibm.com/blogs/bluemix/2016/06/better-analytics-with-apache-quarks/.

[40] W. Chen, I. Paik, and Z. Li. "Cost-Aware Streaming Workflow Allocation on Geo-Distributed Data Centers." In: *IEEE Transactions on Computers* 66.2 (Feb. 2017), pp. 256–271. ISSN: 0018-9340. DOI: 10.1109/TC.2016.2595579.

[41] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. "Energy Efficiency for Large-Scale MapReduce Workloads with Significant Interactive Analysis." In: *7th ACM European Conference on Computer Systems (EuroSys 2012)*. Bern, Switzerland: ACM, 2012, pp. 43–56. ISBN: 978-1-4503-1223-3.

[42] Bin Cheng, Apostolos Papageorgiou, and Martin Bauer. "Geelytics: Enabling On-Demand Edge Analytics over Scoped Data Sources." In: *IEEE International Congress on Big Data (BigData Congress)*. June 2016, pp. 101–108.

[43] Stephanie Clifford and Quentin Hardy. *Attention, Shoppers: Store Is Tracking Your Cell.* New York Times. 2013. URL: http://www.nytimes.com/2013/07/15/business/attention-shopper-stores-are-tracking-your-cell.html.

[44] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. "Vivaldi: A Decentralized Network Coordinate System." In: *SIGCOMM Comput. Commun. Rev.* 34.4 (Aug. 2004), pp. 15–26. ISSN: 0146-4833.

[45] F. Daniel and L. Guida. "A Service-Oriented Perspective on Blockchain Smart Contracts." In: *IEEE Internet Computing* 23.1 (Jan. 2019), pp. 46–53. ISSN: 1089-7801. DOI: 10.1109/MIC.2018.2890624.

[46] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Communications of the ACM* 51.1 (Jan. 2008).

[47] *DistributedLog*. 2016. URL: http://distributedlog.io/.

[48] R. Eidenbenz and T. Locher. "Task allocation for distributed stream processing." In: *IEEE INFOCOM 2016*. Apr. 2016, pp. 1–9.

[49] M. S. Elbamby, M. Bennis, and W. Saad. "Proactive edge computing in latency-constrained fog networks." In: *European Conf. on Net. and Comm.* June 2017, pp. 1–6. DOI: 10.1109/EuCNC.2017.7980678.

[50] Byron Ellis. *Real-time analytics: Techniques to Analyze and Visualize Streaming Data.* Indianapolis, USA: John Wiley & Sons, 2014.

[51] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. "Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management." In: *ACM SIGMOD International Conference on Management of Data.* SIGMOD '13. New York, USA: ACM, 2013, pp. 725–736.

[52] Daniele Foroni, Cristian Axenie, Stefano Bortoli, Mohamad Al Hajj Hassan, Ralph Acker, Radu Tudoran, Goetz Brasche, and Yannis Velegrakis. "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. 2018, p. 2.

[53] K. Gai, M. Qiu, and H. Zhao. "Cost-Aware Multimedia Data Allocation for Heterogeneous Memory Using Genetic Algorithm in Cloud Computing." In: *IEEE Transactions on Cloud Computing* PP.99 (2016), pp. 1–1. ISSN: 2168-7161. DOI: 10.1109/TCC.2016.2594172.

[54] Keke Gai, Meikang Qiu, Hui Zhao, Lixin Tao, and Ziliang Zong. "Dynamic Energy-Aware Cloudlet-Based Mobile Cloud Computing Model for Green Computing." In: *Journal of Network and Computer Applications* 59.Supplement C (2016), pp. 46–54. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2015.05.016. URL: http://www.sciencedirect.com/science/article/pii/S108480451500123X.

[55] J. Gedeon, M. Stein, J. Krisztinkovics, P. Felka, K. Keller, C. Meurisch, L. Wang, and M. Mühlhäuser. "From Cell Towers to Smart Street Lamps: Placing Cloudlets on Existing Urban Infrastructures." In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. Oct. 2018, pp. 187–202. DOI: 10.1109/SEC.2018.00021.

[56] B. Gedik, H.G. Özsema, and Ö. Öztürk. "Pipelined Fission for Stream Programs with Dynamic Selectivity and Partitioned State." In: *Journal of Parallel and Distributed Computing* 96 (2016), pp. 106–120. ISSN: 0743-7315. DOI: http://dx.doi.org/10.1016/j.jpdc.2016.05.003.

[57] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. "Elastic Scaling for Data Stream Processing." In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (June 2014), pp. 1447–1463.

[58] Sylvain Gelly and David Silver. "Monte-Carlo tree search and rapid action value estimation in computer Go." In: *Artificial Intelligence* 175.11 (2011), pp. 1856–1875.

[59] Rajrup Ghosh, Siva Prakash Reddy Komma, and Yogesh Simmhan. "Adaptive Energy-aware Scheduling of Dynamic Event Analytics Across Edge and Cloud Resources." In: *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid '18. Washington, District of Columbia: IEEE Press, 2018, pp. 72–82. ISBN: 978-1-5386-5815-4. DOI: 10.1109/CCGRID.2018.00022. URL: https://doi.org/10.1109/CCGRID.2018.00022.

[60] Lukasz Golab and M. Tamer Özsu. "Issues in Data Stream Management." In: *SIGMOD Record* 32.2 (June 2003), pp. 5–14. ISSN: 0163-5808. DOI: 10.1145/776985.776986. URL: http://doi.acm.org/10.1145/776985.776986.

[61] *Google Cloud Dataflow*. 2015. URL: https://cloud.google.com/dataflow/.

[62] *Google Cloud Storage*. 2015. URL: https://cloud.google.com/storage/.

[63] *Google Compute Engine*. 2015. URL: https://cloud.google.com/compute/.

[64] L. Gu, D. Zeng, S. Guo, Y. Xiang, and J. Hu. "A General Communication Cost Optimization Framework for Big Data Stream Processing in Geo-Distributed Data Centers." In: *IEEE Transactions on Computers* 65.1 (Jan. 2016), pp. 19–29. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2417566.

[65] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. "StreamCloud: An Elastic and Scalable Data Streaming System." In: *IEEE Transactions on Parallel and Distributed Systems* 23.12 (Dec. 2012), pp. 2351–2365.

[66] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments." In: *Software: Practice and Experience* 47.9 (2017), pp. 1275–1296. DOI: 10.1002/spe.2509. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2509. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509.

[67] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. "SASE: Complex Event Processing over Streams (Demo)." In: *Third Biennial Conference on Innovative Data Systems Research (CIDR 2007)*. Asilomar, USA, Jan. 2007, pp. 407–411.

[68] 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)*. Mar. 2013, pp. 166–176.

[69] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. "Towards Wearable Cognitive Assistance." In: *12th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '14. Bretton Woods, New Hampshire, USA: ACM, 2014, pp. 68–81. ISBN: 978-1-4503-2793-0. DOI: 10.1145/2594368.2594383. URL: http://doi.acm.org/10.1145/2594368.2594383.

[70] Jing Han, Haihong E, Guan Le, and Jian Du. "Survey on NoSQL Database." In: *6th International Conference on Pervasive Computing and Applications (ICPCA 2011)*. Port Elizabeth, South Africa: IEEE, 2011, pp. 363–366.

[71] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. "Comet: Batched Stream Processing for Data Intensive Distributed Computing." In: *1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 63–74. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807139.

[72] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges." In: *ACM Comput. Surv.* 51.3 (June 2018), 60:1–60:53. ISSN: 0360-0300. DOI: 10.1145/3199523. URL: http://doi.acm.org/10.1145/3199523.

[73] Thomas Heinze, Leonardo Aniello, Leonardo Querzoni, and Zbigniew Jerzak. "Cloud-based Data Stream Processing." In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. Mumbai, India: ACM, 2014, pp. 238–245. ISBN: 978-1-4503-2737-4. DOI: 10.1145/2611286.2611309. URL: http://doi.acm.org/10.1145/2611286.2611309.

[74] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. "Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems." In: *8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. Mumbai, India: ACM, 2014, pp. 13–22.

[75] Nicolas Hidalgo, Daniel Wladdimiro, and Erika Rosas. "Self-Adaptive Processing Graph with Operator Fission for Elastic Stream Processing." In: *Journal of Systems and Software* (2016). In Press. ISSN: 0164-1212.

[76] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In: *NSDI*. Vol. 11. 2011, pp. 22–22.

[77] Martin Hirzel, Scott Schneider, and Buğra Gedik. "SPL: An Extensible Language for Distributed Stream Processing." In: *ACM Transactions on Programming Languages and Systems* 39.1 (Mar. 2017), 5:1–5:39. ISSN: 0164-0925. DOI: 10.1145/3039207. URL: http://doi.acm.org/10.1145/3039207.

[78] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. "A Catalog of Stream Processing Optimizations." In: *ACM Computing Surveys* 46.4 (Mar. 2014), pp. 1–34. ISSN: 0360-0300.

[79] 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. Enterprise Distributed Object Computing Conf.* Sept. 2016, pp. 1–11.

[80] Liting Hu, Karsten Schwan, Hrishikesh Amur, and Xin Chen. "ELF: Efficient Lightweight Fast Stream Processing at Scale." In: *USENIX Annual Technical Conference*. USENIX Association. Philadelphia, USA, June 2014, pp. 25–36.

[81] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. "Quantifying the Impact of Edge Computing on Mobile Applications." In: *7th ACM SIGOPS Asia-Pacific Wksp on Systems*. APSys '16. Hong Kong, Hong Kong: ACM, 2016, 5:1–5:8. ISBN: 978-1-4503-4265-0.

[82] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. *Mobile Edge Computing: A Key Technology Towards 5G*. Whitepaper ETSI White Paper No. 11. European Telecommunications Standards Institute (ETSI), Sept. 2015.

[83] Navroop Kaur and Sandeep K. Sood. "Efficient Resource Management System Based on 4Vs of Big Data Streams." In: *Big Data Research* 9 (2017), pp. 98–106. ISSN: 2214-5796. DOI: https://doi.org/10.1016/j.bdr.2017.02.002. URL: http://www.sciencedirect.com/science/article/pii/S2214579616300909.

[84] *Kestrel*. 2016. URL: https://github.com/twitter-archive/kestrel.

[85] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. "Software-Defined Networking: A Comprehensive Survey." In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76. ISSN: 0018-9219.

[86] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. "Sketch-based Change Detection: Methods, Evaluation, and Applications." In: *3rd ACM SIGCOMM Conference on Internet Measurement*. IMC '03. Miami: ACM, 2003, pp. 234–247.

[87] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. "Twitter Heron: Stream Processing at Scale." In: *ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne: ACM, 2015, pp. 239–250. ISBN: 978-1-4503-2758-9.

[88] Geetika T. Lakshmanan, Ying Li, and Rob Strom. "Placement Strategies for Internet-Scale Data Stream Systems." In: *IEEE Internet Computing* 12.6 (Nov. 2008), pp. 50–60.

[89] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. "Model-free Control for Distributed Stream Data Processing Using Deep Reinforcement Learning." In: *Proc. VLDB Endow.* 11.6 (Feb. 2018), pp. 705–718. ISSN: 2150-8097. DOI: 10.14778/3199517.3199521. URL: https://doi.org/10.14778/3199517.3199521.

[90] Xunyun Liu, Amir Vahid Dastjerdi, and Rajkumar Buyya. "Internet of Things: Principles and Paradigms." In: ed. by Rajkumar Buyya and Amir Vahid Dastjerdi. Burlington, USA: Morgan Kaufmann, June 2016. Chap. Stream Processing in IoT: Foundations, State-of-the-art, and Future Directions.

[91] Bjorn Lohrmann, Peter Janacik, and Odej Kao. "Elastic Stream Processing with Latency Guarantees." In: *35th IEEE International Conference on Distributed Computing Systems (ICDCS)*. June 2015, pp. 399–410.

[92] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni. "Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems." In: *IEEE Transactions on Parallel and Distributed Systems* 29.3 (Mar. 2018), pp. 572–585. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2762683.

[93] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments." In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592. ISSN: 1572-9184.

[94] Long Mai, Nhu-Ngoc Dao, and Minho Park. "Real-Time Task Assignment Approach Leveraging Reinforcement Learning with Evolution Strategies for Long-Term Latency Minimization in Fog Computing." In: *Sensors* 18.9 (2018), p. 2830.

[95] F. Mehdipour, B. Javadi, and A. Mahanti. "FOG-Engine: Towards Big Data Analytics in the Fog." In: *IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Int. Conf on Pervasive Intelligence and Computing, 2nd Int. Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress*. June 2016, pp. 640–646.

[96] Gabriele Mencagli, Patrizio Dazzi, and Nicolò Tonci. "SpinStreams: A Static Optimization Tool for Data Stream Processing Applications." In: *Proceedings of the 19th International Middleware Conference*. Middleware '18. Rennes, France: ACM, 2018, pp. 66–79. ISBN: 978-1-4503-5702-9. DOI: 10.1145/3274808.3274814. URL: http://doi.acm.org/10.1145/3274808.3274814.

[97] Rob van der Meulen. *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. Gartner Website. Feb. 2017. URL: https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016/.

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

[99] J. Morales, E. Rosas, and N. Hidalgo. "Symbiosis: Sharing mobile resources for stream processing." In: *2014 IEEE Symposium on Computers and Communications (ISCC)*. Vol. Workshops. June 2014, pp. 1–6. DOI: 10.1109/ISCC.2014.6912641.

[100] Yucen Nan, Wei Li, Wei Bao, Flavia C. Delicato, Paulo F. Pires, and Albert Y. Zomaya. "A dynamic tradeoff data processing framework for delay-sensitive applications in Cloud of Things systems." In: *Journal of Parallel and Distributed Computing* 112 (2018), pp. 53–66. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2017.09.009. URL: http://www.sciencedirect.com/science/article/pii/S0743731517302630.

[101] Marco A. S. Netto, Carlos Cardonha, Renato Cunha, and Marcos Dias de Assuncao. "Evaluating Auto-scaling Strategies for Cloud Computing Environments." In: *22nd IEEE International. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. Paris, France: IEEE, Sept. 2014, pp. 187–196.

[102] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. "S4: Distributed Stream Computing Platform." In: *IEEE International Conference on Data Mining Workshops (ICDMW)*. Dec. 2010, pp. 170–177.

[103] L. Ni, J. Zhang, C. Jiang, C. Yan, and K. Yu. "Resource Allocation Strategy in Fog Computing Based on Priced Timed Petri Nets." In: *IEEE IoT Journal* PP (2017), pp. 1–1. ISSN: 2327-4662.

[104] Alexandru Iulian Orhean, Florin Pop, and Ioan Raicu. "New scheduling approach using reinforcement learning for heterogeneous distributed systems." In: *JPDC* 117 (2018), pp. 292–302.

[105] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. "MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing." In: *7th ACM International Conference on Distributed Event-based Systems*. DEBS '13. Arlington, Texas, USA: ACM, 2013, pp. 183–194. ISBN: 978-1-4503-1758-0.

[106] C. Pahl and B. Lee. "Containers and Clusters for Edge Cloud Architectures – A Technology Review." In: *3rd International Conference on Future Internet of Things and Cloud*. Aug. 2015, pp. 379–386.

[107] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. "R-Storm: Resource-Aware Scheduling in Storm." In: *16th Annual Middleware Conf.* Middleware '15. Vancouver, BC, Canada: ACM, 2015, pp. 149–161. ISBN: 978-1-4503-3618-5.

[108] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. "Network-Aware Operator Placement for Stream-Processing Systems." In: *22nd International Conference on Data Engineering (ICDE'06)*. Apr. 2006, pp. 49–49.

[109] Flavia Pisani, Jeferson Rech Brunetta, Vanderson Martins do Rosario, and Edson Borin. "Beyond the Fog: Bringing Cross-Platform Code Execution to Constrained IoT Devices." In: *29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2017)*. Campinas, Brazil, Oct. 2017, pp. 17–24.

[110] *Protocol Buffers*. 2016. URL: `https://developers.google.com/protocol-buffers/`.

[111] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. "TimeStream: Reliable Stream Computation in the Cloud." In: *8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 1–14. ISBN: 978-1-4503-1994-2. DOI: `10.1145/2465351.2465353`.

[112] *RabbitMQ*. 2016. URL: `https://www.rabbitmq.com/`.

[113] Eduard Gibert Renart, Daniel Balouek –, and Manish Parashar. *Edge Based Data-Driven Pipelines (Technical Report)*. Tech. rep. Aug. 3, 2018. URL: `http://arxiv.org/abs/1808.01353`. published.

[114] Eduard Gibert Renart, Daniel Balouek-Thomert, and Manish Parashar. "Edge Based Data-Driven Pipelines (Technical Report)." In: *CoRR* abs/1808.01353 (2018).

[115] Laura Rettig, Mourad Khayati, Philippe Cudré-Mauroux, and Michal Piórkowski. "Online Anomaly Detection Over Big Data Streams." In: *IEEE International Conference on Big Data (Big Data 2015)*. IEEE. Santa Clara, USA, Oct. 2015, pp. 1113–1122.

[116] Heejun Roh, Cheoulhoon Jung, Kyunghwi Kim, Sangheon Pack, and Wonjun Lee. "Joint flow and virtual machine placement in hybrid cloud data centers." In: *Journal of Network and Computer Applications* 85 (2017). Intelligent Systems for Heterogeneous Networks, pp. 4–13. ISSN: 1084-8045. DOI: `https://doi.org/10.1016/j.jnca.2016.12.006`. URL: `http://www.sciencedirect.com/science/article/pii/S1084804516303101`.

[117] Gabriele Russo Russo, Matteo Nardelli, Valeria Cardellini, and Francesco Lo Presti. "Multi-Level Elasticity for Wide-Area Data Streaming Systems: A Reinforcement Learning Approach." In: *Algorithms* 11.9 (2018), p. 134.

[118] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. "Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications." In: *2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1357–1369. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742790.

[119] 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, pp. 168–178.

[120] M. A. Salahuddin, A. Al-Fuqaha, and M. Guizani. "Reinforcement learning for resource provisioning in the vehicular cloud." In: *IEEE Wireless Communications* 23.4 (Aug. 2016), pp. 128–135. ISSN: 1536-1284. DOI: 10.1109/MWC.2016.7553036.

[121] S. Sarkar, S. Chatterjee, and S. Misra. "Assessment of the Suitability of Fog Computing in the Context of Internet of Things." In: *IEEE Transactions on Cloud Computing* PP.99 (2015), pp. 1–1. ISSN: 2168-7161.

[122] Kai-Uwe Sattler and Felix Beier. "Towards Elastic Stream Processing: Patterns and Infrastructure." In: *1st International Workshop on Big Dynamic Distributed Data (BD3)*. Riva del Garda, Italy, Oct. 2013, pp. 49–54.

[123] Mahadev Satyanarayanan. "Edge Computing: Vision and Challenges." In: *Edge Computing: Vision and Challenges*. Santa Clara, CA: USENIX Association, 2017.

[124] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. "Esc: Towards an Elastic Stream Computing Platform for the Cloud." In: *IEEE International Conference on Cloud Computing (CLOUD)*. July 2011, pp. 348–355.

[125] *SensorBee*. 2019. URL: http://sensorbee.io/.

[126] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. "Flux: An Adaptive Partitioning Operator for Continuous Query Systems." In: *19th International Conference on Data Engineering (ICDE 2003)*. Bangalore, India: IEEE Computer Society, Mar. 2003, pp. 25–36.

[127] Zhitao Shen, Vikram Kumaran, Michael J. Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche, and Kim Macpherson. "CSA: Streaming Engine for Internet of Things." In: *IEEE Data Engineering Bulletin* 38.4 (2015), pp. 39–50.

[128] A. Shukla and Y. Simmhan. "Toward Reliable and Rapid Elasticity for Streaming Dataflows on Clouds." In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. July 2018, pp. 1096–1106. DOI: 10.1109/ICDCS.2018.00109.

[129] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. "RIoTBench: An IoT benchmark for distributed stream processing systems." In: *Concurrency and Computation: Practice and Experience* 29.21 (2017), e4257.

[130] Alexandre da Silva Veith, Marcos Dias de Assunção, and Laurent Lefèvre. "Latency-Aware Placement of Data Stream Analytics on Edge Computing." In: *Service-Oriented Computing*. Ed. by Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu. Cham: Springer International Publishing, 2018, pp. 215–229. ISBN: 978-3-030-03596-9.

[131] H. Sun, P. Stolf, J. Pierson, and G. D. Costa. "Multi-objective Scheduling for Heterogeneous Server Systems with Machine Placement." In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. May 2014, pp. 334–343. DOI: 10.1109/CCGrid.2014.53.

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

[133] *System S*. 2008. URL: https://researcher.watson.ibm.com/researcher/view_group.php?id=2531.

[134] 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 Management (IM)*. May 2017, pp. 1222–1228.

[135] Yuzhe Tang and Bugra Gedik. "Autopipelining for Data Stream Processing." In: *IEEE Transactions on Parallel and Distributed Systems* 24.12 (Dec. 2013), pp. 2344–2354. ISSN: 1045-9219. DOI: 10.1109/TPDS.2012.333.

[136] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. "Storm@Twitter." In: *ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, USA: ACM, 2014, pp. 147–156.

[137] Radu Tudoran, Alexandru Costan, Olivier Nano, Ivo Santos, Hakan Soncu, and Gabriel Antoniu. "JetStream: Enabling high throughput live event streaming on multi-site clouds." In: *Future Generation Computer Systems* 54 (2016), pp. 274–291. ISSN: 0167-739X. DOI: http://dx.doi.org/10.1016/j.future.2015.01.016.

[138] N. Tziritas, T. Loukopoulos, S. U. Khan, C. Z. Xu, and A. Y. Zomaya. "On Improving Constrained Single and Group Operator Placement Using Evictions in Big Data Environments." In: *IEEE Transactions on Services Computing* 9.5 (Sept. 2016), pp. 818–831. ISSN: 1939–1374.

[139] Stratis D. Viglas and Jeffrey F. Naughton. "Rate-based Query Optimization for Streaming Information Sources." In: *ACM SIGMOD International Conference on Management of Data*. SIGMOD '02. Madison, Wisconsin: ACM, 2002, pp. 37–48.

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

[141] Eugene Wu, Yanlei Diao, and Shariq Rizvi. "High-performance Complex Event Processing over Streams." In: *ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, USA: ACM, 2006, pp. 407–418.

[142] Y. Wu and K. L. Tan. "ChronoStream: Elastic stateful stream computation in the cloud." In: *IEEE 31st Int. Conf. on Data Engineering*. Apr. 2015, pp. 723–734.

[143] J. Xu, Z. Chen, J. Tang, and S. Su. "T-Storm: Traffic-Aware Online Scheduling in Storm." In: *IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*. June 2014, pp. 535–544.

[144] Le Xu, Boyang Peng, and Indranil Gupta. "Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand." In: *IEEE International Conference on Cloud Engineering (IC2E 2016)* 00 (2016), pp. 22–31.

[145] Ronghua Xu, Seyed Yahya Nikouei, Yu Chen, Erik Blasch, and Alex Aved. "BlendMAS: A BLockchain-ENabled Decentralized Microservices Architecture for Smart Public Safety." In: *CoRR* abs/1902.10567 (2019). arXiv: 1902.10567. URL: http://arxiv.org/abs/1902.10567.

[146] Jian Yang, Zhihui Lu, and Jie Wu. "Smart-toy-edge-computing-oriented data exchange based on blockchain." In: *Journal of Systems Architecture* 87 (2018), pp. 36–48. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2018.05.001. URL: http://www.sciencedirect.com/science/article/pii/S1383762118300638.

[147] Y. Yang, S. Zhao, W. Zhang, Y. Chen, X. Luo, and J. Wang. "DEBTS: Delay Energy Balanced Task Scheduling in Homogeneous Fog Networks." In: *IEEE Internet of Things Journal* 5.3 (June 2018), pp. 2094–2106. ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2823000.

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

[149] A. Yousefpour, G. Ishigaki, and J. P. Jue. "Fog Computing: Towards Minimizing Delay in the Internet of Things." In: *2017 IEEE International Conference on Edge Computing (EDGE)*. June 2017, pp. 17–24. DOI: 10.1109/IEEE.EDGE.2017.12.

[150] Jan Zacho. *Unlocking Game-Changing Wireless Capabilities: Cisco and SITA help Copenhagen Airport Develop New Services for Transforming the Passenger Experience*. Customer Case Study. CISCO, 2012. URL: http://www.cisco.com/en/US/prod/collateral/wireless/c36_696714_00_copenhagen_airport_cs.pdf.

[151] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing." In: *Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 2–2.

[152] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale." In: *24th ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, USA: ACM, 2013, pp. 423–438.

[153] Xinwei Zhao, Saurabh Garg, Carlos Queiroz, and Rajkumar Buyya. "Software Architecture for Big Data and the Cloud." In: Elsevier – Morgan Kaufmann, 2017. Chap. A Taxonomy and Survey of Stream Processing Systems.

[154] J. Zheng, X. Dong, T. Zhang, J. Chen, W. Tong, and X. Yang. "MicrothingsChain: Edge Computing and Decentralized IoT Architecture Based on Blockchain for Cross-Domain Data Shareing." In: *2018 International Conference on Networking and Network Applications (NaNA)*. Oct. 2018, pp. 350–355. DOI: 10.1109/NANA.2018.8648780.

[155] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. “Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System.” In: *On the Move to Meaningful Internet Systems 2006*. Ed. by Robert Meersman and Zahir Tari. Montpellier, France: Springer Berlin Heidelberg, 2006, pp. 54–71.