EdgeDAG: a Latency-Aware Initial Operator Placement Strategy for Edge-Based Distributed Stream Processing Applications

by

Alireza MOHTADI

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS IN SOFTWARE ENGINEERING
M.A.Sc.

MONTREAL, MARCH 31, 2022

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor, Professor Julien Gascon-Samson, who helped me with the current project. He provided valuable guidance and ideas throughout the entire project. My sincere thanks go to him for all his support throughout my studies from the first semester. I am encouraged by his kindness and support to always put my best efforts into the project. I cannot express how grateful I am to be working under his supervision.

My grateful heart goes out to my family members who have helped me throughout my studies, my father Hadi, my mother Fatemeh, my sister Mahsa whose advice has always helped me throughout my life, my newborn niece Nila who always made me smile during my studies and my friend Parham. I could not reach my goals without their love and encouragement.

Additionally, I would like to thank my friends and my colleagues who have supported me through this part of my life.

EdgeDAG : une stratégie de placement initial des opérateurs tenant compte de la latence pour les applications de traitement de flux distribués en périphérie

Alireza MOHTADI

RÉSUMÉ

Au cours des dernières années, l'internet des objets (IoT) a grandi en popularité. Ils sont directement intégrés à la vie humaine. Par conséquent, les données générées par les dispositifs IoT ont augmenté. Le traitement de ces données peut fournir des informations précieuses. De plus, elles peuvent réduire la quantité de données transmises au cloud. L'objectif critique de ces applications est de fournir des résultats à faible latence. Le traitement en continu peut faciliter le traitement en offrant un modèle de programmation pour exécuter l'application en parallèle. Au cours des dernières années, le nombre d'applications IoT qui reposent sur le traitement en flux a augmenté. Ces applications traitent des flux continus de données avec un faible délai et fournissent des informations précieuses. Cependant, les dispositifs IoT sont limités en matière de bande passante et de ressources de calcul. Pour répondre aux exigences strictes en matière de latence et à la nécessité d'obtenir des résultats en temps réel, les composants du pipeline de traitement des flux peuvent être déployés directement en périphérie afin de bénéficier des ressources et des capacités que l'essaim d'appareils *edge* peut fournir. En effet, le réseau edge fournit des ressources proches de l'utilisateur et permet de réduire la latence dans les applications avec contraintes de temps. Dans cette thèse, nous proposons une nouvelle stratégie d'optimisation pour déployer les opérateurs sur les ressources appropriées à la périphérie du réseau dans le but de minimiser la latence tout en s'assurant que les contraintes des dispositifs et leurs capacités réseau sont respectées. En implémentant un nouveau simulateur, en proposant un nouveau type de fonction d'optimisation et en utilisant un algorithme méta-heuristique appelé Grey Wolf Optimizer, nous pouvons fournir une bonne correspondance entre les nœuds de périphérie et les opérateurs.

Mots-clés: traitement en continu distribué, problème de placement, informatique de périphérie, internet des objets

EdgeDAG: a Latency-Aware Initial Operator Placement Strategy for Edge-Based Distributed Stream Processing Applications

Alireza MOHTADI

ABSTRACT

Over the past few years, the Internet of Things (IoT) has become popular. IoT devices are directly integrated into human lives. As a result, the data generated by IoT devices has increased. Processing applications can provide valuable information and it can reduce the amount of data passed to the cloud. The critical goal of these applications is to provide low latency results. Stream processing can facilitate the processing by offering a programming model for running an application in parallel. In the last few years, the number of IoT applications that rely on stream processing has increased. These applications process continuous streams of data with a low delay and provide valuable information. However, IoT devices are restricted in the case of bandwidth and computational resources. To meet the stringent latency requirements and the need for real-time results, the components of the stream processing pipeline can be deployed directly onto the edge layer to benefit from the resources and capabilities that the swarm of edge devices can provide. Edge network provides resources close to the user and helps to reduce the latency in time-sensitive applications. In this thesis, we propose a new optimization strategy for deploying the operators over the suitable resources at the edge of the network with the goal of minimizing latency while ensuring that the constraints of the devices and their network capabilities are respected. By implementing a new simulator, proposing a new type of optimization function and using a meta-heuristic algorithm called Grey Wolf Optimizer, we can provide a good mapping between the edge nodes and operators.

Keywords: distributed stream processing, placement problem, edge computing, internet of things

TABLE OF CONTENTS

Page

INTRODUCTION 1

CHAPTER 1 LITERATURE REVIEW AND BACKGROUND 7
1.1 Cloud, IoT and edge Computing 7
1.1.1 Cloud computing 7
1.1.2 IoT devices 8
1.1.3 Big data 9
1.1.4 Edge computing 9
1.2 Stream processing 11
1.2.1 Operators 12
1.2.1.1 Selectivity ratio 13
1.2.1.2 Source 14
1.2.1.3 Transform 14
1.2.1.4 Filter 14
1.2.1.5 Flat map 14
1.2.1.6 Aggregate 14
1.2.1.7 Duplicate 15
1.2.1.8 Hash 15
1.2.1.9 Sink 15
1.2.2 Frameworks 16
1.2.2.1 Apache Storm 16
1.2.2.2 Apache Flink 18
1.2.2.3 Other frameworks 20
1.2.3 Batch processing 20
1.2.3.1 Apache Spark 20
1.2.4 Distributed stream processing and edge computing 21
1.3 Placement problem 21
1.3.1 Types of tasks 21
1.3.1.1 Independent tasks 22
1.3.1.2 Dependent tasks 22
1.3.2 Deployment 23
1.3.2.1 Non-DAG approaches 23
1.3.2.2 DSP in DAG-based approaches 23
1.3.3 Load balancing models 26
1.3.3.1 Static 26
1.3.3.2 Dynamic 27
1.3.4 Metrics to consider 28
1.3.4.1 CPU 28
1.3.4.2 Memory 28
1.3.4.3 Delay 28

1.3.4.4 Data rate 29
1.3.4.5 Energy 29
1.3.4.6 Cost 30
1.3.4.7 Other metrics 30
1.3.5 Optimization objectives 30
1.3.5.1 Single factor 30
1.3.5.2 Multi factor 31
1.3.6 Constraints 31
1.4 Solution generation 32
1.4.1 NP problems 32
1.4.2 NP-complete problems 32
1.4.3 NP-hard problems 32
1.5 Heuristic approach 33
1.5.1 Optimization strategies 33
1.5.1.1 Heuristics 33
1.5.1.2 Greedy 34
1.5.1.3 Meta-heuristics 34
1.5.1.4 Other strategies 34
1.6 Evaluation strategies 35
1.6.1 Simulators and simulation-based evaluation 35
1.6.2 Deployment over experimental test-beds 35

CHAPTER 2 ARCHITECTURE AND SYSTEM DESIGN 41
2.1 Design goals and hypothesis 41
2.1.1 Assumptions 41
2.2 Research problem 43
2.3 System model 44
2.4 Problem formulation 45
2.4.1 Modeling and parameters 45
2.4.1.1 Task execution time 47
2.4.1.2 Message transfer time 47
2.4.1.3 Probability ratio 49
2.4.2 Constraints 49
2.4.2.1 CPU 49
2.4.2.2 Bandwidth 50
2.4.3 Fitness function 51
2.4.4 Grey Wolf Optimization 51
2.4.4.1 Application of GWO 53
2.5 DAG simulation 54
2.5.1 Source selection 54
2.5.2 Receive event 55
2.5.3 Process event 55
2.5.4 Send event 55

2.5.5 Stop signal 56
2.5.6 Latency calculation 56
2.6 Algorithms 57
2.6.1 GWO model 57
2.6.2 Latency calculation 59

CHAPTER 3 IMPLEMENTATION 63
3.1 Programming languages and libraries 63
3.1.1 Java 63
3.1.1.1 Apache Maven 63
3.1.2 Python 64
3.1.3 Bash scripting 64
3.2 Cluster configuration 64
3.3 Random topology generation 65
3.4 Profiling 66
3.5 Log analyzer 67
3.5.1 Data rate calculations 68
3.5.2 CPU usage calculations 69
3.5.3 Message size calculations 69
3.6 Simulator 69
3.7 DAG deployments 71
3.7.1 Apache Storm custom scheduler 71

CHAPTER 4 EVALUATION 73
4.1 Testbed 73
4.2 Methodology 73
4.2.1 Randomized CPU 74
4.2.2 Randomized data rate 74
4.2.3 Randomized probability ratio 74
4.2.4 Randomized message size 74
4.2.5 Number of repeats 75
4.2.6 Meta-heuristics metrics 75
4.2.7 Randomized bandwidth 75
4.2.8 DAG latency calculations 75
4.3 Comparison against other approaches 76
4.3.1 Default Storm scheduler 76
4.3.2 Cloud only 76
4.4 Test cases 77
4.4.1 Application size 77
4.4.2 Operator dependencies 77
4.5 Deployment results 83
4.5.1 Experiment one: Base experiment 84
4.5.2 Experiment two: Decreasing the bandwidth range 84
4.5.3 Experiment three: Increasing in-rate and out-rate 87

4.5.4 Experiment four: Increasing the latency 87
4.5.5 Experiment five: Reducing the message size 87
4.5.6 Experiment six: Reducing the CPU usage 90
4.6 Simulation results 90
4.6.1 Experiment one: Base experiment 93
4.6.2 Experiment two: Decreasing the bandwidth range 93
4.6.3 Experiment three: Increasing the in-rate and out-rate 93
4.6.4 Experiment four: Increasing the latency 93
4.6.5 Experiment five: Reducing the message size 97
4.6.6 Experiment six: Reducing the CPU usage 97
4.7 Discussion 97
4.7.1 General DAG-based review 102
4.7.2 Real deployment results and simulation results comparison 104

CONCLUSION AND RECOMMENDATIONS 105

BIBLIOGRAPHY 107

LIST OF TABLES

Page

Table 2.1 Parameters 46

Table 4.1 Test case configurations (1) 78

Table 4.2 Test case configurations (2) 79

Table 4.3 GWO configurations for test cases 84

Table 4.4 Experiment results 101

Table 4.5 Results for each DAG 103

LIST OF FIGURES

Page

Figure 0.1 DAG example 4

Figure 1.1 Edge computing network 10

Figure 1.2 DAG example 12

Figure 1.3 Task patterns 13

Figure 1.4 Apache Storm graph 16

Figure 1.5 Apache Storm architecture 18

Figure 1.6 Apache Flink architecture 19

Figure 1.7 Placement problem approaches classifications 22

Figure 1.8 Feature list one (1) 37

Figure 1.9 Feature list one (2) 38

Figure 1.10 Feature list two (1) 39

Figure 1.11 Feature list two (2) 40

Figure 2.1 EdgeDAG system architecture 45

Figure 2.2 Parameters example 47

Figure 2.3 GWO hierarchy 52

Figure 2.4 DAG simulation steps 58

Figure 3.1 Random generation configuration file 66

Figure 3.2 A log example 68

Figure 3.3 Log analyzer class 68

Figure 3.4 Simulator class connections 70

Figure 3.5 Simulator configuration file 71

Figure 3.6 Simulator placement result 71

Figure 4.1 Random generated DAG one: 5 vertices, 5 edges 78
Figure 4.2 Random generated DAG two: 5 vertices, 10 edges 79
Figure 4.3 Random generated DAG three: 15 vertices, 20 edges 80
Figure 4.4 Random generated DAG four: 15 vertices, 45 edges 81
Figure 4.5 Random generated DAG five: 25 vertices, 35 edges 82
Figure 4.6 Random generated DAG six: 25 vertices, 60 edges 83
Figure 4.7 Results for base experiment (experiment one) 85
Figure 4.8 Results for decreasing the bandwidth experiment (experiment two) 86
Figure 4.9 Results for increasing the in-rate and out-rate experiment (experiment three) 88
Figure 4.10 Results for increasing latency experiment (experiment four) 89
Figure 4.11 Results for reducing the message size experiment (experiment five) 91
Figure 4.12 Results for reducing the CPU usage experiment (experiment six) 92
Figure 4.13 Results for simulator prediction for base experiment (experiment one) 94
Figure 4.14 Results for simulator prediction for lower bandwidth experiment (experiment two) 95
Figure 4.15 Results for simulator prediction for increasing the in-rate and out-rate experiment (experiment three) 96
Figure 4.16 Results for simulator prediction for higher latency experiment (experiment four) 98
Figure 4.17 Results for simulator prediction for lower message size experiment (experiment five) 99
Figure 4.18 Results for simulator prediction for lower CPU usage experiment (experiment six) 100

LIST OF ALGORITHMS

Page

Algorithm 2.1 GWO algorithm 59

Algorithm 2.2 Fitness function algorithm 60

LIST OF ABBREVIATIONS

GWO	Grey Wolf Optimization
DSP	Distributed Stream Processing
DAG	Directed Acyclic Graph
MIPS	Million Instruction Per Second
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
AI	Artificial Intelligence
QoS	Quality of Service
FIFO	First In First Out

INTRODUCTION

Internet of Things

The Internet of Things (IoT) represents the connection and communication of a large number of devices over the Internet (Ai, Peng & Zhang, 2018). The number of IoT devices has greatly increased in the recent years (Shi, Pallis & Xu, 2019). IoT devices play a key role in human lives, and they have changed the human lifestyle. The IoT has influenced industries such as smart transportation, smart healthcare and smart homes. In smart transportation, IoT devices provide services such as the time of arrival, the traffic congestion, finding the best available path, determining the arrival times of buses and metro, and so on. Smart healthcare is another important IoT domain. Some IoT services monitor the health status of persons and process the data. In case of emergency situations, IoT devices inform the healthcare providers to get the necessary support. *Google Home*, smart cameras, smart vacuums are other types of IoT devices that are used by people every day and we are considering them as a part of smart homes. All in all, IoT devices are becoming increasingly popular among people due to their variety. It is estimated that there will be 150 billion IoT devices in use by 2025 (Shi *et al.*, 2019).

Nowadays, there are millions of sensors, IoT devices, and mobile phones that produce a huge amount of data rapidly. Data created by these devices is commonly known as big data due to its large volume. Advances in technologies show that the processing of such data give us valuable information (e.g, patterns and behaviours) (Ai *et al.*, 2018).

Stream processing

Stream processing can assist IoT data processing by offering a programming model that can run applications in parallel. In stream processing, data is continuously processed, analyzed, and monitored. Instead of dealing with a batch of data and reading it from a specific file, it is possible to process data whenever it arrives. In traditional batch computing, a large volume of data (e.g., one gigabyte of data) is stored, and then the processing is performed on a single machine. The process of storing and processing data increases the time it takes to prepare the results (e.g., we might need to wait for an hour) and to respond to requests. In contrast to batch computing, stream processing stores tasks on resources and allows data to be passed between tasks. Therefore, tasks do not have to wait for all the data to arrive.

The most common way to run a stream processing application is to use cloud computing. In this type of deployment, the data generated by the IoT devices need to be transferred to the far cloud servers. After processing the data, the results are sent back to the IoT devices (consumers). An alternate model is distributed stream processing, in which the data is processed in a distributed manner. It means that a large task can be divided into small tasks and can be run on different resources. So we process the data in parallel to reduce the latency requirement of applications.

Edge computing

According to the Rajaraman (2014), the idea of cloud computing was first introduced by John McCarthy in 1961 – it took fifty years for his idea to be realized by the tech companies. Cloud computing provides the centralization feature, which reduces the costs of system administration and operations. As a result, many large data centers have been developed around the world to address the needs of companies. However, due to the advances in technologies, centralized cloud based systems cannot meet the needs of all application and scenarios.

There has been an increase in the amount of data produced at the edge of the network over the last few years (hundreds of thousands of events per second (Liu, Da Silva & Hu, 2021)). It is necessary to process the data and detect the behaviours and patterns. Because of the large amount of data produced, transferring it over the network requires a lot of bandwidth. It should be mentioned that the bandwidth on the network can be limited, and having a powerful network infrastructure requires a high budget, and it is not cost-efficient. Because of this, many companies try to store their data at the edge of the network (e.g., Amazon, IBM, Intel). Moreover, several applications require processing the data in real-time and getting the results as soon as possible (e.g., health care applications, safety alarms, self-driving cars) (Shi, Cao, Zhang, Li & Xu, 2016). Due to the centralization feature of cloud resources, they cannot offer low latency services to the end-users. The data needs to be sent to the far cloud servers, and after processing, the result should be sent back to the edge of the network (Yu *et al.*, 2018). Time-sensitive applications suffer from this characteristic of cloud resources. One possible solution to reducing the latency is to consider a new type of computing infrastructure. This infrastructure should provide the resources near the end-user. Edge computing has been proposed as a solution to this problem.

Edge computing does not have a centralized infrastructure and provides computing resources at the edge of the internet, close to the users. By doing so, the time it takes to access resources is shortened, and bandwidth usage is reduced. More precisely, when edge resources are used instead of cloud resources, the network latency can be improved. Consequently, real-time applications can benefit from better services.

Operator placement problem

We may face different challenges when we want to process, store, and transfer large amounts of data. Distributed Stream Processing systems can help in solving these challenges. It can help in managing and performing the processing of a huge amount of data. By combining edge computing and stream processing, massive amounts of data can be processed with low latency.

In such problems, the data flow and the tasks are represented as directed acyclic graphs (DAGs) that model an execution and dataflow pipeline that corresponds to the distributed application's processing. As it is shown in Figure 0.1, vertices represent executable tasks, called operators, and edges represent the flow of data between those operators . One of the significant challenges in stream processing is to provide resources for operators of applications. A key objective is to assign operators to the available resources of an edge or cloud network while ensuring that the Quality of Service (QoS) is maintained. We call this problem distributed operator placement (Dias, Assunção, Veith & Buyya, 2018). Because of the variety in features of the edge nodes, each placement can generate different distributed execution performance results (e.g., latency).

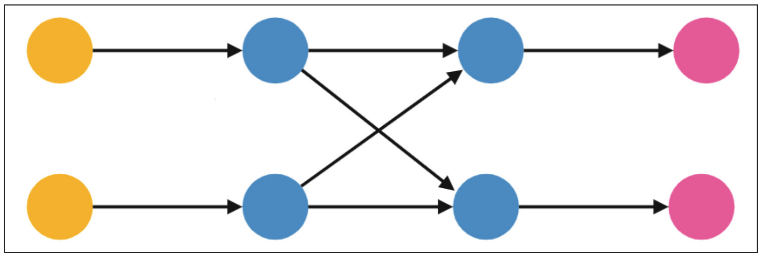


Figure 0.1 DAG example
Taken from Peng *et al.* (2019)

Contribution

In this thesis, we present a new strategy for reducing the latency in distributed stream processing applications. This methodology reduces the latency by deploying the tasks on suitable edge nodes. We implement our simulator and propose a new type of objective function which has not been focused on before. The proposed objective function considers the parallel running of operators. It also focus on the latency in all DAG paths according to the number of messages sent on each path. A meta-heuristic algorithm is used to assist in finding a suitable edge node for each task. This methodology is helpful in the event that we have a high amount of edge nodes in the edge network. It is used to find a locally-minimizing latency solution for mapping the tasks over the edge network. Moreover, this strategy is useful for very complex DAG applications. Currently, complex DAG applications might have more than 20 vertices and high number of edges (da Silva Veith, de Assuncao & Lefevre, 2021). We support the most types of stateful and stateless tasks as defined in the well-known RiotBench paper (Shukla, Chaturvedi & Simmhan (2017)). Besides, it helps to have a good view of the required latency for processing the messages. The proposed initial placement strategy is evaluated over the Apache Storm framework. We validate this methodology by comparing it with the Apache Storm default scheduler and a cloud deployment.

Thesis organization

The thesis is organized as follows: in Chapter 1, the main general topics in this thesis are described, and then we describe the related work. In Chapter 2, the architecture of the proposed system is discussed. The problem is explained in more detail, and the goals are defined. Chapter 3 focuses on the implemented tool and libraries used for building the proposed methodology. In Chapter 4, we evaluate the proposed approach in a real IoT environment.

CHAPTER 1

LITERATURE REVIEW AND BACKGROUND

In this chapter, we look at the background, proposed methods, and existing approaches in the fields of edge computing and IoT. In Section 1.1, we explain the concepts of cloud computing, IoT devices and edge computing. In Section 1.2, the concept of stream processing is reviewed. In Section 1.3, the placement problems are explained. Section 1.4 presents the different types of problems and the solutions for solving them. In Section 1.5, we review different heuristic approaches, and finally, in Section 1.6, we present the strategies for evaluating the problem.

1.1 Cloud, IoT and edge Computing

In this section, we review the general concepts of this thesis which includes cloud computing, IoT devices and edge computing.

1.1.1 Cloud computing

Cloud Computing is defined as the delivery of different services thorough the internet, which allows accessing the services from everywhere. Cloud computing has specific features that makes it unique. These features are summarized as below:

- **Resource pooling**: The computing resources are pooled to provide service to multiple customers. In other word, cloud providers can offer the services from virtually infinite pools of resources immediately.
- **On-demand**: The services are provided as soon as requested. This means that the resources provisioning is done at a time when a customer needs it. This process can be done automatically without any human interaction.
- **Measured services**: This feature refers to the services that are offered by the cloud providers to monitor the resource usage of customers and to provide billing transparency. These services enable the users to pay for the services based on their usage.

- **Broad network access**: Services are available over the network. It means that services are accessible from different locations. Users also can use the services from multiple devices.
- **Elasticity**: Services can adapt to workload changes by providing and releasing the resources automatically. They can rapidly scale up or down.

Services offered by the cloud are categorized in three groups: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

- **Infrastructure as a Service (IaaS)**: It is the most common service model offered by the cloud. It provides the infrastructure like servers, operating system and storage drives. Services are offered in different varieties and users can choose them based on their needs. The services are flexible, reliable and secure. Amazon EC2 and Amazon S3 are examples of IaaS.
- **Platform as a Service (PaaS)**: Cloud computing offers services to developers and companies to create web applications easily. This service model is useful when multiple developers want to collaborate on a single project. Openshift and Google App Engine are examples for PaaS.
- **Software as a Service (SaaS)**: These types of services enable companies to offer their software over the internet. It means that applications can be offered directly on the web browsers and there is no need to install software for the customers. Google Docs and Microsoft 365 are two popular SaaS services.

All in all, cloud computing offers services with high quality while making them accessible over the internet.

1.1.2 IoT devices

The Internet of Things includes computing resources that are wired or wirelessly connected to the network. They communicate with each other and transfer data, and exchange information. They can help to automate services and make our life easier. For example, in a smart home context, smart plugs and a *Google Home* device can be synchronized and scheduled to turn on or off the services without any human interaction.

According to the Yu *et al.* (2018), communication in IoT structures is grouped into three types.

- **Machine-To-Machine communication**: This model enables direct communication between the IoT devices themselves.
- **Machine-To-Cloud communications**: In this model, IoT devices transfer the information through the cloud resources.
- **Machine-To-Gateway communications**: IoT devices communicate with local getaways which acts as the middle layer between the IoT devices and clouds.

In this thesis, we mostly focus on Machine-To-Gateway communication.

1.1.3 Big data

The term *Big Data* has become increasingly important in recent years, especially in computer science. The increasing number of devices that generate data has given birth to the term big data. It is important to process big data to find patterns and behaviours. Raw data in and of itself often cannot be helpful because it needs to be cleaned and the relevant information extracted. So the need for prepossessing of data has increased. Many countries around the world have set aside a special budget for big data development (Oussous, Benjelloun, Lahcen & Belfkih, 2018) which shows the importance of prepossessing for big data. Generally, we have two types of processing for the large volume of data, namely (1) stream processing and (2) batch processing. We review these later in this chapter.

1.1.4 Edge computing

Edge computing is a novel paradigm that aims at bringing the resources near the end user. It involves placing the distributed resources near the clients. As shown in Figure 1.1, the structure of edge computing is divided into three groups: front-end, near-end, and far-end (Yu *et al.*, 2018). Edge nodes are located near end-users, which allows the front-end and near-end groups to provide resources at appropriate locations. In contrast to the cloud computing which offers the resources in one central location, edge computing provides distributed resources at the edge of network. They are connected and can share information. As mentioned, cloud-based architectures have some limitations (e.g., high latency, low or unreliable connectivity and bandwidth and costs), as they are centralized and far from the end users, which can make them unsuitable for some IoT contexts (e.g., traffic monitoring, fire alarm in case of emergency situation). On the other hand, IoT devices must transmit their data to these servers, which results in latency overheads for most IoT applications. As a result, real-time applications face challenges when sending data and deploying tasks on remote cloud servers. Edge computing aims at addressing these problems, by allowing for requests to be handled via edge nodes in a local network, which is made possible by the distribution of resources. In general, edge computing helps IoT devices in terms of transmission, storage and computation. By doing so, tasks are run on the edge nodes more efficiently.

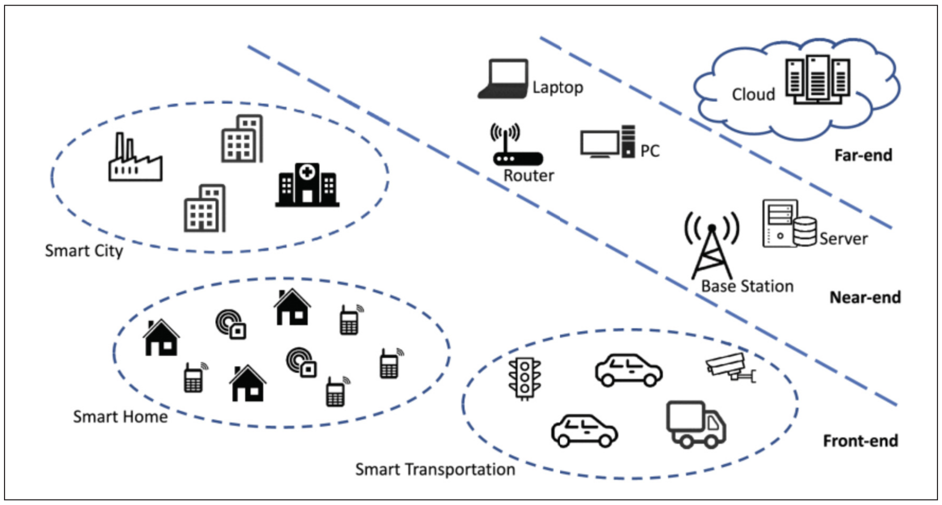


Figure 1.1 Edge computing network
Taken from Yu *et al.* (2018)

Most IoT applications are time-sensitive, and they require real-time results. They aim at processing the data in real-time. We can infer that some IoT applications have stringent latency requirements. This is where edge computing comes into play. But processing data from IoT devices is a challenge for edge computing. Managing resources is a challenge that should be addressed (Khan, Ahmed, Hakak, Yaqoob & Ahmed, 2019). Since the resources are distributed across the edge network, handling and controlling them are critical. Previous works solved resource management problems by applying scheduling algorithms. In scheduling approaches, the main objective is to map the tasks to distributed resources to have an efficient cost, time and power. As an example, in a safety alarm scenario in a nuclear station, we might need to predict potential safety problems according to the current data gathered from different devices. Informing engineers about potential safety problems is critical and it needs to be done with low latency. In this scenario, scheduling algorithms can reduce the amount of time it takes to process the data by finding the best available resources for the safety alarm applications. As a result of this thesis, we try to find an optimized scheduling solution.

1.2 Stream processing

Stream processing was first introduced in the 1960s (Stephens, 1997). Stream processing provides a programming paradigm that enables applications to benefit from a parallel execution of tasks. Stream processing is useful in many fields, such as data flows, signal processing, synchronized algorithms, and real-time applications. In such domains, stream processing allows multiple computational nodes to be run at the same time. In this thesis, we are focusing on stream processing for real-time applications.

The streaming process can be performed over the cloud on a single resource. However, there is another form of stream processing called Distributed Stream Processing (DSP). In the distributed version, each task is run on a specific resource, and we take advantage of different resources that are separate from each other. Typically, stream processing systems allow users to compose distributed applications using data flow graph abstractions. The applications are decoupled into several processing steps (operators) that are interconnected, which allows for modelling a processing pipeline as a directed acyclic graph (DAG).

1.2.1 Operators

Figure 1.2 illustrates a group of operators that are connected and that are able to run simultaneously. These operators transfer the data in an infinite sequence, which is referred to as a *stream*. In fact, the operators represent the tasks that can be run on the resources. These operators process the incoming data stream as it arrives (Pfandzelter & Bermbach, 2019). In stream processing, operators are connected through network links.

Stream processing applications usually contain three types of operators: source, transformation and sink. Source is the starting part of a graph, and model the data inputs (e.g., data from temperature sensors or smoke detectors). Data from the source operators is then sent towards downstream operators. On the opposite side, the sink operators model the last processing step(s) in the graph. They receive the data from upstream operators. At the end, the data is received by the last operators, which are called sinks. Other operators are known as transformation operators, and they perform any type of processing. Graphs are directional because data passes through operators in a particular direction. According to the data-flow graph, data cannot flow twice to a given operator, so each operator only has access to data once. As a result, the flow graph does not contain any loop, and is modelled as a directed acyclic graph (DAG).

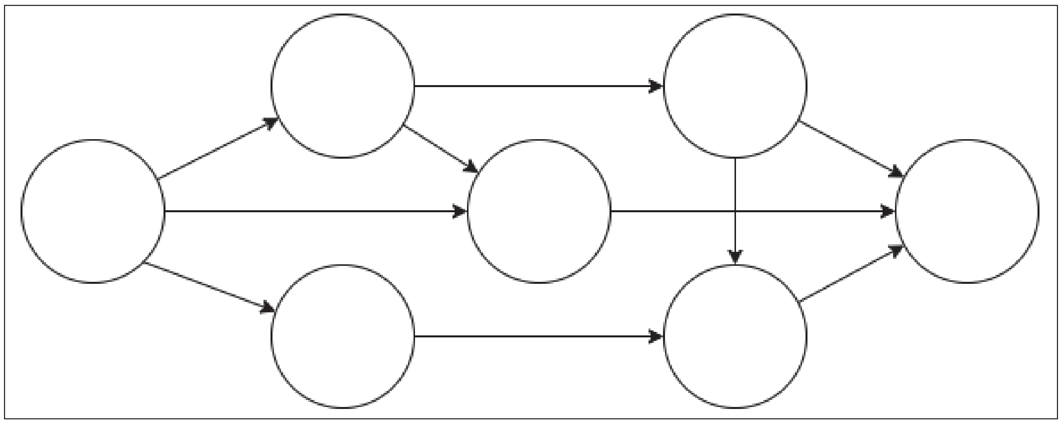


Figure 1.2 Circles shows the tasks and the links are the connections

Every operator in stream processing has a specific semantic. Stream processing is used for a wide variety of applications, all of which follow different patterns depending on their purpose. Riotbench (Shukla *et al.*, 2017) reviews these patterns in applications and categorizes the operators based on their semantics. Figure 1.3 shows a summary of different types of patterns in DSPs (we briefly present them in the next sections). Data is received by the operators, processed, and then passed on to the next operator.

1.2.1.1 Selectivity ratio

The term incoming data rate for an operator refers to the number of messages that need to be received by the operator to enable its execution. A similar definition is used for the outgoing data rate, which means that the operator generates a specific number of messages when it executes. The division of the incoming by the outgoing data rate is called the selectivity ratio. The selectivity ration is modelled as σ = inputrate : outputrate. As an example $\sigma = 3 : 2$ shows that three messages are required to generate two messages. In Figure 1.3, the output and input data rates are shown at the top of each semantic.

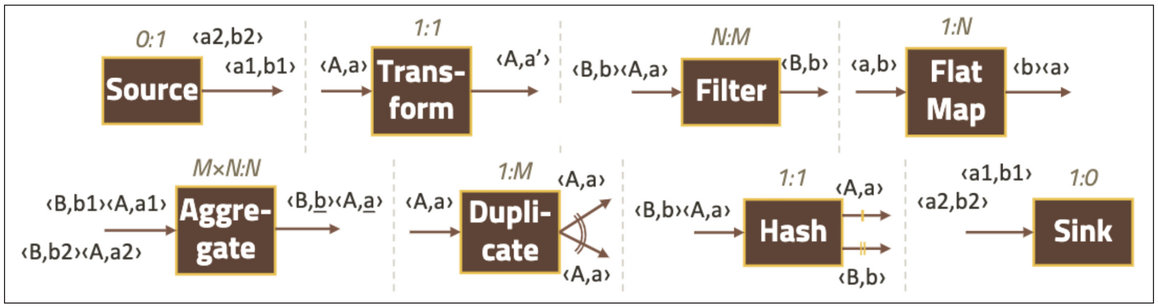


Figure 1.3 Task patterns in distributed stream processing applications
Taken from Shukla *et al.* (2017)

1.2.1.2 Source

A source acts as a starter task – it only pushes the data into the data flow, and it does not do any processing. It defines the rate at which data is transmitted to other operators, which is the data rate in the DAG.

1.2.1.3 Transform

A transform operator attempts to change the type of an incoming message and send it as another type. It models a transformation in the message (e.g., rounding a decimal number). For every message it receives, it sends one message.

1.2.1.4 Filter

A filter operator receives multiple messages and selects some of them. It allows certain messages to proceed and to be passed. For example, it excludes values outside a certain range. The number of messages it generates can differ from the number of messages it receives. In a filter semantic, the number of output messages is always less than the number of input messages.

1.2.1.5 Flat map

The next semantic is the flat map which splits the contents of one message into smaller messages. For example, an operator might receive grouped values of temperature, humidity and pressure, and would then split them into three messages and sends them separately. As a result, it takes one message and divides it into smaller messages.

1.2.1.6 Aggregate

In order to aggregate values, the aggregate operator waits for a certain number of messages or a certain period of time. By aggregating the messages received, it generates a specific number of output messages. For example, an operator can wait to receive ten messages and then sends sums of them.

1.2.1.7 Duplicate

The duplicate operator receives a message and sends it to all the downstream operators. Therefore, the output rate of duplicate operators depends on the number of downstream operators that they have.

1.2.1.8 Hash

The hash operator receives one message and sends it to a specific downstream operator. There are different options for choosing the downstream operator(s). For instance, it is possible to have a hash function that decides the destination of a message (hash semantic). Alternatively, messages can be sent in a specific order (e.g., round-robin).

1.2.1.9 Sink

Lastly, we have the sink semantic, which receives all incoming data from one or more operators. It is the final operator, and it does not do any processing. Some tasks are done by the sink operators; for example, sending the data to the cloud or storing the data.

It is worth mentioning that all the defined semantics are the most popular ones among applications. There might be other types of semantics that can be similar to the mentioned semantics. For example, a join semantic can be used to join multiple messages and generate a combined output message.

1.2.2 Frameworks

In order to solve the problem of distributed stream processing, several frameworks have been proposed. Dias *et al.* (2018) present several of the frameworks which are used for stream processing applications. We review some of the main ones below.

1.2.2.1 Apache Storm

Apache Storm is the base framework for DSP applications, and it forms the basis of all the other frameworks. Apache Storm is suitable for the time-sensitive application. Most of the new frameworks are built according to the Apache Storm framework (e.g., Apache Flink is implemented by improving different parts of Apache storm). Apache Storm is more configurable and flexible than the other frameworks and provides more specific features for the developers (e.g., scheduling configuration).

Apache Storm is an operator-graph framework. In this type of framework, tasks are defined as operators and are connected to each other to form a graph. The connection models the flow of data among the operators. The graph in Apache Storm is called a *topology*. The topology is run continuously until the user stops the application.

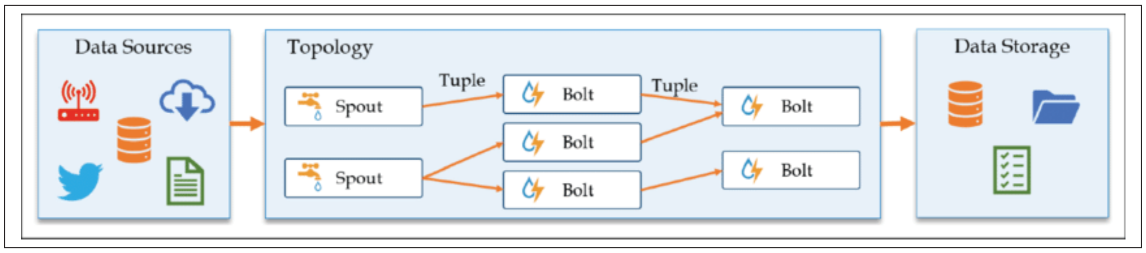


Figure 1.4 Data stream processing in Apache Storm
Taken from Kim *et al.* (2019)

In Apache Storm, the stream of data is decoupled into a series of data items that are referred to as *tuples*. A tuple is a simple unit of data that can be processed by the nodes (operators).

Apache Storm contains different task nodes that are either *bolts* or *spouts*. Both spouts and bolts are operators. A spout is the starting node that feeds an unlimited amount of tuples into the topology, and a bolt is a specific task in the topology. Bolts emit data streams and pass them to other bolts, or they act as the final operator (i.e., sinks).

In Storm, there are three types of objects: executors, tasks, and workers. Executors are the smallest object in the Apache storm. Tasks represent the execution entity of each operator. An executor runs one or more tasks. Additionally, we have workers which host Java virtual machines (JVMs) and are running on the edge nodes.

Apache Storm works according to a controller-worker execution model. The controller node is called the *nimbus*, and the other nodes are the worker nodes. The nimbus (i.e., the central controller) manages all of the worker nodes. It is responsible for health check status, and task assignment. The nimbus also has a scheduler that decides on the deployment of tasks over the workers. In Storm, the deployment is done through another component called Zookeeper. The nimbus directly communicates with Zookeeper to obtain the health status of the worker nodes. By doing so, the scheduler in the nimbus decides to redeploy the tasks in case of failure.

Zookeeper acts as a middleware between the nimbus and worker nodes. Each worker node runs multiple worker processes that can be assigned to one cluster node. The worker node runs the worker processes according to the number of available worker slots. Since we may have multiple worker processes, we need a node to be able to manage all of these workers over one resource. Therefore, the worker node has another component which is called the supervisor. The supervisor sends the status of the worker processes to Zookeeper. Supervisors also assign the tasks, which are defined by the nimbus, to the workers. Each worker process includes the JVM and executors (Dias *et al.*, 2018). In figure 1.5, the architecture of Apache Storm is presented. In this thesis, we have decided to use Apache Storm because it is more configurable than other frameworks.

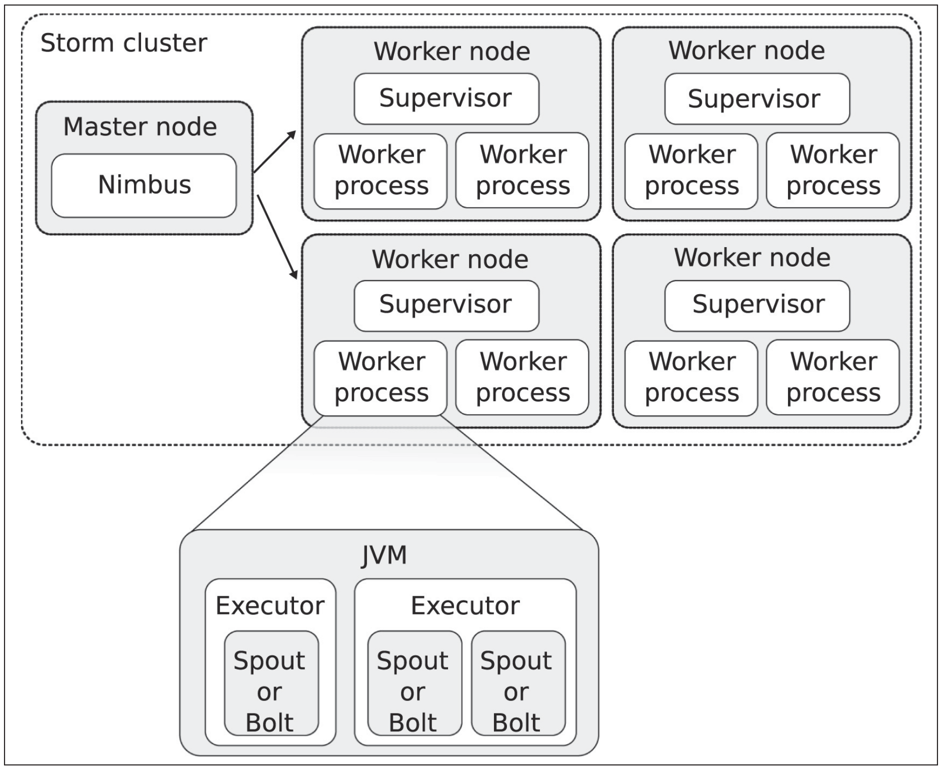


Figure 1.5 Apache Storm architecture
Taken from Dias *et al.* (2018)

We note that stateful tasks are not supported by Apache Storm as it is stateless. However, there is a framework called Trident that supports Apache Storm to run stateful operators. Trident is available in Apache Storm, and it can be used easily by users.

1.2.2.2 Apache Flink

Apache Flink is another framework that is implemented over Apache Storm. It supports both stream and batch processing. It is a DAG-based framework that models operators connected to each other and the data streams flow among them. Apache Flink has three types of operators: source operators, transformation operators, and sink operators. Source operators are the starting operators, transformation operators do the processing, and sink operators are the final operators. In Flink, there might be multiple instances of an operator, and they are called subtasks. The subtasks are executed independently in separate threads and may be deployed on different resources. Similarly, streams are broken into one or more stream partitions (Carbone *et al.*, 2015).

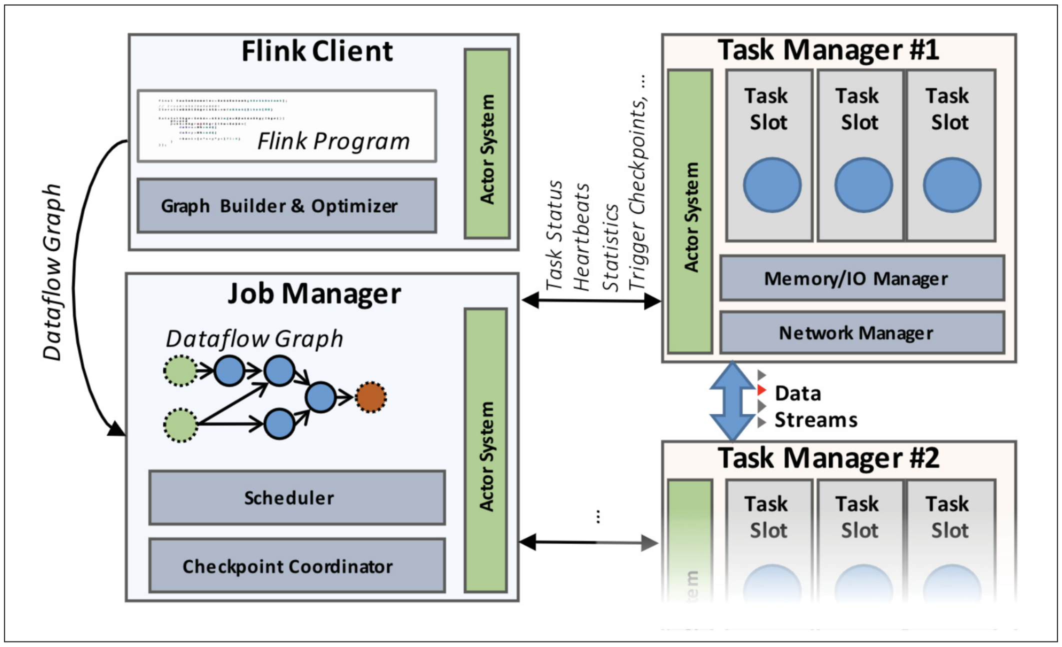


Figure 1.6 Apache Flink architecture
Taken from Carbone *et al.* (2015)

As it is observable in Figure 1.6, Flink contains three main sections. Users submit an application through the Flink client. The Flink client builds the graph, and after optimizing it, sends the graph to the job manager. The job manager is in charge of coordinating checkpoints, tracking the task status and heartbeats. It is also responsible for scheduling tasks and data recovery in case of failure. In Flink, we have another component called the task manager that runs the data processing. It is also known as the worker node. It runs one or multiple operators and sends the status to the job manager. The task manager also sends the streams to other task managers and maintains the network connections with the other task managers. Furthermore, Flink uses Java Virtual Machines and slot concepts to allow operators to be run in multiple threads.

1.2.2.3 Other frameworks

There are multiple other industrial stream frameworks such as Twitter Heron, Apache S4, Apache Samza and Spark Streaming. We do not consider them in our thesis, as some of them are implemented for batch processing and then redeveloped to be compatible with stream processing. Others are difficult to set up and require specific changes to their code to make them work. Apart from industrial frameworks, some researchers also propose their own stream processing frameworks, but they are not popular in the stream processing communities. These researchers define their frameworks according to their requirements and assumptions. We consider them as out-of-scope in this thesis.

1.2.3 Batch processing

In contrast to stream processing, batch processing involves a large amount of data being processed at once. These frameworks typically need to wait for a specific amount of data/time before processing starts. For example, the pipeline can be run every 24 hours or for every 1 Gigabyte of data. The processing of data then starts in a distributed manner. Apache Spark is one of the frameworks which provides an infrastructure for batch processing. In batch processing, we are faced with a huge amount of data to process without considering the latency. In such frameworks, the latency is less important, as the emphasis is put more on the throughput performance of the pipeline, unlike in stream processing, in which the data is processed as soon as it comes, with the goal of reducing the latency of the applications.

1.2.3.1 Apache Spark

Apache Spark is a framework that was first offered for cluster computing by Zaharia *et al.* (2012). It has been developed to perform better than Hadoop Map-reduce, and it is known as the upgraded version of Hadoop. Spark offers an abstraction called Resilient Distributed Datasets (RDDs) to support in-memory computing. It enables the clients to query the data much faster. As explained in Shoro & Soomro (2015), Apache Spark should be avoided in two specific scenarios: in the case that the latency is critical, and in the case that there is a shortage of memory.

1.2.4 Distributed stream processing and edge computing

Edge computing offers plenty of distributed devices near the end-user. These devices can provide resources such as CPU, memory and storage. On the other hand, DSP applications contain small connected tasks which are highly distributed. Therefore, edge computing provides a compelling architectural option for executing tasks in distributed stream processing applications. By the combination of edge computing and DSP, it is possible to meet the requirements of most applications.

1.3 Placement problem

In a distributed computing context, a placement problem consists of deploying tasks over the available resources. The goal is to find the best resources for the the tasks. We review different aspects of the placement problems in this section. Figure 1.7 shows how we classify the approaches in placement problems. We divide the placement problems into two groups: DAG and non-DAG. Each class contains specific types of tasks, and we explain them in the following sections. It is worth mentioning that most of the DSP applications are defined as DAGs.

1.3.1 Types of tasks

Types of the tasks in placement problems are highly dependent on the applications' nature. Generally, applications are divided into two groups: dependent tasks and independent tasks. In a DAG-based placement, we have dependent tasks, whereas, in a non-DAG placement, we have independent tasks that run independently.

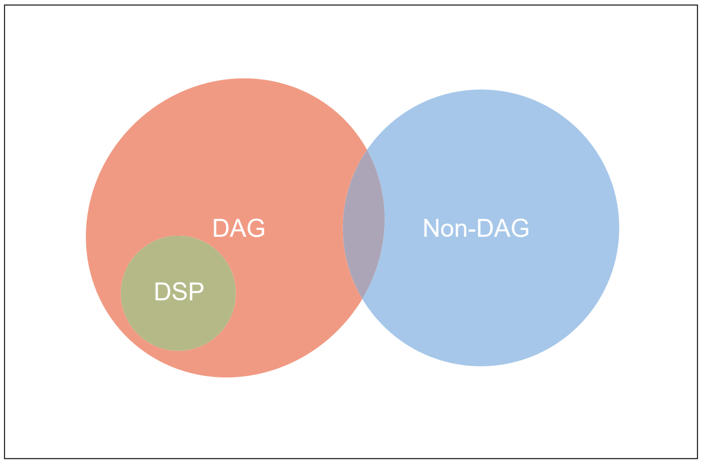


Figure 1.7 Placement problem approaches classifications

1.3.1.1 Independent tasks

Some applications are defined as a series of independent tasks that need to be run on the available resources. In these cases, the literature proposes approaches to scheduling a set of independent tasks on the best available resources to get the best latency. Liu, Mao, Zhang & Letaief (2016) work on offloading the independent tasks form edge layer to the servers. Tasks does not have any relation and the proposed methodology decides about running them locally or offloading them. Scoca, Aral, Brandic, De Nicola & Uriarte (2018) also propose a service scheduling algorithm for edge networks. It focuses on latency-sensitive services which are completely independent. Each service is run separately and there is no dependency among them.

1.3.1.2 Dependent tasks

Directed Acyclic Graphs (DAG) are used primarily for data-driven problems and dependent tasks. DAG-based models are beneficial when tasks are dependent on each other. The vertices represent the tasks, and the dependencies among them are the edges. Khare *et al.* (2019) propose an approach for deploying the tasks onto the available edge resources for DAG-based problems using machine learning algorithms. Cai, Kuang, Hu, Song & Lü (2018) also consider a DAG-based structure. The propose a methodology which alleviates the average response time of complex event processing services in DAGs. In this thesis, we consider DAG-based approaches, as several IoT applications consist of dependent tasks that are highly distributed.

1.3.2 Deployment

Deployment refers to placing the operators onto the specific resources infrastructure. In this section, we review the deployments in the non-DAG approaches and DAG approaches separately.

1.3.2.1 Non-DAG approaches

In this type of application, services are deployed on the distributed resources. We do not have DAGs in this type of application, and they are known as task scheduling approaches.

Lin, Li, Xiong & Lin (2018) work on two challenges for offloading tasks on edge clouds. Due to the existence of distributed cloudlets in edge networks, it is very challenging to run scheduling and workload balancing algorithms without using a centralized server. So they consider the combination of edge and cloud resources for their deployments.

Madej, Wang, Athanasopoulos, Ranjan & Varghese (2020) present a hybrid approach that offloads tasks from a cloud server to edge nodes. This hybrid strategy is based on fairness and considers it for cloud servers. In this paper, fairness is applied to two technical points of view. Firstly, all cloud servers have the same chance to offload their tasks to edge resources. Secondly, tasks that come from clients have a priority factor. By doing so, some tasks become more important than others. These methodologies directly work on offloading the tasks from the cloud to the edge of the network.

1.3.2.2 DSP in DAG-based approaches

In this section, deployments are done for the DAG of tasks. Most papers work on placement problem in a cloud, fog, edge or a combination of these three layers. We review them in the following sections. In the following classification, we consider all papers that combine edge and cloud as edge-cloud-based. Edge-based approaches only consider edge resources, and do not consider cloud resources. We work on edge-based approaches in this thesis as well, as edge-based technologies are important for latency optimization.

1.3.2.2.1 Edge-cloud-based

In Amarasinghe, De Assuncao, Harwood & Karunasekera (2018), the authors propose a model that aims to minimize the end-to-end latency. In this approach, the operators are placed on the cloud or at the edge. They use the R-Storm framework to formulate a constraint satisfaction problem for evaluating the availability of resources for tasks. Then they decide about deploying it on the cloud or at the edge. The key point is to minimize the latency for the critical path of the topology owing to interactions among operators in the path. They propose a heuristic algorithm for minimizing the sum of individual latency (computational latency and network latency).

da Silva Veith, de Assuncao & Lefevre (2018) use two strategies for deploying the tasks. The first one is Response Time Rate (RTR) which places operators based on end-to-end latency of paths. The second method is Response Time Rate with Region Patterns (RTR+RP), which splits dataflows according to region patterns. Then they distribute the tasks between the cloud and edge nodes based on the two proposed approaches.

Elgamal, Sandur, Nguyen, Nahrstedt & Agha (2018) propose a model to estimate the execution time of the dataflow graph. This model focuses on computation, communication, and queueing delay. It considers concurrency for running operators. Afterward, they propose DROPLET to address the operator placement problems in log-linear time. DROPLET is a scalable dynamic programming algorithm that places operators among edge and cloud resources – it decides if an operator should be deployed in the cloud or should be executed on the edge nodes.

Gedeon, Stein, Wang & Muehlhaeuser (2018) offer a model that works with the edge, fog and cloud. It uses two heuristic approaches, which decrease makespan and network latency. The first heuristic focuses on the placement restriction of a particular operator on some nodes. It forces operators to run on the nodes that provide better resources in terms of location, cost and network connection. The second heuristic focuses on the co-location of operators on the same node. In this thesis, we do not consider the co-location of the operators.

Renart *et al.* (2019) implement a framework called R-Plusar that presents a method to offload operators from the cloud to the edge. They integrate a splitter service and an operator placement scheme to build the R-Plusar software stack. The splitter service specifies the separation of data flows and locates operators across edge and cloud resources. The operator placement strategy strives to minimize a cost, which is formulated based on the latency, data transfer, and message transfer costs between the edge and the cloud.

Janßen, Verbitskiy, Renner & Thamsen (2018) also propose a method for stream processing of the tasks. It proposes a scheduling algorithm that deploys the operators over the geo-distributed heterogeneous resources close to the users. It uses a meta-heuristic algorithm and an optimization function for finding the best placement of the operators. However, they do not test their methodology on highly complex applications. Moreover, their optimization function consists of multiple factors such as the response time, resource fitting and bandwidth usage. The semantics used in the evaluations are limited in this paper.

1.3.2.2.2 Edge-based

Khare *et al.* (2019) propose a machine learning model to predict the latency of each linear path. This model is data-driven and uses DAG-based execution semantics and data rate to predict the latency for each path. It is designed to place the operators over the best available edge nodes. Nardelli, Cardellini, Grassi & Presti (2019) also proposes two heuristic approaches that are applicable only for the edge network.

Cai *et al.* (2018) propose two models to capture response time. The basic idea of their algorithm is to predict the end-to-end delay for all paths, and deploy operators on edge nodes with the minimum response time. They reduce the total response time of services by 38% and 45% on average. Generally, they focus on the load balancing of different edge nodes based on their response times.

Peng *et al.* (2019) is another related paper that works on the deployment of operators on edge resources. Apart from the binary genetic algorithm that finds the primary solution, it provides a bottleneck analysis algorithm. It tries to find the longest queue length of the DAG. Then it deploys the bottleneck to the best available resources that provide better processing power.

In this thesis, we use a meta-heuristic algorithm and we target minimization of the latency in all the paths of the topology. We also try to show the effects of a meta-heuristic algorithm on complex applications that include different semantics.

1.3.2.2.3 Fog-based

Unlike other papers that are working on DSPs at the edge of the network, Hiessl, Karagiannis, Hochreiner, Schulte & Nardelli (2019) consider a fog computing environment. They formulate an optimization problem for the placement of operators based on fog computing. Hiessl *et al.* (2019) consider different QoS for the fog environment and provides a plugin for DSP frameworks to apply the optimized solution. The proposed plugin runs the optimization periodically during runtime.

1.3.3 Load balancing models

In this section, different general types of approaches for scheduling the tasks are reviewed. There are different types of task scheduling in the edge taxonomy based on Varshney & Simmhan (2020). They are categorized into two types: static and dynamic deployments. We review them in the following sections.

1.3.3.1 Static

The act of static deployment is known as initial placement. It implies that the mapping is done before starting the application. It uses all of the information from all metrics in the problem to determine the best mapping. A critical step in operator placement is the first deployment. A good placement reduces the likelihood of running into problems during execution. Many approaches try to find the best initial mapping[1]. da Silva Veith *et al.* (2021) work on an initial placement of operators over the edge and cloud resources. It means that changes in the deployment environment would not be considered. Cai *et al.* (2018) define an approach and tests their approach on a static environment, but it can be extended to a dynamic environment. In the same way, our strategy aims at statically scheduling the tasks, but it can also be extended to the dynamic scheduling of operators in future work.

1.3.3.2 Dynamic

Different factors may change over time in the edge structure. Factors like the available resources, available bandwidth and available power affect the performance of applications while they are running. Therefore, we need to gather dynamic information during the execution of the programs. In this case, as the conditions change, the operators might need to be redeployed over the resources. A dynamic approach reschedules the mapping of operators over the resources with the updated information from edge nodes. There are different strategies for dynamic placement. Most of the approaches stop the current running of the program, deploy the program on the new resources and then continue the execution. This increases the latency of applications since we have a complete stop in the application. Liu *et al.* (2016) try to schedule the incoming service in each time slot which can be considered somehow a dynamic approach. Cao, Chen, Jiang & Hu (2018) use two methods of reordering the tasks and rearranging the edge nodes in their algorithm that uses the updated information about the tasks and edge nodes. By doing so, they have a good view of the problem, and they can offer better placement. Madej *et al.* (2020) is also working on dynamic scheduling of tasks which decides about the incoming task requests based on the availability of current resources to deploy them from cloud servers to the edge nodes. So for each task, it first checks the availabilities and then decide on the deployment. Renart *et al.* (2019) define a new framework that splits the applications over the cloud and the edge dynamically. In this thesis, the dynamic scheduling of resources has not been considered due to its complexity in a real deployment. We would be interested in exploring this area as future work.

[1] More information is available in Figure 1.8 and Figure 1.9.

1.3.4 Metrics to consider

Due to the complexity of deploying applications on the edge network, different factors affect the DSP problems. Different types of metrics must be considered as input parameters to optimize the placement. We review these metrics in the following sections:

1.3.4.1 CPU

Due to the resources limitations of the IoT/edge devices, CPU usage is an important factor in the deployment of operators over the edge resources. The CPU is the main factor that affects the execution time of programs. It also models the availability of the computing resources. The CPU usage is measured by its clock cycle, the number of cores, or Million instructions per second (MIPS). The MIPS value indicates the number of instructions can be executed in a second by the CPU. A specific amount of CPU cycles are also needed to run each program. By considering all these factors, it is possible to measure the required execution time for the applications. Renart *et al.* (2019) use the concept of MIPS to model the CPU usage of tasks. On the other hand, Amarasinghe *et al.* (2018) use the clock speed of the CPU in Hz to model the capacity of the CPU and the number of CPU cycles to show the required CPU for running the tasks.

1.3.4.2 Memory

Memory is another factor that is considered in placement problems. It is also known as RAM. It mostly refers to the number of required bytes to be able to load and run an application. Renart *et al.* (2019) is one of the papers which considers memory in their problem formulation. In this thesis, we do not consider the memory since for most applications it is not a limiting factor.

1.3.4.3 Delay

The delay is one of the most important factors in placement problems that attempt to minimize the latency. It needs to be calculated precisely to accurately model the problem. To be more precise, the delay is used differently. The notion of delay can refer to several notions, such as the transmission delay, the propagation delay, the queuing delay, the network delay, the execution time, the processing delay, the response time and the end-to-end latency – they are all time-based factors. Wei, Wei, Li, Zhuang & Yue (2018) propose a latency guaranty task allocation methodology. It allocates tasks considering the end-to-end latency (transfer latency) and processing latency in DAG-based approaches using the proposed greedy approach. The end-to-end latency refers to the amount of time between each pair of operators. Cai *et al.* (2018) also consider the queuing delay; i.e., it shows the time that a task needed to be in the queue to be processed. The transmission delay is another factor that is measured based on the provided bandwidth between two edge nodes and the size of the messages. The propagation delay is the time it takes for an edge node to prepare the messages. Due to the complexity that queueing delays add to the evaluation section, we do not consider it in this thesis. The propagation delay is also not taken into account, since the value is close to zero. Other time-based metrics are used in the methodology.

1.3.4.4 Data rate

For DAG-based problems, it is essential to know at what rate the data is being fed into the DAG. It directly affects the other metrics, and it is always recommended to keep track of data rates. The data rate is also considered for each edge node. Generally, two types of data rates are mostly taken into account (da Silva Veith *et al.*, 2018): the incoming data rate shows the rate at which the data goes into an edge node, and the outgoing data rate represents the rate at which the data is generated by the tasks deployed on the edge nodes.

1.3.4.5 Energy

Energy is one of the factors which is critical in IoT devices. Cao *et al.* (2018) propose a scheduling algorithm and considers the energy in their model. Amarasinghe *et al.* (2018) also propose a methodology to minimize the energy usage in edge computing applications. Since some IoT devices have limited energy storage, it is important to deploy the operators over the edge nodes in a way that energy reduces.

1.3.4.6 Cost

Operator placement problems are mostly used for time-sensitive applications when the cost is not so important. Peng *et al.* (2019) propose an approach that targets the cost in placement problems. In most cases, however, it is assumed that the edge resources are available to be used, and that the applications are to be deployed on the unused resources to reduce the latency. Therefore, in the most of the current approaches, cost is not a main factor.

1.3.4.7 Other metrics

There are plenty of metrics such as the message size and the bandwidth which are used by some other papers for the operator placement problems. They are not discussed in this section because their effects are presented in other metrics such as network delay.

1.3.5 Optimization objectives

There are different objectives in DSP applications, and they are defined according to the problem requirements. We divide the papers into two groups of single factor and multi-factor objectives. We explain the main objectives as in the following sections.

1.3.5.1 Single factor

The proposed methodologies focus on one specific optimization goal. Due to the demands for processing data in real-time, most edge-based applications are latency-sensitive. da Silva Veith *et al.* (2021) propose scalable approaches that aim at minimizing the end-to-end latency in complex and large DSP applications. It splits the graph and distributes the operators among the edge and cloud resources. Similarly, Elgamal *et al.* (2018) propose a dynamic approach that tries to share the tasks between the cloud and the edge while reducing the end-to-end completion times. Eskandari, Mair, Huang & Eyers (2021) propose a scheduler for DSP applications that aims at reducing the task size of the graph. In other words, it partitions a DAG into sub-graphs according to the network traffic between nodes. By doing so, a DAG will be grouped into a small amount of subgraphs. We also target the latency in our approach and we try to minimize it.

1.3.5.2 Multi factor

Some other approaches combine different goals and create a formula as the goal. These approaches define an objective function by combining different metrics and then attempt to assign tasks based on that optimization function. For example, Cardellini, Grassi, Lo Presti & Nardelli (2017) work on an aggregated formula that considers the response time, availability, cost and network. It proposes a scheduler called S-ODPR for the Apache Storm framework, which works based on the mentioned metrics. It tries to re-balance the edge nodes and then reassigns the operators. It also considers the replication status based on the proposed algorithms. Then, in the evaluation section, they display the impacts of deployment on the different metrics. Similarly, Hiessl *et al.* (2019) introduce an objective function that represents the budget, processing duration and resource metrics in DSP applications deployed in a fog environment.

1.3.6 Constraints

In this section, we review different constrains that are considered in similar approaches. Constraints are the characteristics that influence the optimization problem. In DSP, most of the constraints are chosen according to the main factors which are discussed in Section 1.3.4. For example, in da Silva Veith *et al.* (2021), memory, CPU and bandwidth are considered as the constraints for finding a good mapping for the initial placement of operators over the edge nodes. Cardellini, Grassi, Lo Presti & Nardelli (2016) considers bandwidth as a constraint and propose a new scheduler for the Apache Storm framework. It also considers multiple QoS metrics (CPU, memory, and latency) as the objective function. In general, each paper considers its own set of constraints according to the assumptions that it states. In this thesis, we consider bandwidth as the main constraint of the placement problem.

1.4 Solution generation

In this section, we discuss how it is possible to find an optimal solution for the proposed problem. By understanding the concept of problem complexity, we can identify the best way to solve the problem. We review the definitions of NP problems, NP-complete and NP-hard.

1.4.1 NP problems

Sometimes, it may be difficult to find a solution to a problem. But it is possible to solve and verify it using non-deterministic algorithms in polynomial time. These types of problems are referred to as NP problems. An algorithm is solvable in polynomial time if it is bounded by a polynomial expression. In non-deterministic algorithms, we might have a different output for the same input of the problem. Moreover, we have the P problems that are solvable using a deterministic algorithm in polynomial time. Generally, P is the subset of NP since NP can solve the P problems. But we might have another type of problem in NP which is not P. In this case, finding a solution requires a very hard search.

1.4.2 NP-complete problems

NP-complete problems are a subset of NP problems, and they are as hard as NP problems. In these types of problems, any other NP problem like P is reducible to them in polynomial time. It means that we can solve a problem with NP complexity if we know how to solve the problem with NP-complete complexity. A non-deterministic machine can solve such problems.

1.4.3 NP-hard problems

A problem is NP-hard and solvable if there is an NP-complete problem that is reducible to the NP-hard problem in polynomial time. NP-hard problems are as hard as NP-complete problems, and they are not subset of NP-problems.

Prior approaches like da Silva Veith *et al.* (2021) have shown that the operator placement problem on an edge network is NP-hard (according to the Benoit, Dobrila, Nicod & Philippe (2013)). It shows that we need to solve a NP-hard problem in the operator placement problem.

1.5 Heuristic approach

Finding a solution to an NP-hard problem becomes increasingly difficult as the search space is expanded. As a result, polynomial algorithms become inefficient. In this case, heuristic algorithms can be very helpful, as they provide us with a solution that approximates an optimal solution in a reasonable amount of time. As we discussed in 1.5.1.3, meta-heuristics are helpful since they prevent the heuristics from getting stuck in the local minima solution. They have a randomness factor that helps to find the solution in the global search space.

1.5.1 Optimization strategies

Different strategies have been considered for solving the placement problems. Some papers define their own algorithm to achieve their goals in the system. They try to use an algorithm to optimize their goals. Some of these optimization algorithms uses specific heuristics to find a better solution. In optimization algorithms, an optimization function is defined that is used to estimate how good a solution is.

1.5.1.1 Heuristics

As we mentioned, heuristics are approaches that are designed to solve the problem in a quicker way while the traditional solving methods take a lot of time to find a result. Instead, in heuristic approaches, an approximate result is shown and finding the best solution is not considered. Most of the time a heuristic function is presented to rate the solution in the space problem.

Gedeon *et al.* (2018) define three types of heuristics (Restriction, Pinning and Co-location) that help to reduce the solving time in placement problems. Nardelli *et al.* (2019) define a multi-objective optimization function and tries to solve the problem with different heuristic algorithms, and then divide them into two groups of model-based and model-free. The model-based heuristic is about Optimal Distributed Stream processing. On the contrary, model-free heuristics are basic meta-heuristic algorithms like local search, greedy first, etc. They integrate two heuristic groups to find a suitable heuristic that includes most factors of QoS.

1.5.1.2 Greedy

Heuristics have different types and one of the most used type in recent approaches is greedy algorithms. In a greedy algorithm, we choose the best available option at each stage that we want to find the solution. Greedy algorithm helps in finding the local optimum solution. By using greedy algorithms, we are confined to local exploration, while meta-heuristic algorithms allow us to explore outside of the local zone. Yin, Li, Li & Li (2019) offers a greedy algorithm called *stream routing* which optimizes the latency in streaming applications.

1.5.1.3 Meta-heuristics

Meta-Heuristics are other heuristic-based algorithms which are problem-independent unlike heuristics. It means that they consider the problem as a black box and they know nothing about the problem details itself. They can be applied to various unrelated problems.

Peng *et al.* (2019) uses a binary genetic algorithm and compares response time and costs with some base approaches. It also uses a bottleneck algorithm to find the bottleneck of a DAG.

1.5.1.4 Other strategies

There are other strategies considered for the deployment of DSP operators over the resources. Some papers use Artificial Intelligence (AI) techniques to find a mapping between operators and resources. Some other algorithms combine different strategies and use them to discover the mapping. As an example, Khare *et al.* (2019) uses a machine-learning algorithm to predict the longest latency path in the DAG of operators. In addition, it employs a greedy algorithm to allocate resources to operators.

1.6 Evaluation strategies

There are different ways in which papers have tried to evaluate their approaches. In placement problems in edge computing, having a specific number of devices is very important, as in the real world, we might have many devices in the edge layer of the network. Due to this fact, we have two main types of evaluation strategies: simulators and testbeds. We review them in the following section.

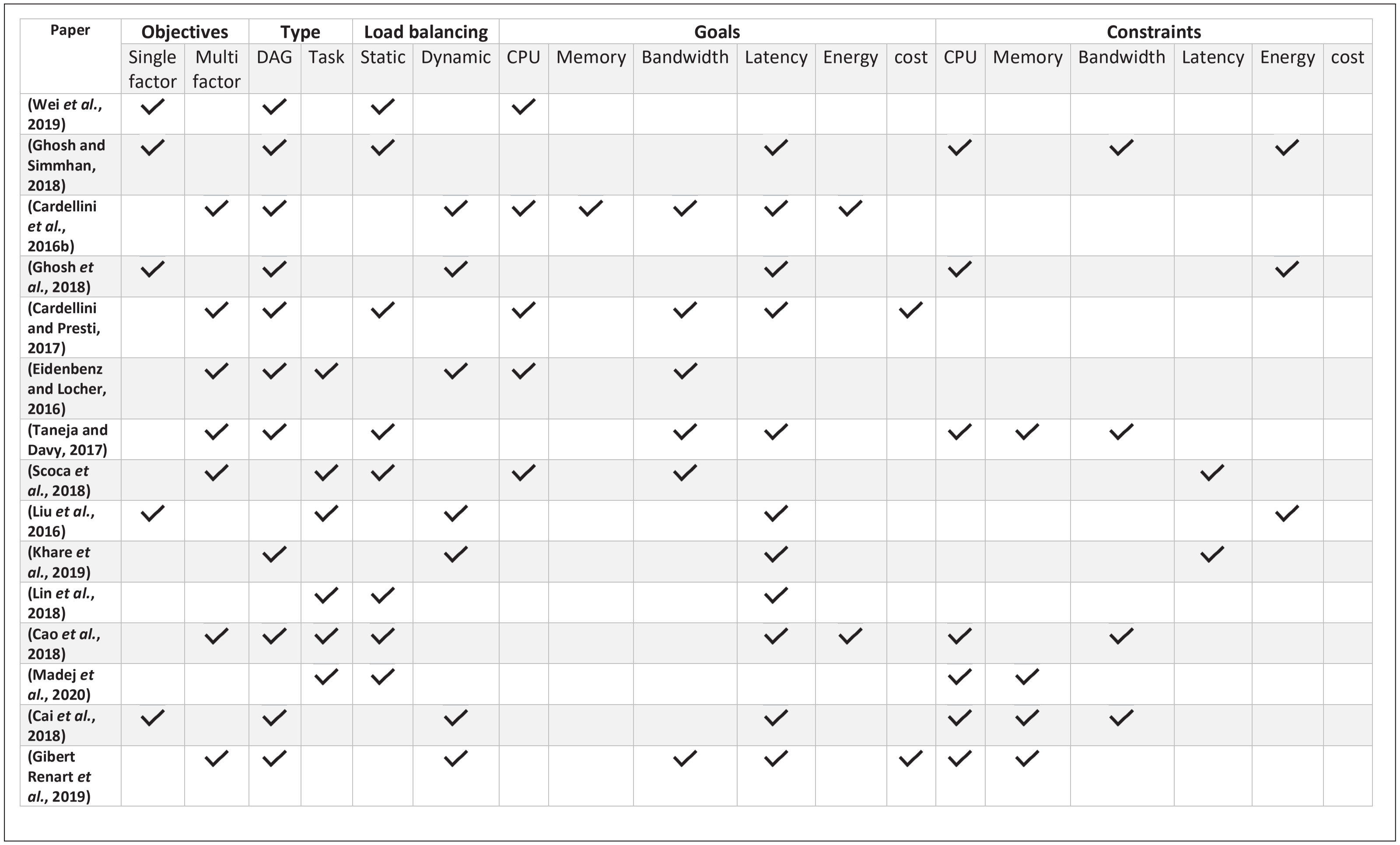
1.6.1 Simulators and simulation-based evaluation

These are the most popular types of evaluations for operator placement problems. Due to the limitations on providing physical resources, most papers test their approaches in a simulation environment. OMNET++ is one of the most popular simulator environments, which is used mostly in papers like da Silva Veith *et al.* (2021) to simulate the topologies. Another simulator is ECSSim++ which is used by Amarasinghe *et al.* (2018). CloudSim is another simulator which is provided by Calheiros, Ranjan, Beloglazov, De Rose & Buyya (2011), and it is used in papers like Cao *et al.* (2018). EdgeCloudSim is another simulator that is defined by Sonmez, Ozgovde & Ersoy (2018) for the combination of edge and cloud environments. Scoca *et al.* (2018) simulate their scenarios in EdgeCloudSim. Moreover, some papers implemented their own simulators. For example Peng *et al.* (2019) uses Matlab for simulating the environments and testing their approaches. In this thesis, we implemented our simulators based on the constraints and goals of our problem.

1.6.2 Deployment over experimental test-beds

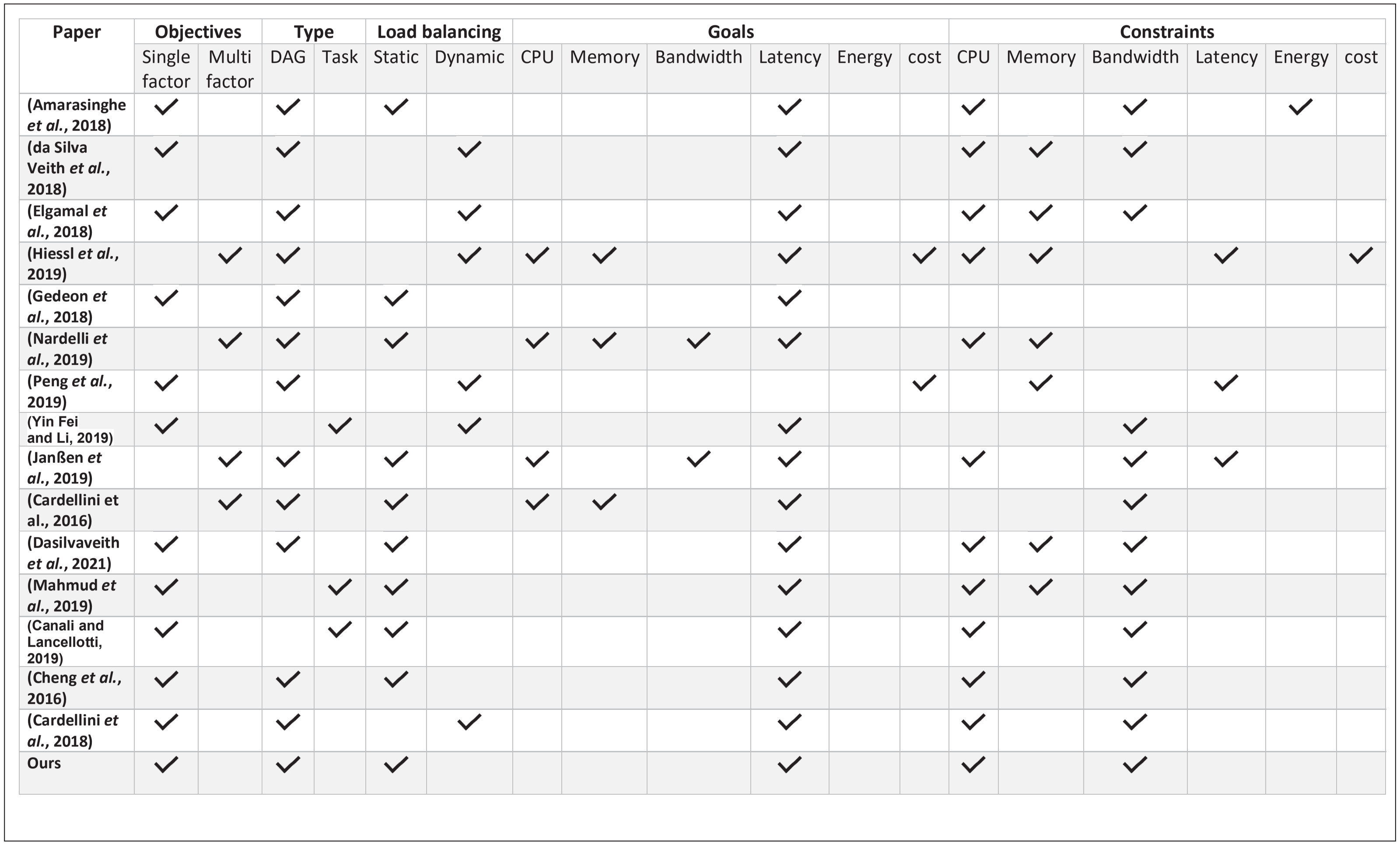
This evaluation approach is not used a lot. Some of the papers tried to use powerful systems and create VMs. Some others limit the size of physical devices and use a laptop as the edge device and a server as the cloud part. To the best of our knowledge, the maximum size of IoT devices that are used in papers is 13 Raspberry Pis (in Renart *et al.* (2019)). Wei *et al.* (2018) is another paper that uses 40 blade servers to evaluate their scheduler on Apache Storm. On the other hand, there is a limitation in the currently provided testbeds because none of the current approaches use physical edge nodes provided at the edge layer of the network. In this thesis, we use a testbed that contains 36 Raspberry Pis which act in a similar manner to edge nodes (in fact, these devices are often used in edge deployments). Despite the added difficulty of deployment, we decided to use the real evaluation approach to obtain more representative results.

We synthesize all of the mentioned features in the placement problem for each related paper in Figures 1.8, 1.9, 1.10 and 1.11. For some papers, we consider a static method for the load balancing since the paper does not provide any information on the load balancing. Similarly, some papers do not mention the infrastructure for the proposed methodology. We consider all types of resources for those papers.



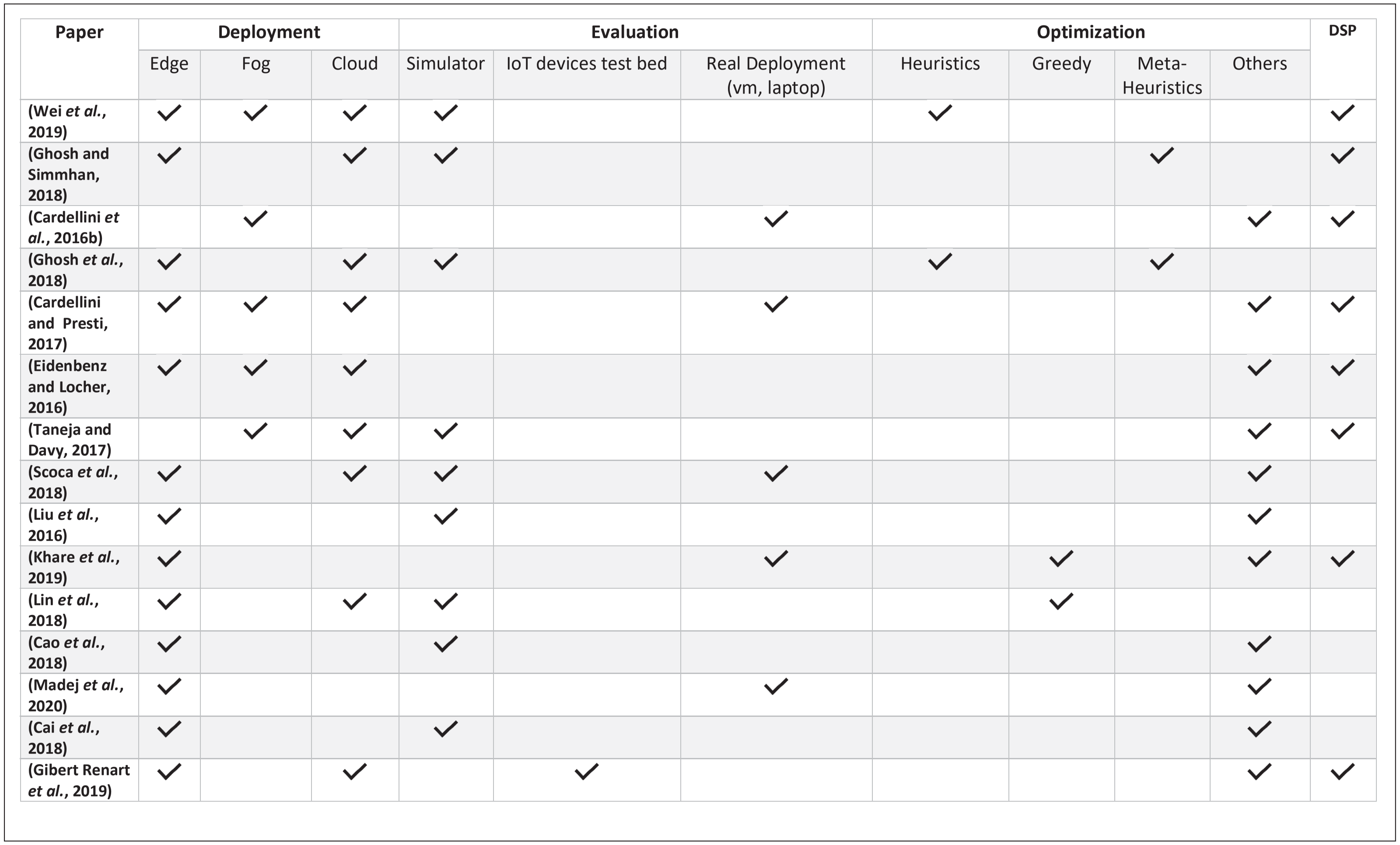
Paper	Objectives		Type		Load balancing		Goals						Constraints					
	Single factor	Multi factor	DAG	Task	Static	Dynamic	CPU	Memory	Bandwidth	Latency	Energy	cost	CPU	Memory	Bandwidth	Latency	Energy	cost
(Wei *et al.*, 2019)	✓		✓		✓		✓											
(Ghosh and Simmhan, 2018)	✓		✓		✓					✓			✓		✓		✓	
(Cardellini *et al.*, 2016b)		✓	✓			✓	✓	✓	✓	✓	✓							
(Ghosh *et al.*, 2018)	✓		✓			✓				✓			✓				✓	
(Cardellini and Presti, 2017)		✓	✓		✓		✓		✓	✓		✓						
(Eidenbenz and Locher, 2016)		✓	✓	✓		✓	✓		✓									
(Taneja and Davy, 2017)		✓	✓		✓				✓	✓			✓	✓	✓			
(Scoca *et al.*, 2018)		✓		✓	✓		✓		✓							✓		
(Liu *et al.*, 2016)	✓			✓		✓				✓							✓	
(Khare *et al.*, 2019)			✓			✓				✓						✓		
(Lin *et al.*, 2018)				✓	✓					✓								
(Cao *et al.*, 2018)		✓	✓	✓	✓					✓	✓		✓		✓			
(Madej *et al.*, 2020)				✓	✓								✓	✓				
(Cai *et al.*, 2018)	✓		✓			✓				✓			✓	✓	✓			
(Gibert Renart *et al.*, 2019)		✓	✓			✓			✓	✓		✓	✓	✓				

Figure 1.8 Feature list one (1)



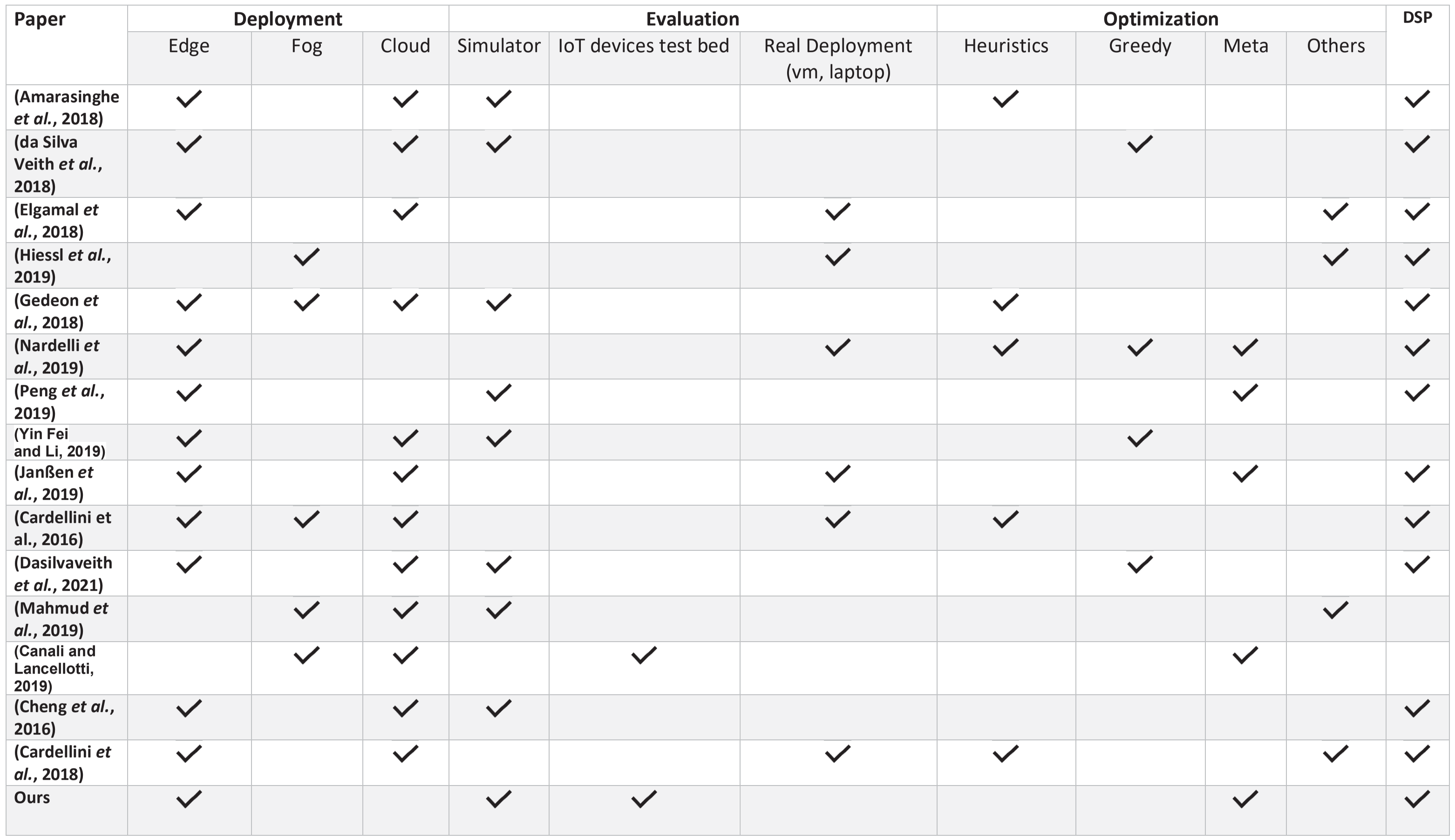
Paper	Objectives		Type		Load balancing		Goals						Constraints					
	Single factor	Multi factor	DAG	Task	Static	Dynamic	CPU	Memory	Bandwidth	Latency	Energy	cost	CPU	Memory	Bandwidth	Latency	Energy	cost
(Amarasinghe *et al.*, 2018)	✓		✓		✓					✓			✓		✓		✓	
(da Silva Veith *et al.*, 2018)	✓		✓			✓				✓			✓	✓	✓			
(Elgamal *et al.*, 2018)	✓		✓			✓				✓			✓	✓	✓			
(Hiessl *et al.*, 2019)		✓	✓			✓	✓	✓		✓		✓	✓	✓		✓		✓
(Gedeon *et al.*, 2018)	✓		✓		✓					✓								
(Nardelli *et al.*, 2019)		✓	✓		✓		✓	✓	✓	✓			✓	✓				
(Peng *et al.*, 2019)	✓		✓			✓						✓		✓		✓		
(Yin Fei and Li, 2019)	✓			✓		✓				✓					✓			
(Janßen *et al.*, 2019)		✓	✓		✓		✓		✓	✓			✓		✓	✓		
(Cardellini et al., 2016)		✓	✓		✓		✓	✓		✓					✓			
(Dasilvaveith *et al.*, 2021)	✓		✓		✓					✓			✓	✓	✓			
(Mahmud *et al.*, 2019)	✓			✓	✓					✓			✓	✓	✓			
(Canali and Lancellotti, 2019)	✓			✓	✓					✓			✓		✓			
(Cheng *et al.*, 2016)	✓		✓		✓					✓			✓		✓			
(Cardellini *et al.*, 2018)	✓		✓			✓				✓			✓		✓			
Ours	✓		✓		✓					✓			✓		✓			

Figure 1.9 Feature list one (2)



Paper	Deployment			Evaluation			Optimization				DSP
	Edge	Fog	Cloud	Simulator	IoT devices test bed	Real Deployment (vm, laptop)	Heuristics	Greedy	Meta-Heuristics	Others	
(Wei *et al.*, 2019)	✓	✓	✓	✓			✓				✓
(Ghosh and Simmhan, 2018)	✓		✓	✓					✓		✓
(Cardellini *et al.*, 2016b)		✓				✓				✓	✓
(Ghosh *et al.*, 2018)	✓		✓	✓			✓		✓		
(Cardellini and Presti, 2017)	✓	✓	✓			✓				✓	✓
(Eidenbenz and Locher, 2016)	✓	✓	✓							✓	✓
(Taneja and Davy, 2017)		✓	✓	✓						✓	✓
(Scoca *et al.*, 2018)	✓		✓	✓		✓				✓	
(Liu *et al.*, 2016)	✓			✓						✓	
(Khare *et al.*, 2019)	✓					✓		✓		✓	✓
(Lin *et al.*, 2018)	✓		✓	✓				✓			
(Cao *et al.*, 2018)	✓			✓						✓	
(Madej *et al.*, 2020)	✓					✓				✓	
(Cai *et al.*, 2018)	✓			✓						✓	
(Gibert Renart *et al.*, 2019)	✓		✓		✓					✓	✓

Figure 1.10 Feature list two (1)



Paper	Deployment			Evaluation			Optimization				DSP
	Edge	Fog	Cloud	Simulator	IoT devices test bed	Real Deployment (vm, laptop)	Heuristics	Greedy	Meta	Others	
(Amarasinghe *et al.*, 2018)	✓		✓	✓			✓				✓
(da Silva Veith *et al.*, 2018)	✓		✓	✓				✓			✓
(Elgamal *et al.*, 2018)	✓		✓			✓				✓	✓
(Hiessl *et al.*, 2019)		✓				✓				✓	✓
(Gedeon *et al.*, 2018)	✓	✓	✓	✓			✓				✓
(Nardelli *et al.*, 2019)	✓					✓	✓	✓	✓		✓
(Peng *et al.*, 2019)	✓			✓					✓		✓
(Yin Fei and Li, 2019)	✓		✓	✓				✓			
(Janßen *et al.*, 2019)	✓		✓			✓			✓		✓
(Cardellini et al., 2016)	✓	✓	✓			✓	✓				✓
(Dasilvaveith *et al.*, 2021)	✓		✓	✓				✓			✓
(Mahmud *et al.*, 2019)		✓	✓	✓						✓	
(Canali and Lancellotti, 2019)		✓	✓		✓				✓		
(Cheng *et al.*, 2016)	✓		✓	✓							✓
(Cardellini *et al.*, 2018)	✓		✓			✓	✓			✓	✓
Ours	✓			✓	✓				✓		✓

Figure 1.11 Feature list two (2)

CHAPTER 2

ARCHITECTURE AND SYSTEM DESIGN

This chapter discusses the main characteristics of the problem, the research questions, and the implementation approaches. In Section 2.1 the main goals and assumptions are explained, 2.2 describes the problem. Section 2.3 shows the system model used for the proposed tools, and in Section 2.4, we review different aspects of the problem in more detail. Section 2.5 explains how we implement our simulator and in 2.6, the main used algorithms are presented.

2.1 Design goals and hypothesis

In this thesis, we propose a stream processing system that minimizes the latency, by finding the best mapping between the resources and the operators in the DSP applications. The proposed system targets an initial deployment, and provides static scheduling of the tasks by optimizing the deployment of resources on the best available edge resources. A dynamic version of this system will be explored in the future work.

2.1.1 Assumptions

Several metrics, such as CPU and bandwidth, are considered in the proposed system for the purpose of optimization. We consider different assumption in this thesis. We review them as follows:

- CPU usage refers to the running time of the tasks, and the bandwidth models the network utilization. We discuss this metric more in Section 2.4.2.
- System memory (RAM) is not taken into account, as we assume that most edge devices nowadays have enough memory resources (e.g. Raspberry Pi 3 has 1GB of RAM) to run most IoT applications. It is worth mentioning that memory (RAM) is different from the required storage for saving the data.
- Another factor is buffering, which is not supported in the system because measuring and tracking the available buffer sizes would make the problem too complicated. There are too

many potential factors, such as context switching, packet preparation and packet forwarding, which are not considered in this thesis because of the complexity they add to the problem. Currently, other metrics such as the data rate and selectivity ratio are considered.

- We do not consider any limitations on the size of the operators or edge nodes, which allows us to handle many stream processing applications. A given operator is usually a simple straightforward operation that is executed in a certain amount of time.
- We assume that the execution of each operator is done in a single thread deployed on one edge node. We will consider the deployment of multiple operators on one edge node in future work.
- Operator replication is another feature in placement problems, and it means creating a new instance of an operator in the problem. This feature would be considered as future work in this thesis.
- In this thesis, only one instance of each operator is permitted. As a result we do not have repetitive operators and we can have more different operators. Having multiple instances of one operator can make the problem complex. We simplify the model by considering one instance for each operator.
- Operators can perform different operations on the incoming data. These operations can change the size of each message. As a result, the output messages might have a different size. Similarly, the in-rate and out-rate messages of an operator can be changed based on the operations of operators.
- Every application requires a constant value of CPU cycles or a specific number of assembly instructions.
- We assume that all the parameters of the applications are constant in EdgeDAG. Average value is considered for the constant parameters.

As we explain in Section 1.4, given that the placement problem is NP-hard, the purpose of the proposed system is to provide a good solution, as we cannot guarantee that an optimal solution will be found.

2.2 Research problem

As we mentioned in the background section, most of the current stream processing applications are time-sensitive. To meet this requirement, operators in such applications need to be deployed over the best available resources provided by the edge network. In this thesis, the goal is to find the best placement of operators over the edge nodes to achieve the lowest latency for delivering the messages to the sinks, while meeting all the constraints of the problem. To be more precise, we are aiming at minimizing the average per-message latency of applications deployed over the edge network. Latency for a message refers to the time it takes to transfer a message from a source operator to a sink operator. The bandwidth of edge devices and CPU usage of operators are taken into account as the hard constraints. To this end, we propose EdgeDAG, a system that takes as input the aforementioned parameters and that finds a suitable solution that optimizes the latency for the operator placement problem. EdgeDAG provides the following features:

- Receiving the configuration of the designed topology and the application.
- Profiling the application using a real edge network.
- Running a simulation of the DAG, and optimizing using the Grey Wolf Optimizer algorithm.
- Provide the best mapping between the edge nodes and operators in the topology

To validate EdgeDAG, we use it with the Apache Storm framework. We create multiple random applications on Apache Storm and then execute them using the proposed tools. Then we compare EdgeDAG with other strategies: random deployment (which is the default scheduler of Apache Storm) and cloud deployment.

It is worth mentioning that the proposed strategy does not impose any limitations on application complexity. But in more complex applications, the proposed methodology requires more time to find a good placement. More precisely, in a complex application, the number of operators and edges is increased. As a result, we can place operators in more different ways over the edge network. In addition, by increasing the size of devices in an edge network, there are more options for the placement of operators. Similarly, the proposed methodology requires more time to find a better placement when the size of the edge network is increased.

2.3 System model

This section introduces EdgeDAG, our scheduling tool for Apache Storm. We describe its architecture and show why it is suitable for various stream processing applications.

To find the best available initial placement of operators, we propose EdgeDAG that finds the best available nodes for each specific task (operator). As it is shown in Figure 2.1, the system contains five main components:

- **Random problem generation**: This component is used to generate random DAG stream processing topologies. It considers parameters such as the amount of processing [1] to consider, the size of the topology, the selectivity ratio of operators, the message size, the density of connections between the operators, the data rate at the sinks and the probability ratio of the downstream operators. This component is used to define a topology. It would also be possible to generate a topology manually by specifying the DAG of operators on a DOT file and defining the required topology features on the configuration file.
- **Profiling**: In the next step, we run Apache Storm with the default scheduler to extract the required information from the topology. We run this step for 7 minutes and use the last 5 minutes of data since there might be some overhead on the metrics in the first 2 minutes of running Apache Storm. This step needs to be done over Apache Storm to be able to collect the characteristics of applications in a real deployment. The information gathered contains the incoming message size and outgoing message size of each operator, the CPU usage of each application and the input and output data rate of each operator.
- **Log analyzer**: In this step, we collect the logs for each node created by Storm and process the logs to have access to the information for each node. The *analyzer* calculates the incoming and outgoing average message size by monitoring the amount of data transferred and the size of packets that have been transferred. It also finds the average CPU usage for each run of the operators based on both the in-rate and out-rate of operators.
- **Simulator**: After receiving all the collected metrics from the Log analyzer, the simulator simulates the exact environment of the edge network and tries to find the best deployment of

[1] Processing is simulated

tasks over the resources using the information gathered in the profiler. The *simulator* uses a meta-heuristic algorithm called Grey Wolf Optimization (GWO) to find the optimal solution in the large search space.

- **Deployer**: This component provides the results from the simulator for the Apache Storm scheduler, and runs Apache Storm from scratch. In the end, the *deployer* also collects the results to compare them with the other approaches.

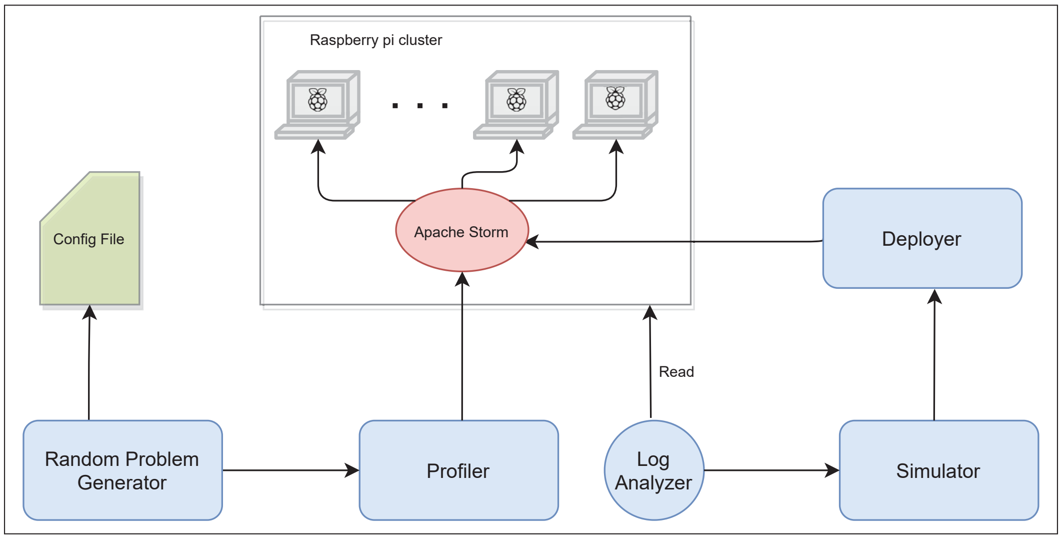


Figure 2.1 EdgeDAG system architecture

2.4 Problem formulation

In this section, we review the different components of problem in more detail. In Section 2.4.1, the parameters are described, then we review the constraints in Section 2.4.2, we explain the fitness function in our tools in Section 2.4.3 and we present the Grey Wolf Optimizer in more details in Section 2.4.4.

2.4.1 Modeling and parameters

In 2.3, we defined the general structure of our system and review the main concepts used in the operator placement of distributed stream processing applications. As we discussed in 1.3.4, we have multiple metrics in the operator placement problems. In this thesis, we summarize the main metrics in Table 2.1. It is worth mentioning that we assume that these parameters are constant during execution.

Table 2.1 Parameters

Notation	**Description**
$G = (N, O)$	Problem graph
N_i	Edge node
O_i	Operator *i*
I_i	Instruction size of operator *i*
R_i	Available resource of edge node *i* in MIPS
B_i^{in}	Incoming bandwidth to node *i*
B_i^{out}	Outgoing bandwidth from node *i*
$B_{(i,j)}$	Bandwidth between node *i* and *j*
λ_i^{in}	Incoming throughput to node *i*
λ_i^{out}	Outgoing throughput from node *i*
θ_i^{out}	Outgoing message size for operator *i*
θ_i^{in}	incoming message size to operator *i*
ω_i^{in}	Input data rate into operator *i* per second
ω_i^{out}	Output data rate from operator *i* per second
$L_{(i)}^{in}$	Incoming latency to node *i*
$L_{(i)}^{out}$	Outgoing latency from node *i*
in_i	Number of required messages for one time running of operator *i*
out_i	Number of messages that operator *i* generates in each run
$P_{(i,j)}$	Probability ratio from operator *i* to operator *j*

To have a better understanding of parameters, we provide an example on Figure 2.2. We can see operators O_1 and O_2 which are placed on the edge nodes N_1 and N_2 respectively. The selectivity ratio of each operator is presented on the top of the operators ($in_{O_1} : out_{O_1}$ and $in_{O_2} : out_{O_2}$). The size of data generated by operator O_1 is calculated according to the output data rate of operator O_1 ($\omega_{O_1}^{out}$) and the outgoing message size of operator O_1 ($\theta_{O_1}^{out}$). We have the same parameters for operator O_2. For operator O_2, the size of incoming data is depend on the incoming data rate ($\omega_{O_2}^{in}$) and incoming message size ($\theta_{O_2}^{in}$). The throughput values of the two edge nodes are presented as the variable λ. We review all these parameters in more detail in Section 2.4.2.2.

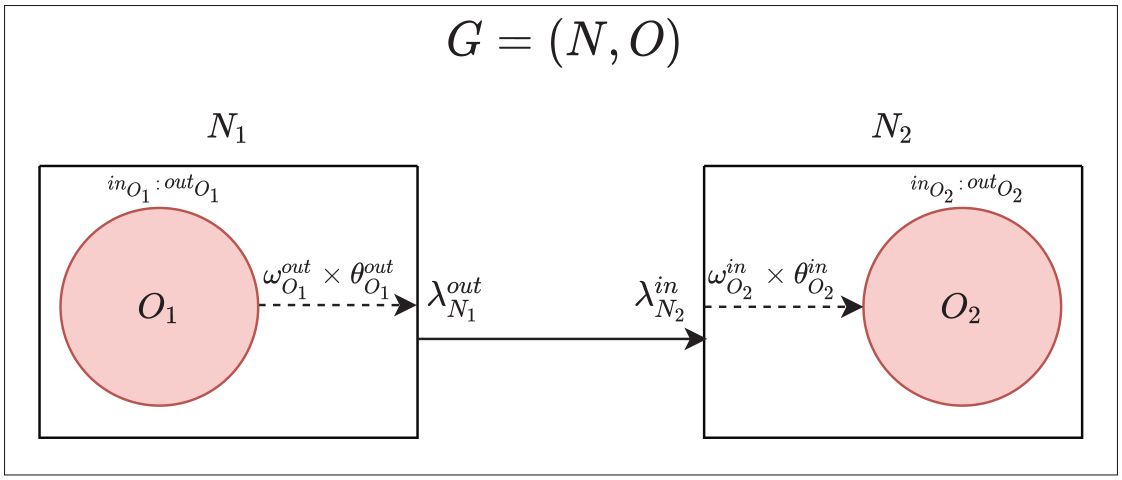


Figure 2.2 An example of parameters in a placement problem

We reformulate the problem as a directed acyclic graph $G = (N, O)$ where N represents the set of edge nodes in our edge network, and O is the set our operators.

2.4.1.1 Task execution time

Each edge node has a constant CPU power, which is measured in MIPS [2]. On the other hand, every operator has a specific instruction size. As it is shown in Equation 2.1, $T_{(i,k)}$ shows the execution time of operator i which is placed on the edge node k. It is calculated by dividing the instruction size by the CPU power of the device. In the equation, I_i indicates how many *millions of instructions* are required for running operator i, and R_k models the total amount of MIPS that resource k can handle.

$$T_{(i,k)} = \frac{I_i}{R_k} \quad \forall i \in O; k \in N \tag{2.1}$$

2.4.1.2 Message transfer time

The edge nodes are connected, and they have specific incoming and outgoing bandwidth values. It is worth mentioning that the bandwidth between two edge nodes is limited to the minimum

[2] Million Instructions can be run Per Second

bandwidth of each edge node as Equation 2.2 represents. B_i^{out} shows the outgoing bandwidth of edge node i and B_j^{in} shows the incoming bandwidth to edge node j. The bandwidth between these two edge nodes would be the minimum values of them.

$$B_{(i,j)} = \min\{B_i^{out}, B_j^{in}\} \quad \forall i, j \in N \tag{2.2}$$

The throughput indicates the bandwidth usage of the network. Since we do not consider the changes in the network condition during execution (i.e., as we are providing a static partitioning model), we have specific constant latency between two edge nodes. The first one is the latency applied to outgoing messages, and the second one is the incoming latency which causes the latency on incoming messages. On the other hand, every message has a specific size. According to all these factors, the transfer times of messages are calculated in Equation 2.3. θ_i^{out} models the outgoing message size from operator i and θ_j^{in} presents the incoming message size to operator j. We first calculate the average message size between two operators and then divide it by the bandwidth. $B_{(p,k)}$ shows the bandwidth between two edge nodes p and k that run the operators i and j respectively. Then we add the incoming latency and outgoing latency to it. L_p^{out} shows the outgoing latency from node p, and L_k^{in} represents the incoming latency to node k. As it is observable, the message size is calculated for the operators and the bandwidth and latency values are considered for the edge nodes.

$$TT_{(p,k)} = \frac{Average(\theta_i^{out}, \theta_j^{in})}{B_{(p,k)}} + L_p^{out} + L_k^{in} \quad \forall p, k \in N; \forall i, j \in O \tag{2.3}$$

The data rate is another parameter that is considered for each edge node, and it includes both the incoming and outgoing data rates of the operators. Another parameter is the in-rate value and out-rate value for an operator. Some operators generate one or more outgoing messages (i.e., the out-rate) upon receiving a certain number of messages (i.e., the in-rate).

2.4.1.3 Probability ratio

Another factor is the probability ratio, which is relevant when there are multiple downstream operators. It is used to balance the flow of messages among all of the downstream operators. Each operator has a specific probability ratio to each downstream operator between 0 and 1. Then a weighted random selection decides on how to send the data to the downstream operators, according to all the probability ratios.

2.4.2 Constraints

As we mentioned, in our case, the constraints are the CPU and bandwidth capacities of the edge nodes. The CPU and bandwidth are the two most effective factors that have the highest impact on the operator placement problem. We do not consider memory as a constraint since we believe nowadays many edge devices have plenty of memory resources (e.g., Raspberry Pi 3 has 1GB of RAM). Moreover, it allows us to reduce the complexity of the problem. Additionally, as we have mentioned, we have not considered buffering in this thesis. We review the CPU and bandwidth constraints in more details in the next sections.

2.4.2.1 CPU

As we discussed in literature review section, edge computing provides resources that are closer to the end-users, but these resources have some limitations, as they are not as powerful as general-purpose servers. In a given edge network, all the devices have limited CPU power. They can run a specific amount of processes and it means that they have constant CPU power. As a result, running applications on the resource can take more time than on a dedicated server (e.g., cloud). In the situation in which we have a limited buffer on devices, the execution times of the tasks become very important. When tasks take a long time to execute, the buffer fills up. Because of this, the tasks are sometimes rejected and the running would be halted as a result of the buffer overflow. Parameter R_i in Table 2.1 shows the CPU limitation of each edge node.

2.4.2.2 Bandwidth

Another limitation of edge computing is the network. Edge nodes are connected and transfer the data. But because of the limitations in the network connectivity of various edge devices (e.g., wifi, cellular, Ethernet), variability in the available bandwidth can lead to different transfer speeds. We consider bandwidth as one of our constraints – we define a speed limit on the data transfer. In this thesis, we model two types of per-device bandwidth (i.e., incoming and outgoing); these are modelled as the parameters B_i^{in} and B_i^{out} respectively.

To meet the bandwidth requirements, we need to be assured that the data transferring speed of the devices is always lower than the bandwidth. Data transferring speed is known as the throughput of devices. It refers to the amount of data that comes into a device and goes out. Two types of incoming and outgoing are defined, and they are shown as λ_i^{in} and λ_i^{out} respectively. ω_i^{in} and ω_i^{out} show the input data rate and output data rate of the operator i. By multiplying the data rate by the size of the message, we can calculate the throughput of a device which are shown in Equations 2.5 and 2.7. Variable θ_i shows the message size in the operator i.

We model the outgoing bandwidth constraint by the following equations:

$$\lambda_k^{in} \leq B_k^{in} \quad \forall k \in N \tag{2.4}$$

Where:

$$\lambda_k^{in} = \omega_i^{in} \times \theta_i^{in} \quad \forall k \in N; i \in O \tag{2.5}$$

Similarly, the following equations are defined for the incoming bandwidth constraint:

$$\lambda_k^{out} \leq B_k^{out} \quad \forall k \in N \tag{2.6}$$

Where:

$$\lambda_k^{out} = \omega_i^{out} \times \theta_i^{out} \quad \forall k \in N; i \in O \tag{2.7}$$

Equations 2.4 and 2.6 show that the throughput is lower than the bandwidth. By defining these constraints, we are assured that there is enough bandwidth for sending the messages.

2.4.3 Fitness function

To solve the problem mentioned in Section 2.2, we focus on the optimization methods. In the optimization problems, an objective function that represents the goals is defined. In most optimization problems, the goal is to optimize the objective function. We also have another function called the fitness function, which is an objective function that shows how close to the best global solution of the search space the results are. The fitness function is also generated according to the goals of the problem.

The fitness function considers different factors. In this thesis, the metric is the average latency for delivering the messages from multiple sources to sinks while the constraints are met. Messages are sent by the sources and travel along specific paths before being received by the sinks. Our tools calculate the average latency by considering all paths a message takes to be received by the sinks. As the message passes through a path, it can be modified, aggregated with other messages, and changed. The message may be doubled and sent to another path. The average latency is measured according to the latency of every message and in all the paths. We monitor the start time for a message at the source and finish time at all the sinks affected by the message.

- Considering $M = [m_1, m_2, .., m_N]$ all the messages generated by the sources and $P_m = [p_1, p_2, .., p_N]$ all the paths that the message m affects.
- The fitness function is summarized as the average of $Latency_{(p,m)}$ while $m \in M$ and $p \in P_m$.

2.4.4 Grey Wolf Optimization

Grey Wolf Optimization (GWO) was first proposed in Mirjalili, Mirjalili & Lewis (2014). It has been used in a variety of applications and has become a popular meta-heuristic algorithm. It is a population-based meta-heuristic algorithm that runs the optimization based on a set of solutions.

It first starts the optimization process using a random solution, and then calls the fitness function, which informs the optimizer on how good the solution is.

Meta-heuristic algorithms contain two phases: exploration and exploitation. The exploration phase mainly refers to the random and global search of the search space. In the exploitation phase, the optimizer works on the local search and searches a specific region of search space. It is important to have a logical balance between these two phases in GWO.

GWO is inspired by the attack of wolves in the group. So it splits the population into the four types, namely α, β, γ and ω. It has a specific type of social dimension that is shown in Figure 2.3, and each group has specific tasks to do depending on the level.

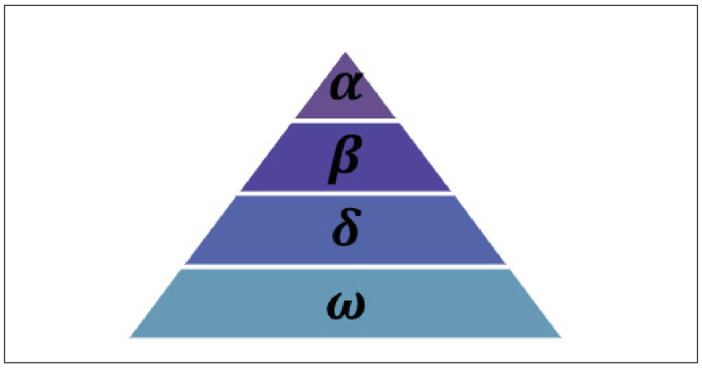


Figure 2.3 Hierarchy of grey wolf
Taken from Mirjalili *et al.* (2014)

The main steps in GWO are summarized as below:

1. **Social hierarchy**: We save the fittest solution in α, the second one in β and the third one in γ. Other solutions are represented in ω.
2. **Encircling prey**: In the next step, GWO considers the encircled prey during the hunt for the wolves. It means that each grey wolf updates its position inside the search space considering random numbers. They move around the solutions obtained up until now.
3. **Hunting**: Wolves are able to encircle the prey. In the hunting step, α tries to guide the other wolves. β and γ also can help in identification. Since α, β, γ represent the best solution so

far, we need to update the position of the other wolves according to the location of these three types of wolves.

4. **Attacking prey**: Wolves attack the prey when they stop moving. Wolves update their position near the prey by bounding random factors. So the local search or exploitation happens in this step.
5. **Search for the prey**: Most of the time, grey wolves update their position based on α, β, γ and ω. This step helps the wolves search globally by inserting a random value larger than 1 or -1 into the GWO.

Generally speaking, GWO starts searching by creating random positions for wolves. Over the iteration, wolves update their positions according to the α, β and γ wolves. They consider both the local search and global search by changing the range of random factors and updating their positions.

2.4.4.1 Application of GWO

In this section, we review the usage of GWO into the simulator. In the simulator, GWO is used to find a suitable solution in the search space. In order to map the GWO to our problem, we define the fitness function in the topology. The fitness function connects the GWO and our problem. Generally, GWO receives the size of the edge network and the size of the operator and consider an ID for every edge node and operator. Then, it assigns an edge node to every operator in the topology. After generating a solution, the fitness function returns a value that shows how good the solution is. As we discussed, GWO is a population-based meta-heuristic algorithm. So it means that it calls the fitness function multiple times. Fitness functions calculate latency based on the start and finish times of each message ID. This process is repeated several times for different mapping configurations according to the number of populations and number of iterations defined for the GWO. GWO saves the best mapping for the usage of Apache Storm scheduler.

Since running the simulator for one time does not cover all the probabilities that can happen in a real deployment, we decided to create another variable called the number of repetitions. It shows the number of times the DAG simulator should be run for one mapping configuration. By doing so, we can run one mapping configuration multiple times and have more random values in sending the messages to downstream operators.

2.5 DAG simulation

In this section, we explain the simulation in more details. The simulator is implemented in an event-based manner. It means that various events happen at specific times, and then we update all the information in the simulation components. In our simulator, the main components are the operators and edge nodes. The simulator starts running after assigning each operator to a specific edge node. An event is another component that contains information such as the sending time, the message and its receiving time in the simulator. Events can be created by the operators and can be transferred among the operators. Further, we have a global time variable that starts from zero and increases until the simulator is stopped. Different methods are implemented that facilitate the update process in the DAG.

2.5.1 Source selection

In the simulation, sources need to know when they should send the messages. Our source selection involves determining when to send the data to the sources. The calculated time allows us to determine when sources should send the data. Whenever we want to send a message, we select the source for the message by using source selection. Then we update the global time variable. Since we have an in-rate and out-rate of more than one for each operator, we need to use source selection several times. In other words, the downstream operators require more messages to be able to generate new messages according to their in-rate and out-rate. This function is triggered whenever the DAG cannot be updated due to the limited number of messages received by the downstream operators.

All the operators in the simulator have special functionality. We summarize them in the following sections.

2.5.2 Receive event

After selecting the source, we create an event and use the receive event functionality on the source operators. In other words, we start feeding the DAG by using this function on the source operators. This functionality is used to inform an operator that it has received an event. So the operator receives the message by the defined event. All the received events are saved as a queue in the operator. In this step, the operator checks if it is a good time to start processing the messages. The receive event determines if there are enough messages to start processing. If there are not enough messages, it sends a signal to the upstream operator and informs them that it requires more messages to continue running. If there are messages in the saved queue, it calls the process event to start processing the messages.

2.5.3 Process event

If the operator is ready to process the message, this function starts working. The processing of messages is done using FIFO (First-In, First-Out). It selects the first messages for processing by checking the queue and removing the event from the queue. Then it calculates the processing time, and as a result, it knows when it is ready to send the result. So it uses the send event function for sending an event.

2.5.4 Send event

In this function, an event is created and defined by the provided finishing time from the process event. Then it uses a weighted random function to choose the destination for each created event. The weighted random function uses the probability ratio to decide the destination of each event. We calculate the network latency in this function according to the bandwidth and latency. Each operator is assigned to the edge node before starting the process. So we know the current edge node and the destination edge node. As a result, we can calculate the network latency between the current operator and the destination operator. Then we determine the receiving time of the event by the destination operator. So we call the received event on the destination operator and pass the event. It is worth mentioning that we create events based on the out-rate of the operator, and for each event, we decide about the destination.

2.5.5 Stop signal

As we discussed, operators inform their upstream operators if they need more messages for processing. The process of informing the upstream operators is defined using a signal. Operators return a signal and inform the sources that they require more messages to start processing. The sources continue to send the messages as far as they receive this signal from operators. This process continues until a sink finishes processing the messages. It means that it is run at least on time. So it returns a new type of signal to the sources, which is called a stop signal. The dag simulation would be stopped whenever the stop signal is received from all the sinks.

2.5.6 Latency calculation

To be able to calculate the average latency per message, our system creates a unique identifier for each message and records the time at which the message was sent. The information is stored in a map structure and transferred to downstream operators. Each downstream operator sends a new Map variable containing all the received map variables. This process continues until the sink receives the message IDs and start times. Then, the sink assigns an end time for all the received messages. In the end, we calculate the average time it takes for each messages id to be received by the sinks using the assigned start time and end time.

To better illustrate the mechanisms of the simulator described in the previous sections, we provide Figure 2.4 which shows how the messages walk through the operators.

- **Step A**: The source sends the data randomly based on the probability ratio to each downstream operator.

- **Step B**: Operator $O1$ sends the data again to the downstream operators. Simultaneously, the messages are received and saved in $O2$, $O3$ and $O4$.
- **Step C**: Operator $O1$ continues sending the new messages. Operators $O2$, $O3$, and $O4$ receive the new data that has been sent by $O1$ in the previous step. The processing happens only in $O2$ and $O3$, and they send the generated messages to the downstream operator. $O4$ also has received the required number of messages and can start processing them.
- **Step D**: $O1$ continues sending since it receives the signal from downstream operators (not the stop signal from sinks). Operators $O2$, $O3$ and $O4$ again receive the new data. Operators $O2$, $O3$ and $O4$ use the process event function and create new messages. Operator $O4$ sends the message to the sink. Operators $O2$ and $O3$ send the messages to $O4$.
- **Step E**: New messages are generated by the source. Operators $O2$, $O3$ and $O4$ have received the previous messages. They send the generated messages to their downstream operators. Operator $O5$ receives a message, and after calculating the finish time, it would send the stop signal recursively to the source, and the DAG would be stopped.

2.6 Algorithms

Different algorithms are used in this thesis. We explain the two algorithms of GWO and fitness function in this section.

2.6.1 GWO model

As we discussed, GWO is a black box that searches the space and is a meta-heuristic that can be applied to most of the problems. In Section 2.4.4.1, we described the mapping of GWO to our problem. In Algorithm 2.1, we review the algorithm in more details.

We have two main components in GWO. The first one is population number which shows the number of wolves or search agents in the search space. The second one is the number of iterations, which shows the number of times that the search should be done with the search agents. We define the solution as an array of edge nodes with the size of operators. So each value

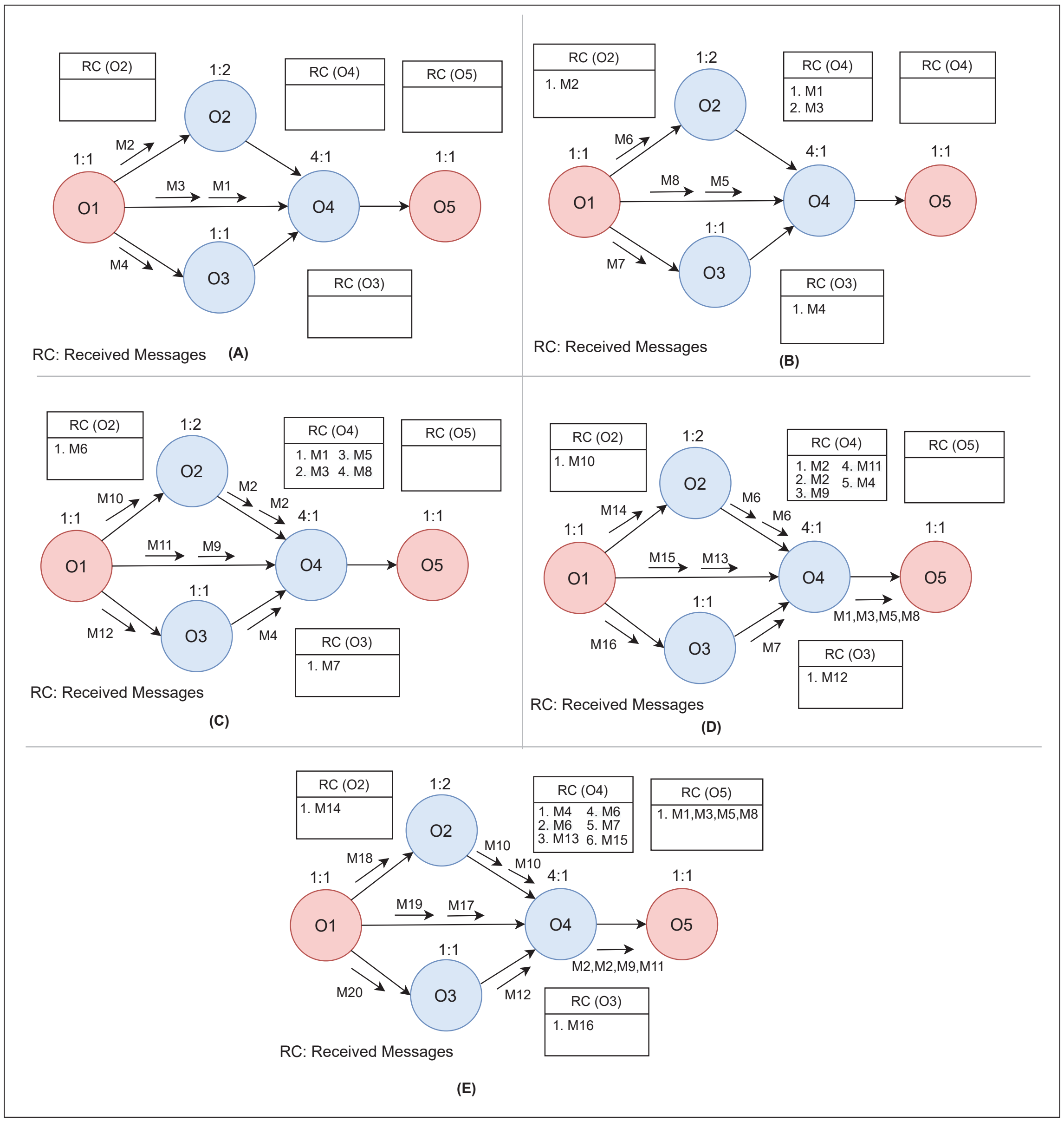


Figure 2.4 DAG simulation steps

in the solution array shows a specific edge node. At first, we use random values as a solution for each wolf and call the fitness function (line 1-3). Then we set the α , β and γ parameters (line 4-6). In the next step, we start iterating and updating the position of each search agent according to the α, β and γ. In other words, we find another array of edge nodes for the operators, which is close to the arrays of α, β and γ (line 9). Then we calculate the fitness function for all the new proposed solutions (line 11). The repeat number is also defined, which shows how many times

Algorithm 2.1 GWO algorithm

```
Input: Dimensions which is number of operators $\mathcal{O} = \{\mathbf{o}_1, ..., \mathbf{o}_J\}$, Edge node id
       number $\mathcal{N} = \{\mathbf{n}_1, ..., \mathbf{n}_L\}$, number of iteration $i$, number of population
       $\mathcal{K} = \{\mathbf{k}_1, , ..., \mathbf{k}_n\}$, $\mathcal{R}$ number of repeats for each solution
Output: placement of each operator based on the edge node id $\mathcal{O}' = \{\mathbf{n'}_1, ..., \mathbf{n'}_J\}$,
        $n' \in N$
1  Assign random solutions to the GWO wolves population;
2  Initialize the random factors of GWO algorithm;
3  Call fitness function for all the GWO wolves population;
4  $\alpha$ = best solution;
5  $\beta$ = second solution;
6  $\gamma$ = third solution;
7  for $i' \in \{0, ..., i\}$ do
8      for each search agent $\in k$ do
9          Update the positions of search agents by choosing the edge nodes from $N$ for
             operators $O$;
10     end for
11     Calculate the fitness of all search agents with the repeat size of $R$ ;
12     Update the random factors of GWO algorithm;
13     Update $\alpha$ , $\beta$ and $\gamma$;
14 end for
15 Return $\alpha$;
```

the DAG-simulator should be run for each proposed solution by the wolves (line 11). After updating the random factors in the GWO algorithm, we can update α, β and γ (line 12-13). At line 15, after finishing the iteration, we return the α as the best solution.

2.6.2 Latency calculation

We define a new type of fitness function in the simulation for the latency calculations. It is summarized as below:

The fitness function works with the DAG simulator. We run the simulation, calculate the average latency and return it as the value of the fitness function. At first, we receive an array of solutions from GWO. Then we set up the operators and topology according to the received array. After that, we define the variable $sinkSizeSeen$, which shows the number of sinks that received at least

Algorithm 2.2 Fitness function algorithm

```
**Input:** placement of each operator based on the edge node id $O' = \{\mathbf{n'}_1, ..., \mathbf{n'}_J\}$,
           number of repeats of DAG
**Output:** Average Latency $L$

1  Setup the topology using the received mapping;
2  $sinksSeenSize \leftarrow 0$;
3  $time \leftarrow 0$;
4  $messageId \leftarrow 0$;
5  $averageLatencyArray \leftarrow 0$;
6  for $i \in \{0, ..., numberOfReapet\}$ do
7      for $TRUE$ do
8          $S \leftarrow findNextSource(time)$ ;
9          $signal = S.receiveEvent(messageId, time)$;
10         $messageId{+}{+}$;
11         $time \leftarrow findCurrentTime(S)$ ;
12         if $isNewStopSignal(signal)$ then
13             $sinksSeenSize{+}{+}$;
14         end if
15         if $sinksSeenSize > Sinks.size()$ then
16             $Break$;
17         end if
18     end for
19     $i{+}{+}$;
20     $latency \leftarrow calculateLatency(sinks)$;
21     $averageLatecnyArray.add(latency)$;
22     $resetTopology()$;
23 end for
24 $calculateAverage(averageLatecnyArray)$;
```

one message. The variable *time* defines the current time in the DAG-simulator. We also specify the message-id, which models the id for each incoming message. Then in the line 6, we define a loop with the size of repeat value. In the second loop, the execution of the DAG-simulator takes place. Our system runs until all stop signals have been received from the sink operators. At line 8, we find the next source that should be triggered, and we call the receive event method on that source operator. It will start sending the message. At line 11, we update the current time based on the source data rate and at the next line, we check if the returned signal is a new stop signal.

If it is a new stop signal, it means that a new sink has received at least one message. At line 15, we check if we saw all the sinks – if this is the case, then we would break the loop. At line 20, the latency is calculated according to the saved messages in the sinks. At the next line, we add the calculated latency to an array. By doing so, it is possible to calculate the average of collected latencies for all the iterations that have been done (line 24). We reset the topology information such as the signals and saved messages (line 22) then we repeat the whole process. Since we have a random selection of downstream operators, we will get a different result for each run. Consequently, we repeat the mapping multiple times to reduce the effects of a random selection of operators. By doing so, we reduce the error rate we might face in a real deployment.

CHAPTER 3

IMPLEMENTATION

In this chapter, we describe our system implementation that we use to evaluate our approach and the fitness function. Our implementation contains the following modules: cluster configuration, random topology generation, profiling, log analyzer, simulator and the deployment. In the next sections, we explain them in more details.

3.1 Programming languages and libraries

Several libraries and tools have been used that help us to implement the proposed method. We also used different programming languages based on our needs.

3.1.1 Java

Java is an object-oriented programming language that was first released in 1995. A key characteristic that makes it a popular language is the portability it offers developers. This feature is designed to help developers run Java applications on various operating systems and hardware. In this thesis, the most used programming language is Java. The simulator and GWO are completely written in Java. Aside from that, we used the Apache Storm framework, which is implemented in Java.

3.1.1.1 Apache Maven

Maven is a tool that helps build Java projects in an automated manner. It was first developed in 2002 and it is supported by Apache Software. In general, Maven automatically downloads and adds all the required dependencies and libraries to the project. This list of dependencies is defined in a file called `pom.xml`. There are two types of repositories where libraries are stored: local repositories and public repositories. From these repositories, all dependencies can be downloaded. It is also possible to put a library in the local repository and use it in a different project.

3.1.2 Python

Python is another popular programming language used in this thesis. Python was released in 1991, and it is known for its code readability. The developers can write their code much more easily with Python. Because of this feature, Python is used in different fields of science. In this thesis, we used Python for reading the logs and latency calculation.

Python also supports scripting, so it is possible to run a Python script without any compilation. Python scripts can be run faster and can call other types of scripts. We used it for running the cluster configuration and running Apache Storm.

3.1.3 Bash scripting

Bash is a programming language first released in 1989. It is a helpful language which is used for system administration. This tool is mainly used for automating the running of tasks and saving time. Due to the large scope of the project, we had to automate the running of Apache Storm and setting up of the system using bash scripts.

3.2 Cluster configuration

To be able to run Apache Storm over the cluster, different main commands and libraries have been used. Additionally, we need to simulate the limited network over the Raspberry Pis. To do so, we used different commands and tools, which we represent as below:

- **`ip netns`**: Netns is known as the network namespace. It gives us the ability to create a completely new container for the network, and it would act as another network inside the current network. To simulate a network with limited bandwidth and latency, we generate another network with `netns`, which behaves like a bad network.

- **`ip6table`**: To be able to forward all the messages from the real network to the virtual network created by `netns`, we use the command `ip6tables`. We changed the NAT table of the device and enabled forwarding on all the Raspberry Pis. These NAT rules and forwarding rules are added to the `ip6tables`. The destination for all incoming messages with the specific port are changed by the NAT rules and forwarded by the forwarding rules.
- **`qdisc`**: It is a command which is used for controlling network traffic. The limited bandwidth on the ingress and egress networks is applied using qdisc.
- **`netem`**: With `qdisc`, it is possible to use another command called `netem`, which is helpful to emulate additional latency over specific device network interfaces.
- **`parallel-ssh`**: This command helps to run the same command over a list of hosts. It is required to pass the hosts on a text file with the flag `-h`, and it will run the same command over all defined hosts.
- **`parallel-scp`**: This provides the ability to copy one file or folder over the defined hosts. The host file is passed with the flag `-h`, and it will copy the file or folder to all the requested hosts.
- **`iperf3`**: This library allows us to test the connections between two devices. We can create a server-side instance on one device and a client-side instance on the other device. Then we can check the connections with the specified port using the flag -p. It helps us troubleshoot connections among Raspberry Pis.

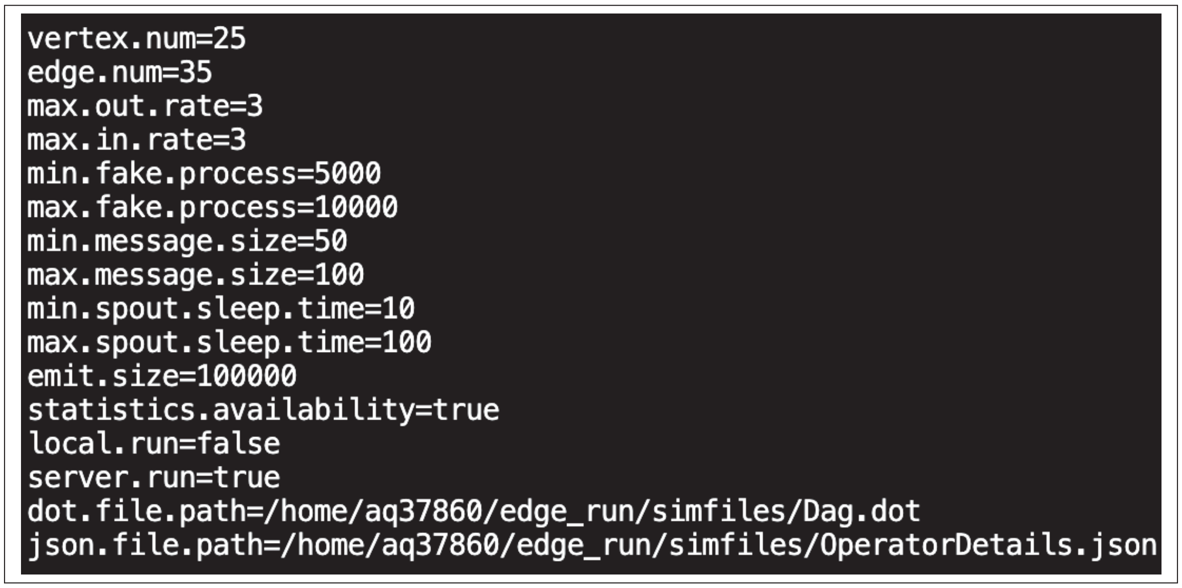
3.3 Random topology generation

As we discussed in Section 2.3, our random topology generator generates topologies which are randomly defined. The libraries used in the module are summarized as below:

- **JgraphT**: This library is used in Java for the creation of random DAGs with specific amounts of edges and vertices. We use dot exporter of JgraphT to export the dag into a file.
- **Random**: This library is used to generate random values. It is used for specifying the in rate and out rate of operators, message size, data rate and the number of processes in each operator.

- **Gson**: This library converts a Java object to a JSON file. We keep all the topology configurations in a Java object. Then we save it as a JSON file. The simulator can use this JSON file to determine topology.
- **FileWriter**: File writer is a library that is used for storing the contents as a file on the specific path of the system. We used it to store the topology configuration like JSON in the file.

Our random topology is configured using a property file, which contained all of its configurations. As it is shown in Figure 3.1, different factors need to be specified for the random generation module. It is also possible to use a seed in the random library to have the exact behaviour of the application when we want to decide on sending messages to a specific downstream operator.



```
vertex.num=25
edge.num=35
max.out.rate=3
max.in.rate=3
min.fake.process=5000
max.fake.process=10000
min.message.size=50
max.message.size=100
min.spout.sleep.time=10
max.spout.sleep.time=100
emit.size=100000
statistics.availability=true
local.run=false
server.run=true
dot.file.path=/home/aq37860/edge_run/simfiles/Dag.dot
json.file.path=/home/aq37860/edge_run/simfiles/OperatorDetails.json
```

Figure 3.1 Random generation configuration file

3.4 Profiling

In this step, we run the generated topology over the cluster of Raspberry Pis. The running was automated using bash scripts in this part. In order to generate random bandwidth values, we used Python, which generates the random bandwidth and latency for each Raspberry Pi. A *bad network* configuration (e.g., using netem) is then applied to every device based on random bandwidth and latency values.

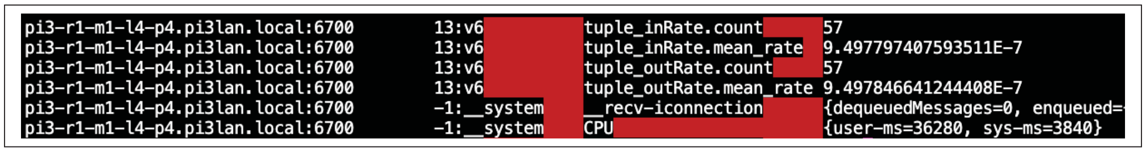
After applying the random network limitations, the Apache Storm framework uses the metric class and logs all the required metrics every 60 seconds on all the devices. The log files are collected from all Raspberry Pis, and the relevant parts are selected for analysis. Additional libraries and commands are used as follows:

- **Sigar**: To measure the CPU usage of Java applications over the resources, it is required to track the JVM using Sigar. Sigar shows us how much CPU is used every 60 seconds. It is added to Apache Storm as a custom metric. Unfortunately, Sigar was not supported by default on Raspberry Pis, which forced us to build the library from source code on the ARM architecture. We were then able to add it to the local maven repository so that the projects can use it.
- **rsync**: This command helps us to synchronize two separate folders over the two devices. It checks the changes and then copy the data. We transfer the logs of devices to one specific node for processing using this command.
- **grep**: Grep is used for searching text patterns. It is used to find relevant text strings from a specific file. The processing of log files and getting the required lines of logs are done using the grep command.

3.5 Log analyzer

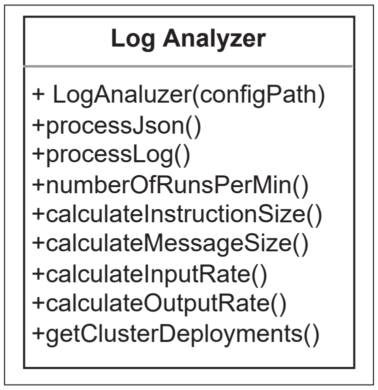
In this section, we used libraries like `Gson` and `fileWriter` to collect information about the topology. The log analyzer then starts processing the logs, which are files containing strings. Figure 3.2 shows an example of a log. By processing the logs, we can determine the incoming and outgoing data rate and message sizes, as well as the CPU usage.

We provide the Figure 3.3 which shows the exact methods in the log analyzer.



```
pi3-r1-m1-l4-p4.pi3lan.local:6700        13:v6[redacted]tuple_inRate.count[redacted]57
pi3-r1-m1-l4-p4.pi3lan.local:6700        13:v6[redacted]tuple_inRate.mean_rate[redacted]9.497797407593511E-7
pi3-r1-m1-l4-p4.pi3lan.local:6700        13:v6[redacted]tuple_outRate.count[redacted]57
pi3-r1-m1-l4-p4.pi3lan.local:6700        13:v6[redacted]tuple_outRate.mean_rate 9.497846641244408E-7
pi3-r1-m1-l4-p4.pi3lan.local:6700        -1:__system[redacted]__recv-iconnection[redacted]{dequeuedMessages=0, enqueued=
pi3-r1-m1-l4-p4.pi3lan.local:6700        -1:__system[redacted]CPU[redacted]{user-ms=36280, sys-ms=3840}
```

Figure 3.2 A log example



Log Analyzer

+ LogAnaluzer(configPath)
+processJson()
+processLog()
+numberOfRunsPerMin()
+calculateInstructionSize()
+calculateMessageSize()
+calculateInputRate()
+calculateOutputRate()
+getClusterDeployments()

Figure 3.3 Log analyzer class

3.5.1 Data rate calculations

To know the amount of messages that flow into each operator, it is critical to track the incoming and outgoing data rate. For that purpose, we use the *Storm Metrics* which is a built-in library for collecting and monitoring required metrics from Apache Storm application; more precisely we use the Meter object which has specific methods to collect data rate information such as `meanRate` and `count`. We mark an event whenever the execute method of each bolt and spout is called. This metric can be used to determine incoming data rate. Before sending a message, we mark the outgoing data rate for each generated message. In the log analyzer, we divide the total incoming or outgoing messages by the time that the operator has been executing. By doing so, we obtain the average data rate for each operator separately. To be able to use the in built metrics in Apache Storm, it is necessary to enable the `TOPOLOGY_ENABLE_V2_METRICS_TICK` configuration settings before running it.

3.5.2 CPU usage calculations

The CPU usage is calculated using the Sigar library and the metric API (second version) of Apache Storm. Their combination provide the CPU usage every 60 seconds – they provide the time that the CPU is used over 60 seconds intervals. Sigar provides the user-ms value in the log files in milliseconds. In the log analyzer, we calculate the number of times that each operator has been run, then we divide the total CPU usage over 7 minutes by this number. As a result, we have the average of required CPU usage for a one-time execution of each operator. Due to the MIPS capacity of each device, we can determine the *instruction size* for each operator. However, it is worth mentioning that in our evaluation, we have the same MIPS value for each device. Our system and its implementation nevertheless makes it possible to change the CPU power, which can be convenient if different types of edge devices are provided.

3.5.3 Message size calculations

Apache Storm has a specific configuration setting called `TOPOLOGY_SERIALIZED_MESSAGE_-SIZE_METRICS`. By enabling this metric in the `config` object, before starting the application, Apache Storm will present the amount of messages transferred between the operators in bytes. This information is saved in the log files. In our log analyzer, we process the log file and calculate the sum of data size that comes into an operator and we divide it by the total number of messages that come into the same operator. By doing so, we will have the size of each messages. The same process takes place for the outgoing message sizes.

3.6 Simulator

We implemented the simulator in Java. Similarly, the GWO is also implemented in Java and is added to the simulator in another package. Figure 3.4 shows the class communications in the simulation. GWO is used in the method `solve()`, and then GWO calls the `push()` method. As we discussed, in order to decrease the error rate, the process of running the GWO is repeated several times. So the `reset()` method helps us to repeat everything.

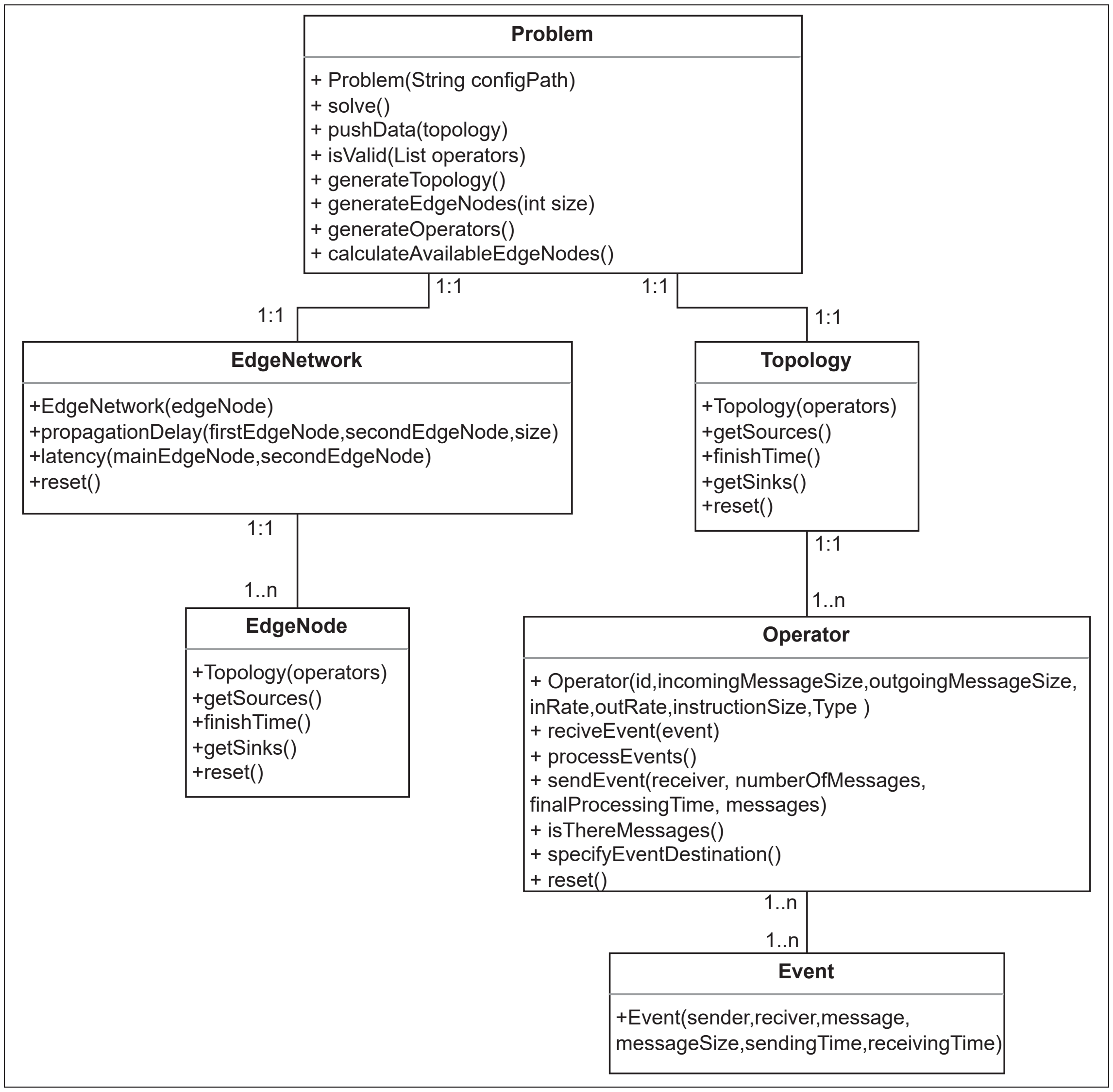
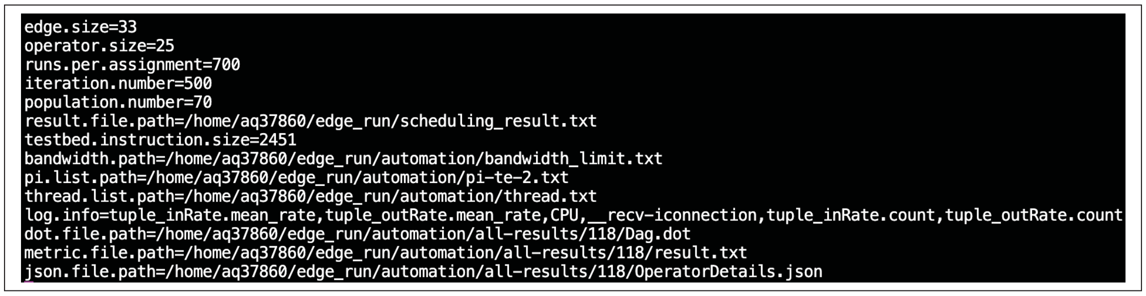

Problem
+ Problem(String configPath)
+ solve()
+ pushData(topology)
+ isValid(List operators)
+ generateTopology()
+ generateEdgeNodes(int size)
+ generateOperators()
+ calculateAvailableEdgeNodes()
1:1
1:1
1:1
1:1
EdgeNetwork
+EdgeNetwork(edgeNode)
+propagationDelay(firstEdgeNode,secondEdgeNode,size)
+latency(mainEdgeNode,secondEdgeNode)
+reset()
Topology
+Topology(operators)
+getSources()
+finishTime()
+getSinks()
+reset()
1:1
1..n
EdgeNode
+Topology(operators)
+getSources()
+finishTime()
+getSinks()
+reset()
1:1
1..n
Operator
+ Operator(id,incomingMessageSize,outgoingMessageSize,
inRate,outRate,instructionSize,Type)
+ reciveEvent(event)
+ processEvents()
+ sendEvent(receiver, numberOfMessages,
finalProcessingTime, messages)
+ isThereMessages()
+ specifyEventDestination()
+ reset()
1..n
1..n
Event
+Event(sender,reciver,message,
messageSize,sendingTime,receivingTime)

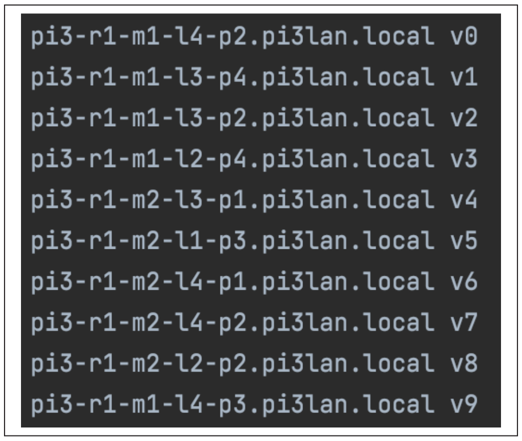
Figure 3.4 Simulator class connections

Different parameters must be specified for the execution the simulator. The configuration parameters are presented in Figure 3.5. At the end, the simulator outputs a text file containing the name of each task and the edge nodes to which they are mapped. An example of a result file is shown in Figure 3.6



```
edge.size=33
operator.size=25
runs.per.assignment=700
iteration.number=500
population.number=70
result.file.path=/home/aq37860/edge_run/scheduling_result.txt
testbed.instruction.size=2451
bandwidth.path=/home/aq37860/edge_run/automation/bandwidth_limit.txt
pi.list.path=/home/aq37860/edge_run/automation/pi-te-2.txt
thread.list.path=/home/aq37860/edge_run/automation/thread.txt
log.info=tuple_inRate.mean_rate,tuple_outRate.mean_rate,CPU,__recv-iconnection,tuple_inRate.count,tuple_outRate.count
dot.file.path=/home/aq37860/edge_run/automation/all-results/118/Dag.dot
metric.file.path=/home/aq37860/edge_run/automation/all-results/118/result.txt
json.file.path=/home/aq37860/edge_run/automation/all-results/118/OperatorDetails.json
```

Figure 3.5 Simulator configuration file



```
pi3-r1-m1-l4-p2.pi3lan.local v0
pi3-r1-m1-l3-p4.pi3lan.local v1
pi3-r1-m1-l3-p2.pi3lan.local v2
pi3-r1-m1-l2-p4.pi3lan.local v3
pi3-r1-m2-l3-p1.pi3lan.local v4
pi3-r1-m2-l1-p3.pi3lan.local v5
pi3-r1-m2-l4-p1.pi3lan.local v6
pi3-r1-m2-l4-p2.pi3lan.local v7
pi3-r1-m2-l2-p2.pi3lan.local v8
pi3-r1-m1-l4-p3.pi3lan.local v9
```

Figure 3.6 Simulator placement result

3.7 DAG deployments

The last section to run is to deploy Apache Storm with the same configuration but with the updated scheduler over the Raspberry Pis.

3.7.1 Apache Storm custom scheduler

To be able to apply the generated mapping to Apache Storm, it is necessary to add a custom scheduler that reads the deployment from the file and then deploys the mapping to the nimbus assignments. So the custom scheduler checks the available supervisors and then assigns each task (executor) to an available worker slot of the supervisor.

To enable the use of the new scheduler, we use Maven with command `mvn package` to package the scheduler, and then the jar file is copied into the bin directory of Apache Storm. We then add the new scheduler to the Storm configuration file (`storm.yaml`).

CHAPTER 4

EVALUATION

In this chapter, we present the results gathered from our evaluation. In Section 4.1, we review the available resource infrastructure for running the tests. In Section 4.2, the configurations and calculation methods for evaluating the approaches are reviewed. Section 4.3 explains the different approaches used for the comparison. Section 4.4, the main test cases are presented. In Section 4.5 we review the real deployment results. In section 4.6, we present the results provided by our simulator. In the end, we analyze the results in Section 4.7. To test our approach, we use the Apache Storm framework because of its flexibility in configuring the applications.

4.1 Testbed

As part of this thesis, we use a real infrastructure to evaluate our methodology. All the generated random DAGs are deployed onto a real test-bed. Since we are running complex programs, a testbed with a lot of IoT resources is required. To that end, we use a testbed including 35 Raspberry Pi 3, Arm Cortex-A53 (4 CPUs @ 1.2 GHz, 4 Cores), 1 GB Ram with the Ubuntu 18.04.5 LTS operating system. Apart from the Raspberry Pis, a ThunderX CN8890 cluster (96 CPUs @ 1.9 GHz, 48 Cores of Arm architecture) with 128 GB of RAM and Ubuntu 18.04.5 LTS, 64-bits, is provided. The cluster acts as a cloud server for us. These infrastructures can provide valuable information regarding simulation and real deployments.

4.2 Methodology

This section discusses the various metrics that we use to evaluate our methodology. Different parameters affect the deployment environment, and they need to be defined. Then, the *Random Problem Generator* can use these configurations to create a random DAG application. In other words, we first need to define these metrics, and then the analyzer can process the logs and collect the required information.

4.2.1 Randomized CPU

In order to model the CPU capacity of a given operator, we used a fake random loop in the operator itself, which randomly generates and increments a number, and which consumes CPU usage onto the target device. A range is defined for the number of loops that need to be run. We choose a random number in the specified range as the number of loops for each operator separately.

4.2.2 Randomized data rate

We define the maximum in-rate and maximum out-rate of operators as random numbers. 80% of the operators have the in-rate and out-rate set to one, and we assign values greater than one and lower than the maximum defined values for 20% of the operators. Another factor is sleep time which affects the data rate. Source operators *sleep* based on this value, which causes a delay in sending the messages and changes the incoming data rate in the DAG.

4.2.3 Randomized probability ratio

Operators might have a different number of downstream operators. In the experiments, we assign each downstream operator a weight between 0 and 1. These random numbers indicate the weight for each downstream operator. When upstream operators want to send messages, they choose a random downstream operator based on their weight.

4.2.4 Randomized message size

Another configuration for the experiment is the message size. We define fake messages by generating a string that contains a certain amount of characters. We consider each character as two-byte data. For each operator, a random message size is assigned according to the defined message size range. Each operator then creates a message with the specific string size using a for loop.

4.2.5 Number of repeats

We need to specify the number of times that the simulation process will be repeated. The decision for sending the messages to downstream operators is random, and is based on the probability ratio. As a result, certain paths do not transfer messages, especially in a complex application. It is possible to reduce this *risk* by running the simulator multiple times. By increasing this value in the configuration, the accuracy of the simulation process increases. We can infer that we can cover more paths in the DAG by executing the simulator multiple times.

4.2.6 Meta-heuristics metrics

As part of the evaluation, we define variables such as the number of populations and the number of iterations in the meta-heuristic algorithm. It informs the meta-heuristic algorithm how deep it should search for a locally-optimized configuration in the search space. These variables are defined using empirical observation. We perform some pre-tests until we find a probable value for these variables.

4.2.7 Randomized bandwidth

A network limitation is applied to the test environment through a file containing the bandwidth and latency values. These values are chosen randomly according to the bandwidth and latency range provided. Before running each test, we prepare the environment according to this configuration file. The incoming and outgoing bandwidth, and the latency are defined separately. The bandwidth is defined in Mbits per second and the latency is defined in milliseconds. We reset the environment after running each test case.

4.2.8 DAG latency calculations

Calculating the latency in Apache Storm is done by writing the *message timings* in the logs. To put it simply, the time at which a given message is created is stored along with the message ID. This information is passed on to the next bolt until it reaches the sinks. When the sinks receive a string message, we add the end time to its message. Then we write them in the log file. We then process the logs using a Python script and make an average of all the start and end times. As we discussed, one message can be received into multiple sinks. Therefore, in some sinks, it is possible to have one end time for multiple start times. The most tricky part of calculating the DAG latency is the accuracy. Due to the fact that the source and sink are deployed on different devices, the time is not perfectly synchronized. So we use an NTP server that synchronizes the time on the devices.

4.3 Comparison against other approaches

It is necessary to determine the performance of the proposed methodology. To that end, we compare the results with other approaches. In this thesis, we compare our result with two alternative approaches: cloud-only and random.

4.3.1 Default Storm scheduler

In the random approach, the placement happens completely randomly. We use the Apache Storm default scheduler for the comparison. It deploys the operators over the available supervisors using a round-robin approach. By comparing our results to this approach, we can determine how good the results are. We consider this approach as a baseline approach.

4.3.2 Cloud only

Another strategy for comparing the results is using the cloud instead of the edge for the deployment of the DAG. It means that all the source and sinks operators will be deployed in the edge layer, and all the transformation operators are run over the cloud resources. However, the communications between the operators at the edge layer and the cloud layer would be slower because of higher latency. But the bandwidth ranges would be the same as for the EdgeDAG deployment and the default Storm scheduler deployment to have a similar environment for testing the approaches. We run the transformation operators over the cluster, we then manually deploy all the sources and sinks on the Raspberry Pis, and we add the latency value to the network. We consider a 45 ms incoming latency and a 45 ms outgoing latency to/from the cloud, so a total latency of 90 ms is considered for the cloud-edge connections.

4.4 Test cases

In this thesis, we test our proposed methodology using six test cases, which are randomly generated DAGs. These DAGs act as time-sensitive applications. Our test cases can be categorized according to the two main factors which affect the applications: application size and operator dependencies. We review them in the following sections.

4.4.1 Application size

We divide the defined test cases based on the DAG size of the applications (number of operators in the DAG – small size, medium size and large size). By doing so, we can test our approach for different applications.

4.4.2 Operator dependencies

Dependencies between tasks are another factor that affect the applications. We consider two types: high dependency and low dependency, which impact the number of edges in the DAG (connectivity). In other words, we define different *densities* for the DAGs.

Figures 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6 show all the DAGs which are used in the evaluation section. The number of edges and vertices are chosen based on our understanding of different application types. We show the result for each DAG separately in the results section.

We define the base test cases according to Tables 4.1, 4.2 and 4.3. The latency and bandwidth are stable for the each test case, and they are completely random, but we need to change the latency in the network for the cloud tests environment. For running the complex DAGs, we increase the number of iterations and populations, because the search space gets increasingly large. Similarly,

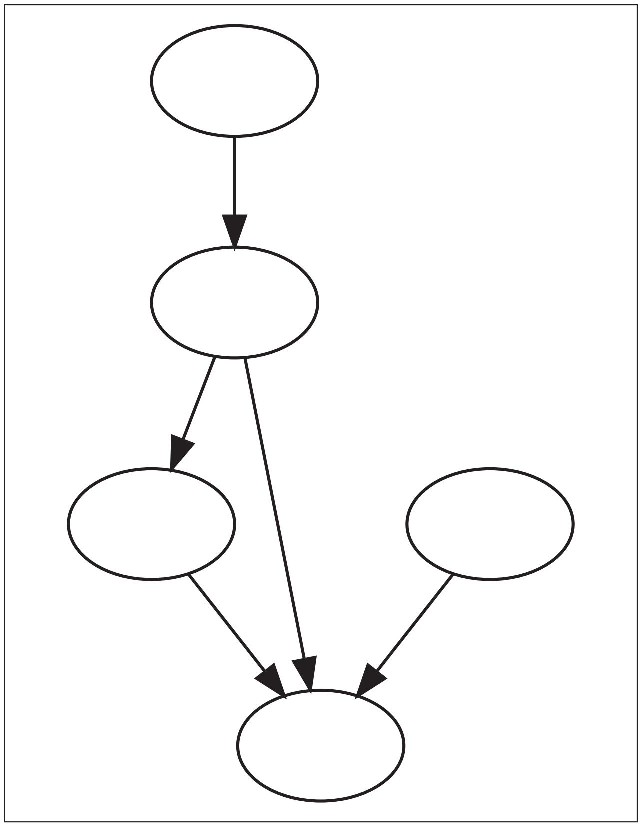


Figure 4.1 Random generated DAG one: 5 vertices, 5 edges

Table 4.1 Test case configurations (1)

Test-id	Operators	Edges	Max. out-rate	Max. in-rate	Fake process range
1	5	5	3	3	8000-12000 loops
2	5	10	3	3	8000-12000 loops
3	15	20	3	3	8000-12000 loops
4	15	45	3	3	8000-12000 loops
5	25	35	3	3	8000-12000 loops
6	25	60	3	3	8000-12000 loops

we increase the number of repetitions for the complex DAGs because we want the simulator to walk through all the paths. The fake processing loop (used to simulate the processing at the operators) is defined between 8000 and 12000, and it causes different processing latency in each run for the operators. Sleep time also causes a data rate of 10 messages per second to 50

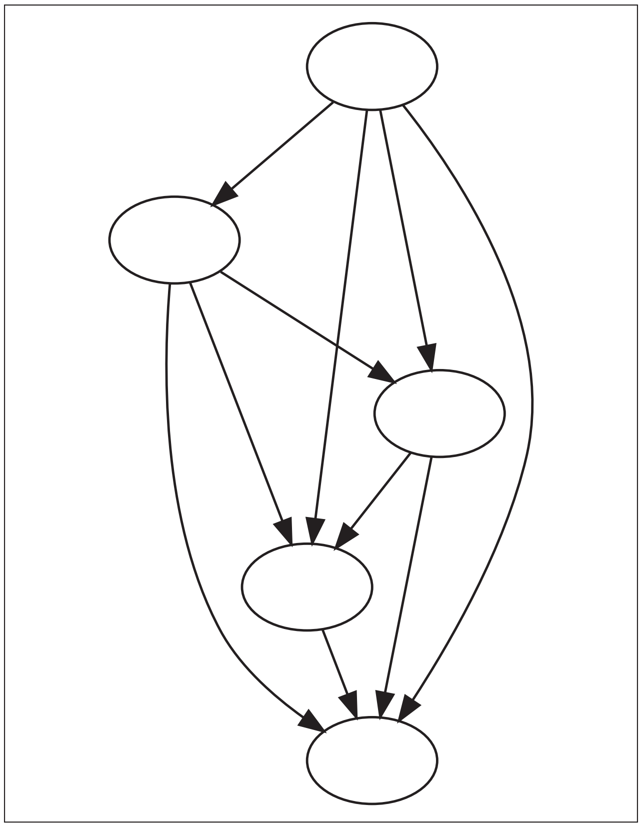


Figure 4.2 Random generated DAG two: 5 vertices, 10 edges

Table 4.2 Test case configurations (2)

Test-id	**Message size**	**Sleep time**	**Bandwidth**	**Latency**
1	15000-25000 B	20-100 ms	1-50 Mbit/sec	2-30 ms
2	15000-25000 B	20-100 ms	1-50 Mbit/sec	2-30 ms
3	15000-25000 B	20-100 ms	1-50 Mbit/sec	2-30 ms
4	15000-25000 B	20-100 ms	1-50 Mbit/sec	2-30 ms
5	15000-25000 B	20-100 ms	1-50 Mbit/sec	2-30 ms
6	15000-25000 B	20-100 ms	1-50 Mbit/sec	2-30 ms

messages per second in the sources. The defined bandwidth range in Table 4.2 is for both the incoming and outgoing bandwidth. The latency range is described for each edge node. The incoming and outgoing network latency varies from one milliseconds to thirty milliseconds for

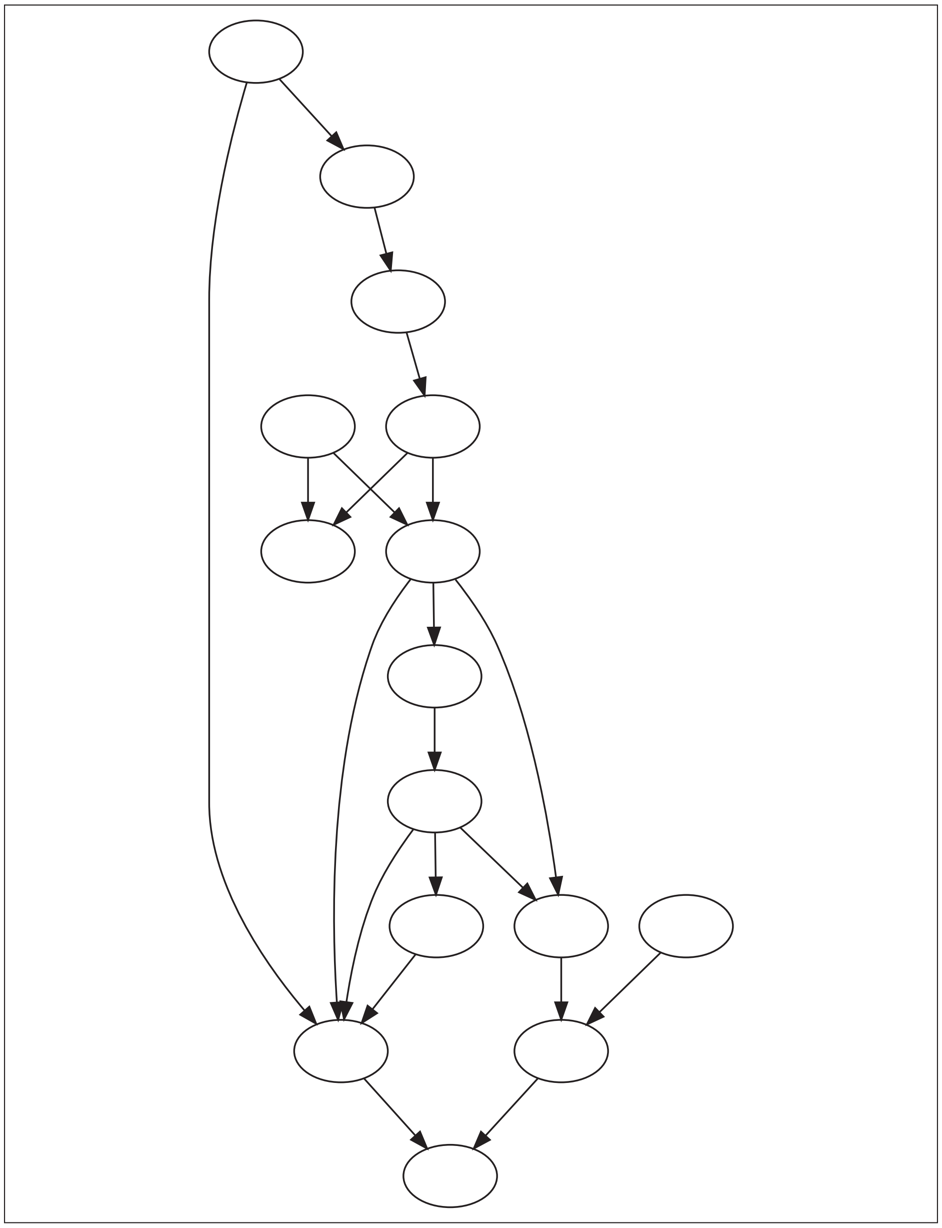


Figure 4.3 Random generated DAG three: 15 vertices, 20 edges

each edge node. For the base experiment, we assumed the maximum in-rate and out-rate of three for the operators.

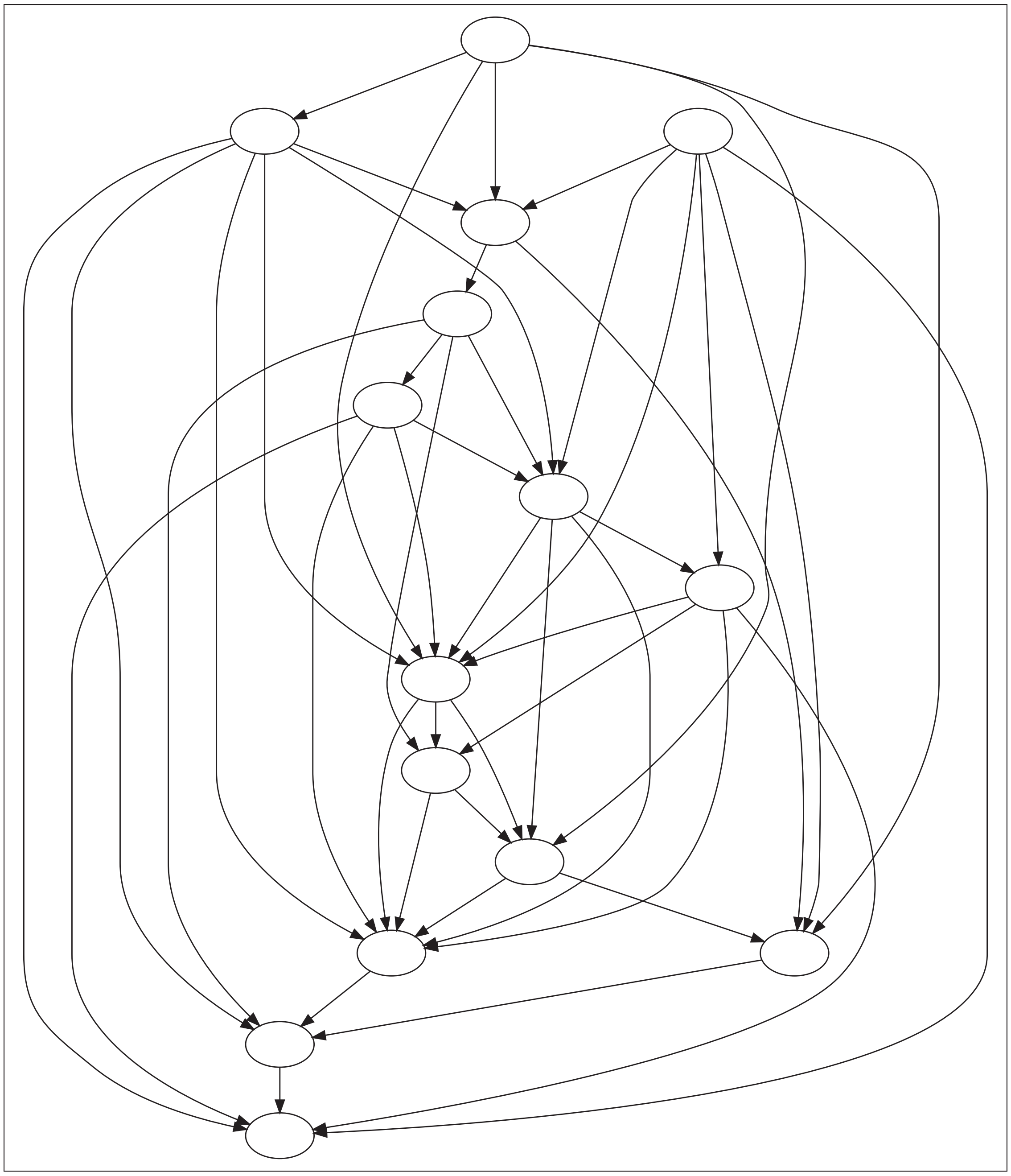


Figure 4.4 Random generated DAG four: 15 vertices, 45 edges

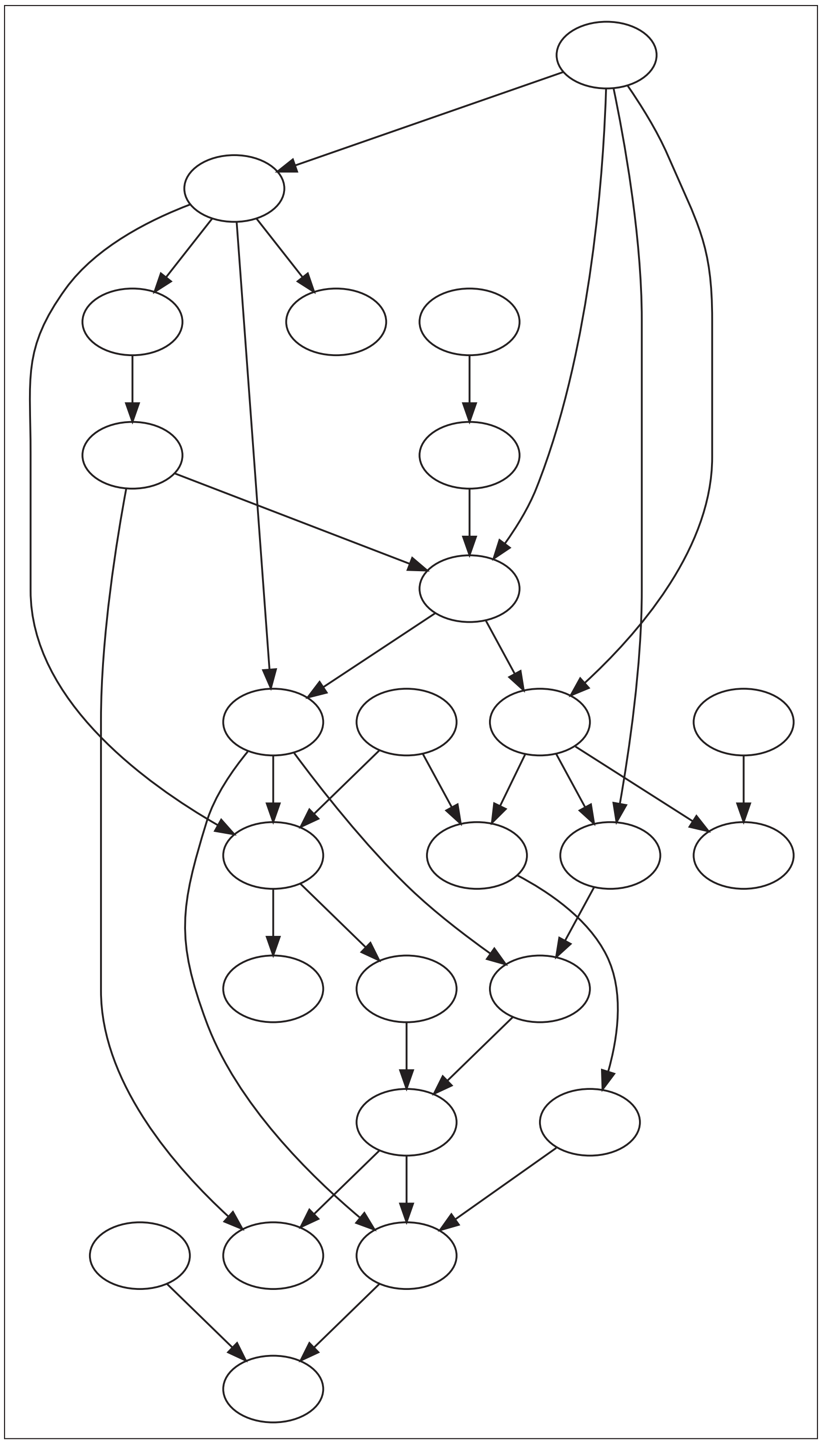


Figure 4.5 Random generated DAG five: 25 vertices, 35 edges

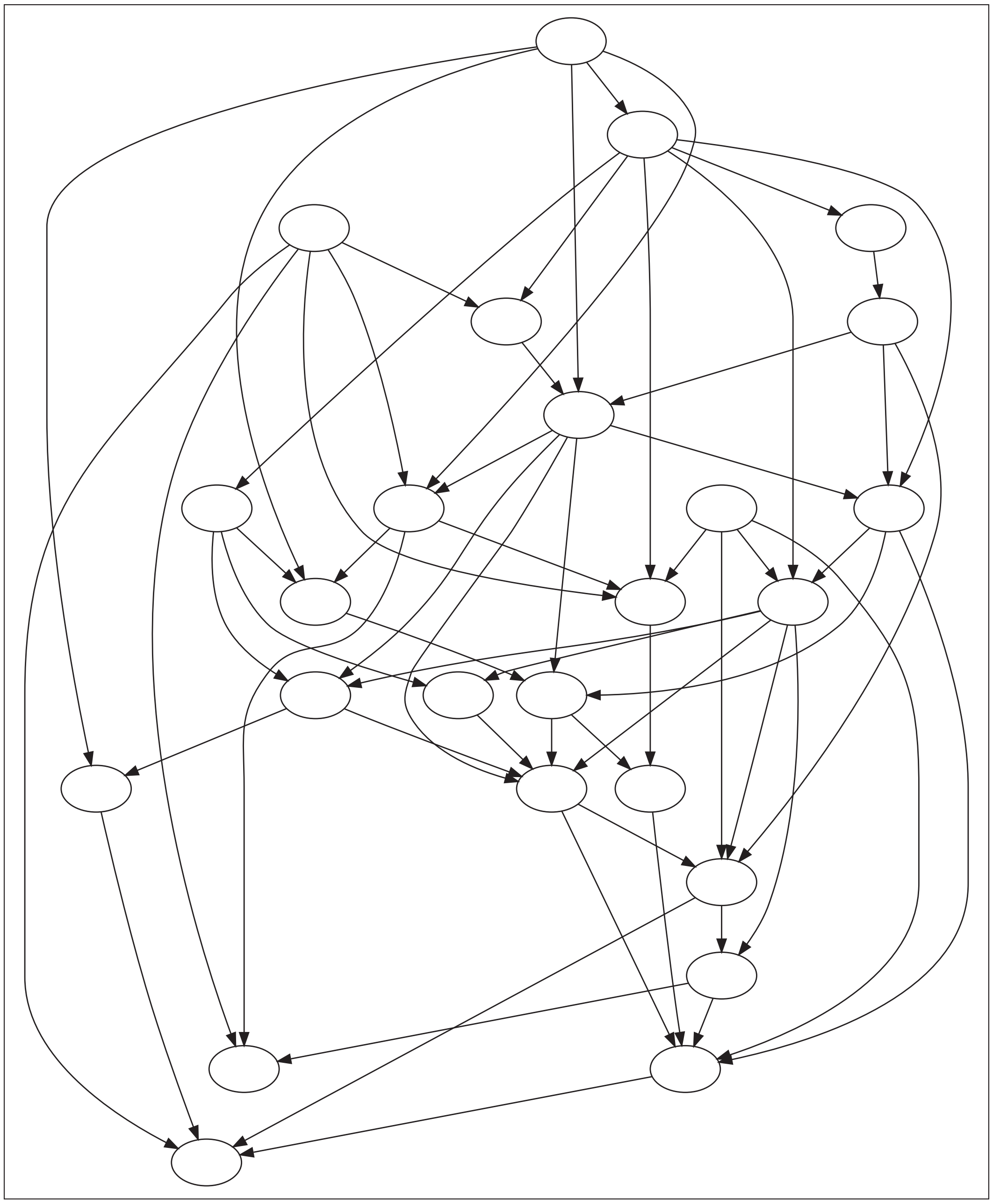


Figure 4.6 Random generated DAG six: 25 vertices, 60 edges

4.5 Deployment results

We define six experiments that consider the most important factors that affect the test environment. These experiments can help us to validate our approach in regards to the changes that can happen. We run each experiment three times to ensure that the results are consistent.

Table 4.3 GWO configurations for test cases

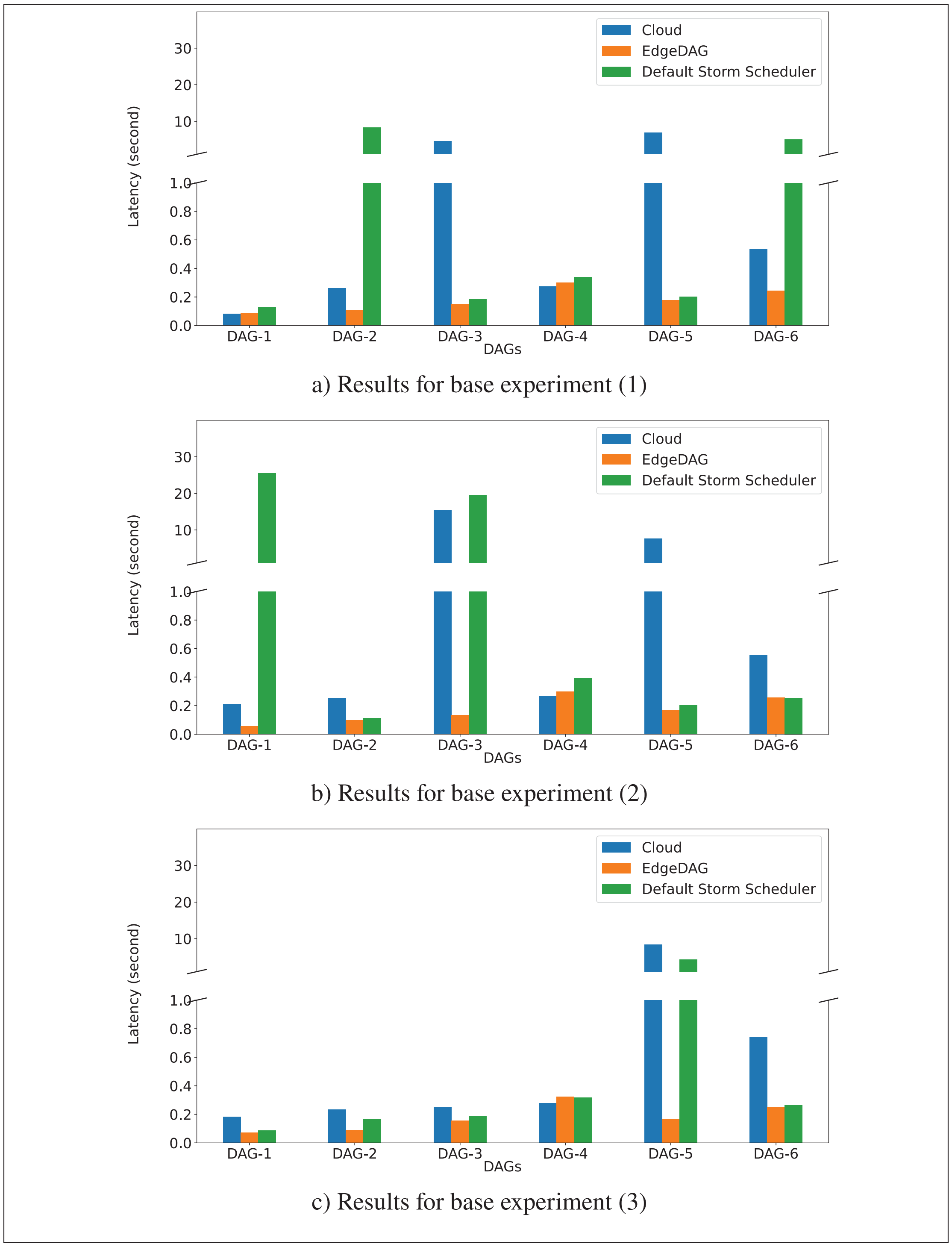
Test-id	Number of iteration	Number of population	Number of repeats
1	500	70	700
2	500	70	700
3	500	70	700
4	700	200	1000
5	700	200	1000
6	700	200	1000

4.5.1 Experiment one: Base experiment

We defined our base experiment according to the configurations in Tables 4.1, 4.2 and 4.3. Results are presented in Figure 4.7. According to the results of the base experiments, we can infer that in most cases, EdgeDAG can provide a good placement for the Apache Storm framework. More precisely, in 72% of the cases, EdgeDAG finds a better placement in comparison with the cloud and default Storm scheduler, in 88% we beat the default Storm scheduler, and in 77% of cases, we obtain better results compared to the cloud deployment. As it is observable in Figure 4.7, the default Storm scheduler and the cloud deployment scenarios can sometimes generate very high latencies due to the network limitations they face on the devices, while EdgeDAG does not provide any placement that causes any *very high* latencies (more than one second).

4.5.2 Experiment two: Decreasing the bandwidth range

In this experiment, we decrease the bandwidth range for the edge nodes. We consider a range of 1 Mbits per second to 30 Mbits per second, so that we can observe how the changes in bandwidth can affect the results. The results of the three runs are observable in Figure 4.8. Because of the limited bandwidth, it takes more time to transfer the messages between the edge nodes. As a result, the latency is higher in contrast with the base experiment. The results show that in 83% of cases, we have better results compared to the cloud deployment and the default Storm scheduler, in 88% of cases, better results are provided in comparison with the default Storm scheduler, and in 94% of cases, our approach beats the cloud deployment.

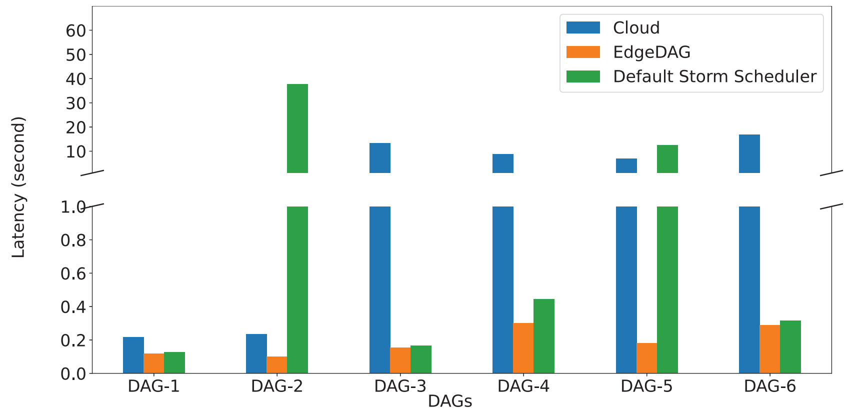


a) Results for base experiment (1)

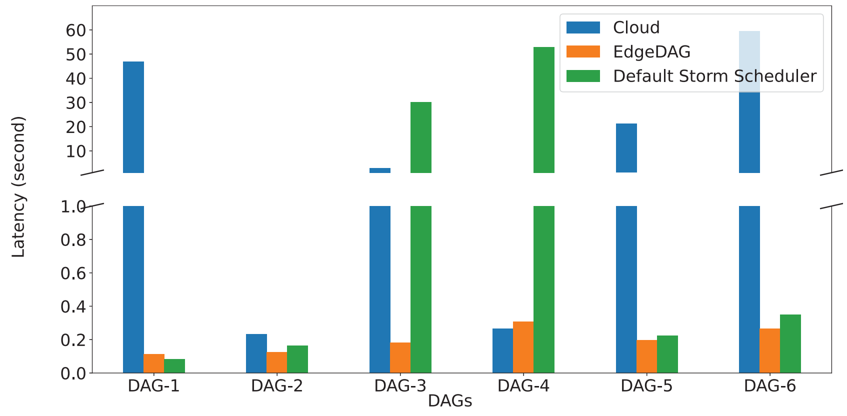
b) Results for base experiment (2)

c) Results for base experiment (3)

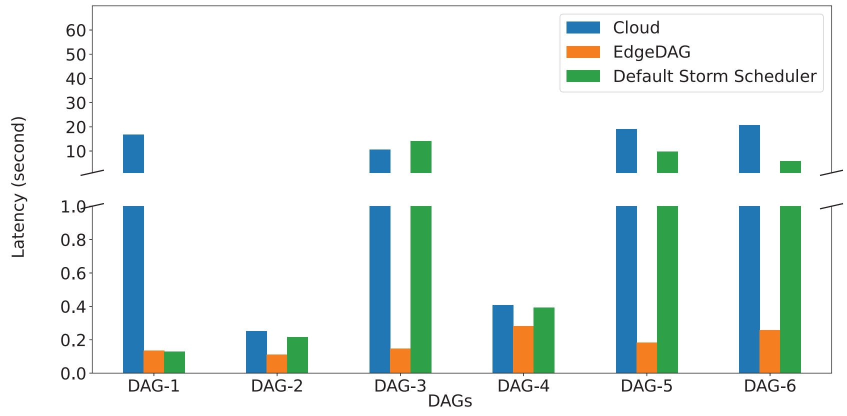
Figure 4.7 Results for base experiment (experiment one)



a) Results for decreasing the bandwidth experiment (1)



b) Results for decreasing the bandwidth experiment (2)



c) Results for decreasing the bandwidth experiment (3)

Figure 4.8 Results for decreasing the bandwidth experiment (experiment two)

4.5.3 Experiment three: Increasing in-rate and out-rate

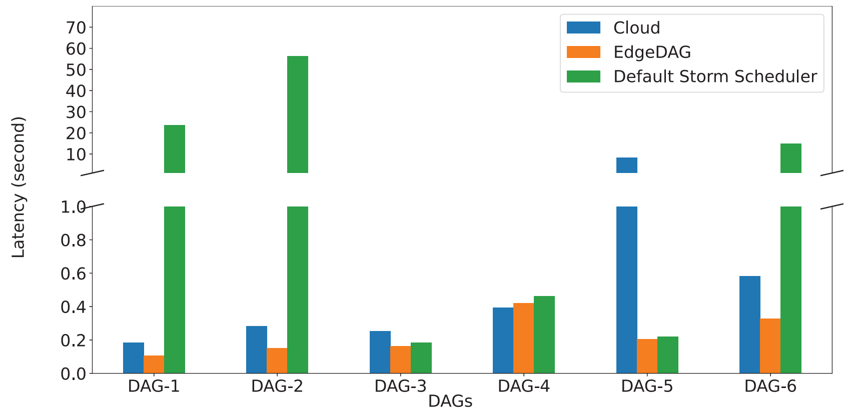
In this experiment, we increase the in-rate and out-rate for the operators – we allow a maximum in-rate and out-rate of 5 (unlike 3 in the base experiment). It means that operators need to wait for more messages, and they create more messages after processing. As we can observe (Figure 4.9), EdgeDAG provides a better placement in 77% of cases compared to cloud and default Storm scheduler. 94% of times, better results are provided in comparison to the default Storm scheduler, and in 83% of cases, EdgeDAG finds better placement rather than the cloud deployment.

4.5.4 Experiment four: Increasing the latency

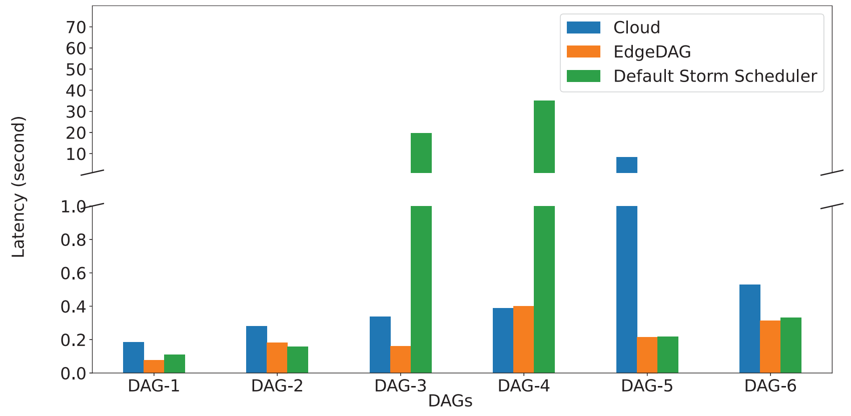
By increasing the latency, we have more pressure on the network. In this test, the latency range is between 1 ms and 30 ms for both the incoming latency and outgoing latency. The results are shown in Figure 4.10. In 66% of cases, our approach beats the other two methodologies. In 83 % of cases, our approach performs better than the default Storm scheduler, and in 77% of cases, it outperforms the cloud deployment.

4.5.5 Experiment five: Reducing the message size

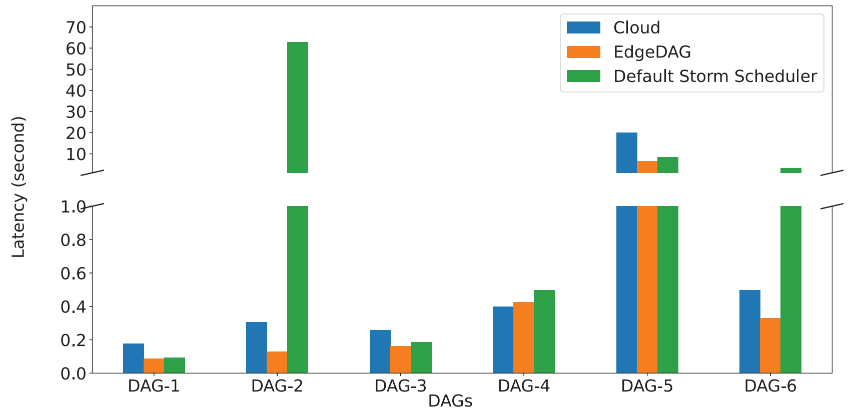
In this experiment, we decrease the size of the messages emitted by the operators (10000 bytes to 20000 bytes). We can infer that the network is more available for sending the messages. Figure 4.11 illustrates how it can affect the applications. As it is clear, the number of very high latency (more than one second) is lower in comparison with the other experiments, as the throughput is lower. As a result, the effects of buffering are decreased and we do not have very high latencies. It is worth mentioning that buffering is a very important factor and it can become problematic in other experiments (e.g., experiment four (increasing latency), experiment two (reducing bandwidth)). In this experiment, EdgeDAG provides better results in 88% of the cases both the cloud and default Storm scheduler. Our approach also beats the cloud deployment in



a) Results for increasing the in-rate and out-rate experiment (1)

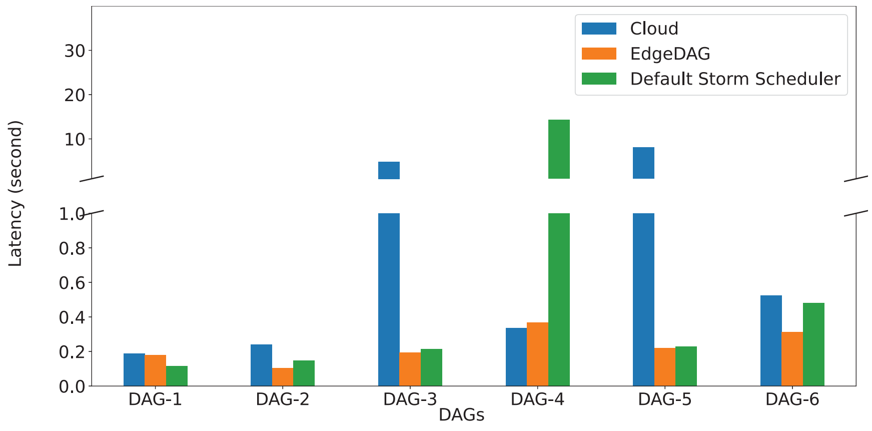


b) Results for increasing the in-rate and out-rate experiment (2)

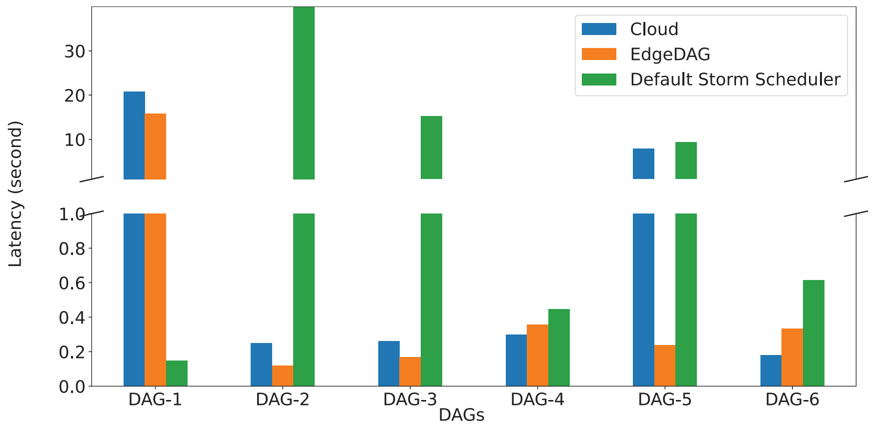


c) Results for increasing the in-rate and out-rate experiment (3)

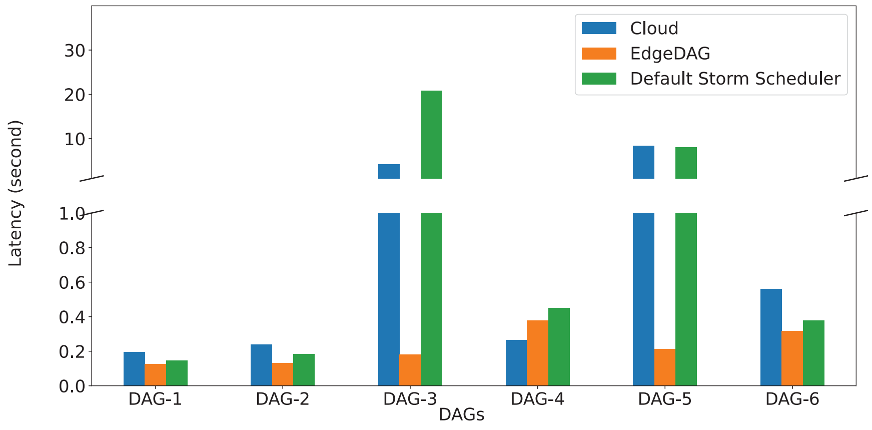
Figure 4.9 Results for increasing the in-rate and out-rate experiment (experiment three)



a) Results for increasing latency experiment (1)



b) Results for increasing latency experiment (2)



c) Results for increasing latency experiment (3)

Figure 4.10 Results for increasing latency experiment (experiment four)

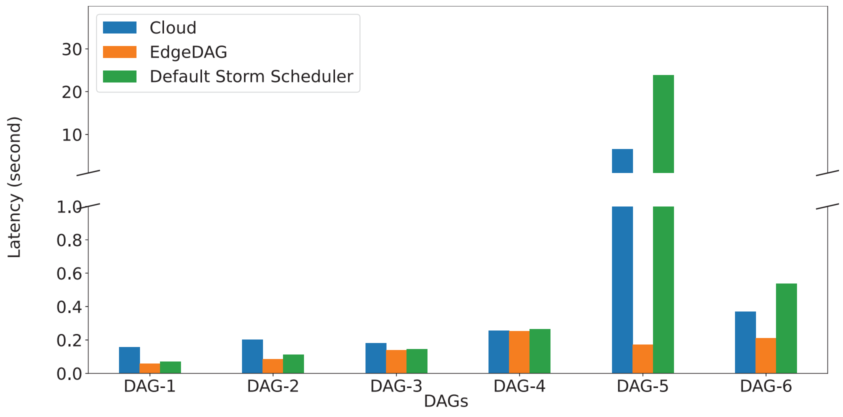
94% of tests. Similarly, better results are obtained in 94% of tests in comparison to the default Storm scheduler.

4.5.6 Experiment six: Reducing the CPU usage

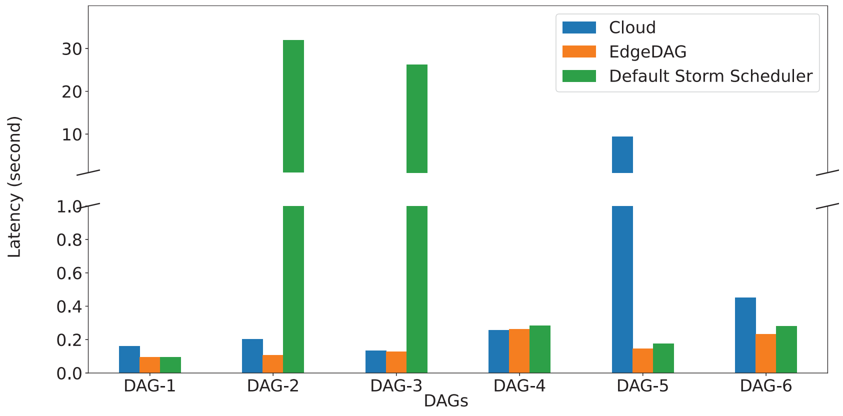
By reducing the CPU usage of the operators, they can process the messages and emit new messages faster. The fake processing range is between 5000 loops to 10000 loops in this test. It is worth mentioning that we cannot increase the CPU usage, because the CPU constraint of the devices would not be met in most cases. In other words, if we increase the CPU usage, it will not be possible for EdgeDAG to generate a placement that will respect the capabilities of the edge devices. Figure 4.12 shows the results – in 88% of the tests, EdgeDAG beats both the default Storm scheduler deployment and the cloud deployment. Our approach outperforms the default Storm scheduler in 94% of cases. Moreover, it beats cloud deployment in 94% of cases.

4.6 Simulation results

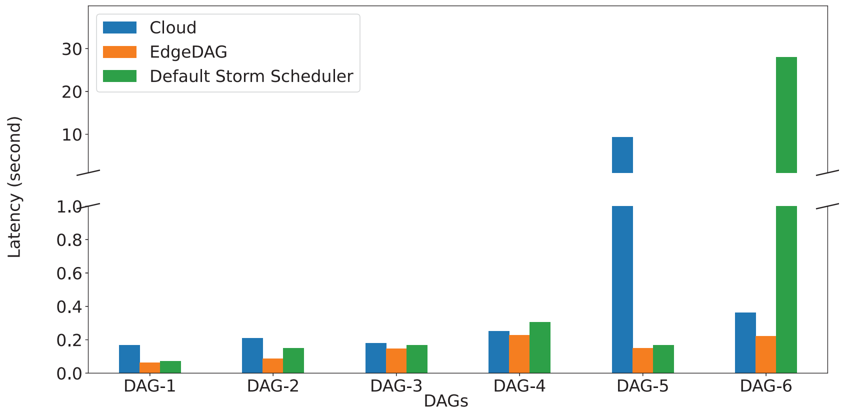
In this section, we present the latency calculated by our simulator. We refer to it as the *predicted* latency (by the simulator). We present the predicted latency for the default Storm scheduler and the best placement found by our simulator. Two comparisons are shown in the predicted latency. The first one is the best mapping which is the predicted latency of placement which is offered by our simulator (good placement). The second one is the predicted latency for the default Storm scheduler (latency prediction of our simulator for the placement of the default Storm scheduler). The results and findings for each experiment are summarized in the next sections. The experiments in this section are same as the experiments of the real deployment. We have six experiments and we run each one three times.



a) Results for reducing the message size experiment (1)

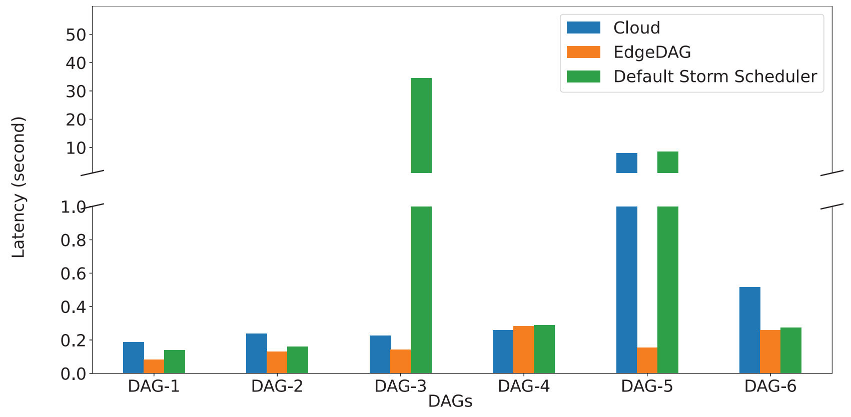


b) Results for reducing the message size experiment (2)

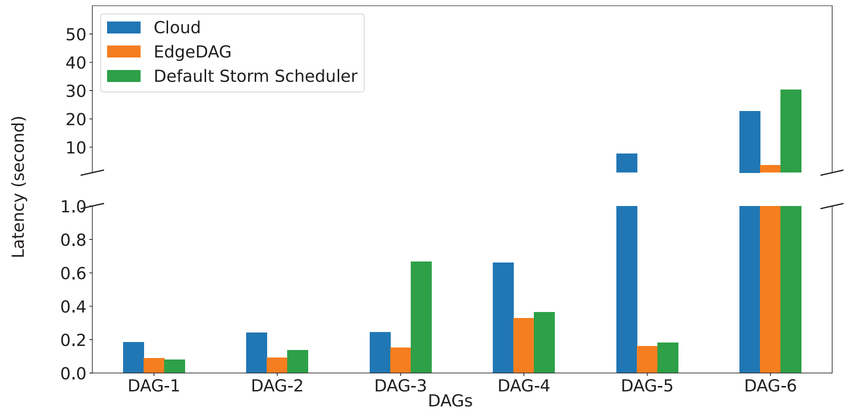


c) Results for reducing the message size experiment (3)

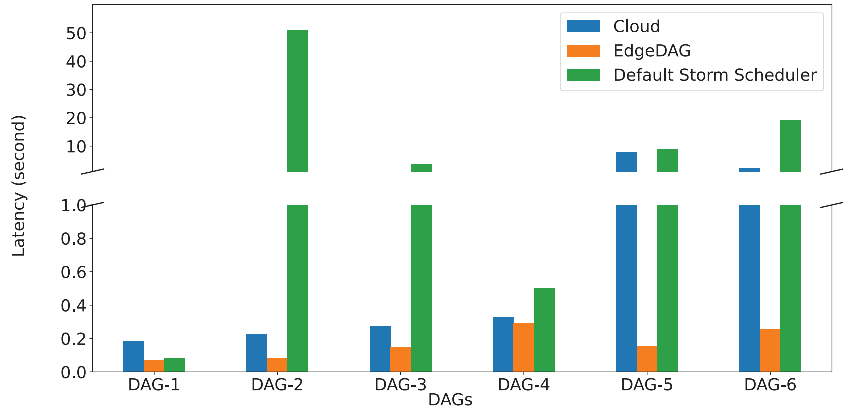
Figure 4.11 Results for reducing the message size experiment (experiment five)



a) Results for reducing the CPU usage experiment (1)



b) Results for reducing the CPU usage experiment (2)



c) Results for reducing the CPU usage experiment (3)

Figure 4.12 Results for reducing the CPU usage experiment (experiment six)

4.6.1 Experiment one: Base experiment

This experiment is the default experiment according to the test cases in section 4.4. Figure 4.13 shows how our simulator could find a better placement in contrast with the default scheduler of Apache Storm. We find that EdgeDAG provides a better placement in 100% of cases.

4.6.2 Experiment two: Decreasing the bandwidth range

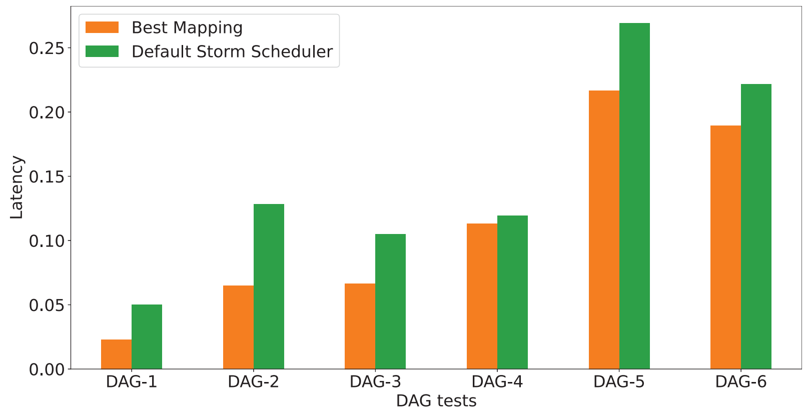
In this experiment, the bandwidth range is reduced (between 1 Mbits per second and 30 Mbits per second). We present the results for the simulator prediction in Figure 4.14. We can deduce that in 100% of the cases, our simulator offers a better plan for the deployment. In the first part of Figure 4.14, our simulator could not predict the latency for the default Storm scheduler since the placement is invalid for our simulator. A placement is invalid if the constraints are not met.

4.6.3 Experiment three: Increasing the in-rate and out-rate

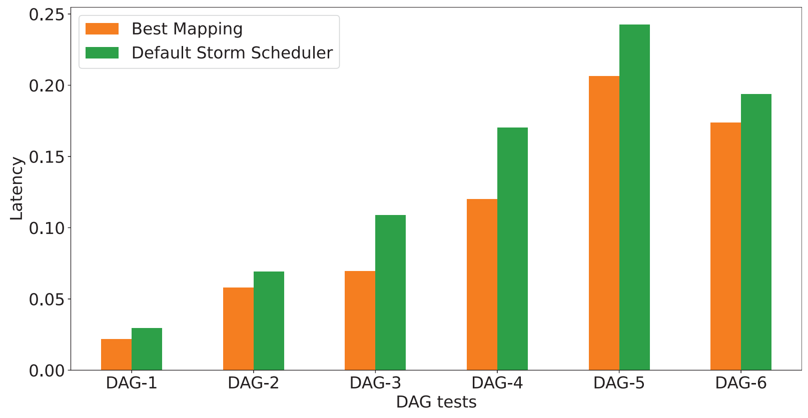
By increasing the in-rate and out-rate, the complexity of the DAG is increased. So we change the in-rate range and out-rate range of each operator to the range of 1 to 5 (in the base experiment, the range is 1 to 3). The simulator prediction is presented in Figure 4.15. In two cases, our simulator could not find a better placement (DAG-3 in part one and DAG-5 in part two of the Figure 4.15). We expect that by increasing the in-rate and out-rate, the complexity increases. As a result, it would be hard for the simulator to find a better placement. By increasing the number of iterations and population in GWO, we expect a better placement for our simulator. In this test, the simulator predicts a better placement in 88% of cases.

4.6.4 Experiment four: Increasing the latency

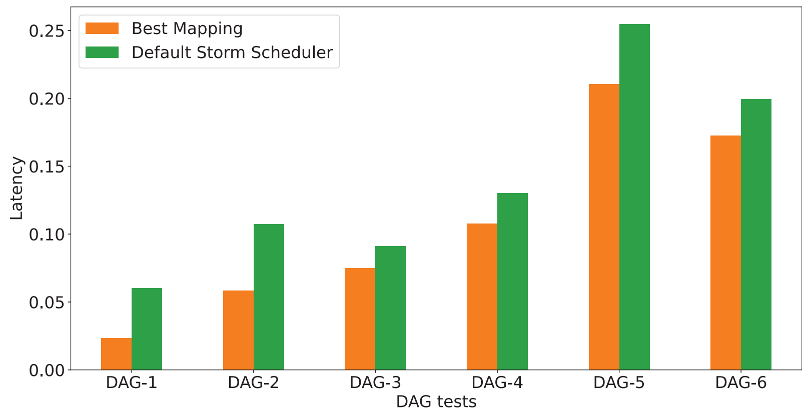
In this experiment, the incoming latency range and outgoing latency range are between 1 ms and 30 ms. By a cursory glance at the figure 4.16, we can figure out that our simulator is always predicting a better placement rather than the default Storm scheduler (in 100% of the cases). In



a) Results for simulator prediction (1)

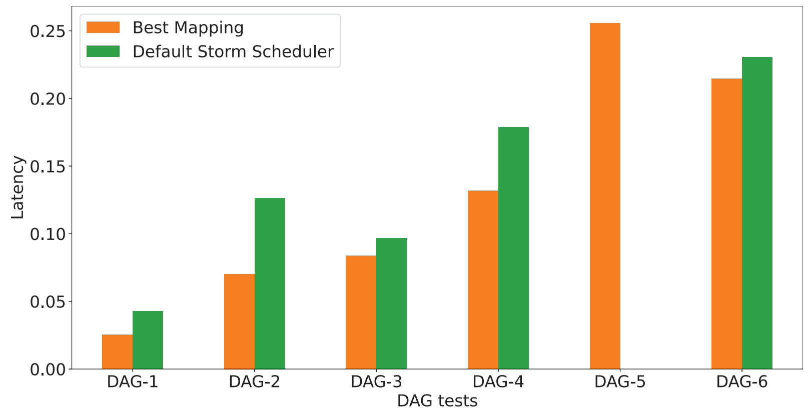


b) Results for simulator prediction (2)

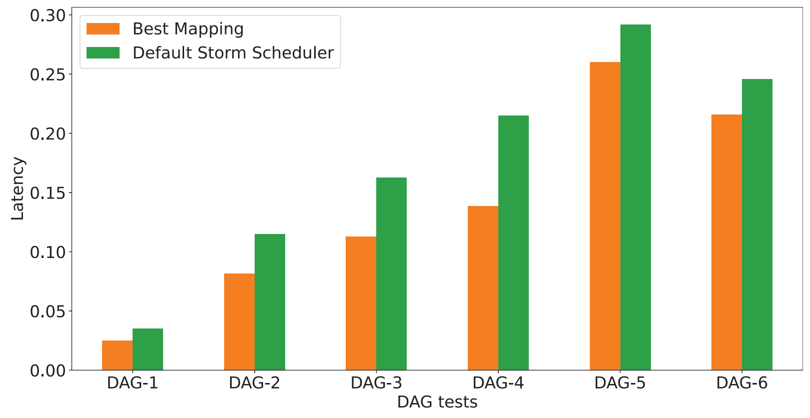


c) Results for simulator prediction (3)

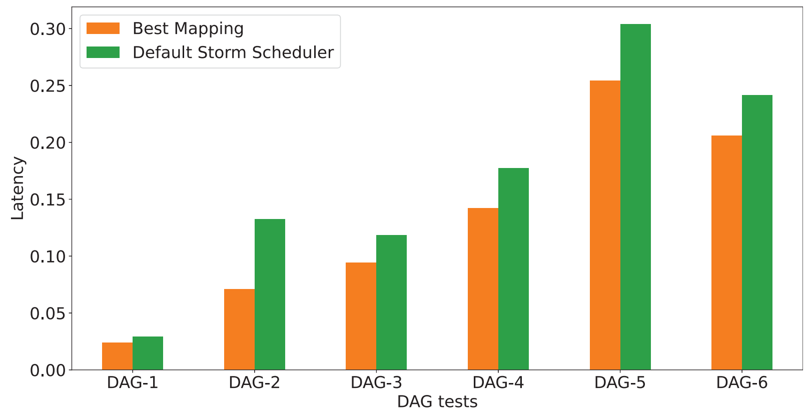
Figure 4.13   Results for simulator prediction for base experiment (experiment one)



a) Results for simulator prediction (1)

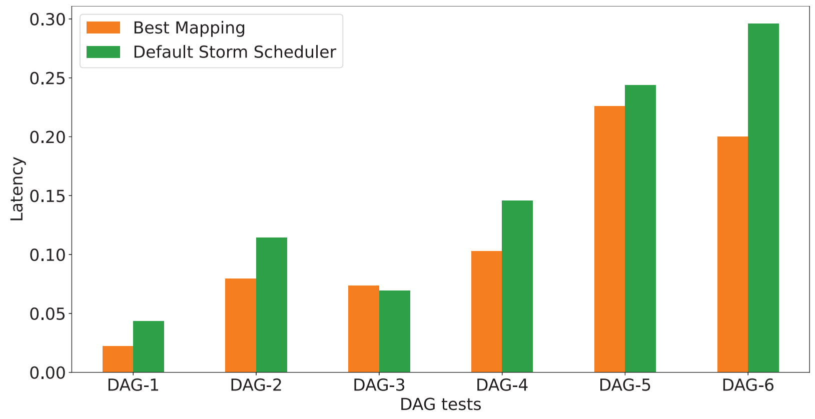


b) Results for simulator prediction (2)

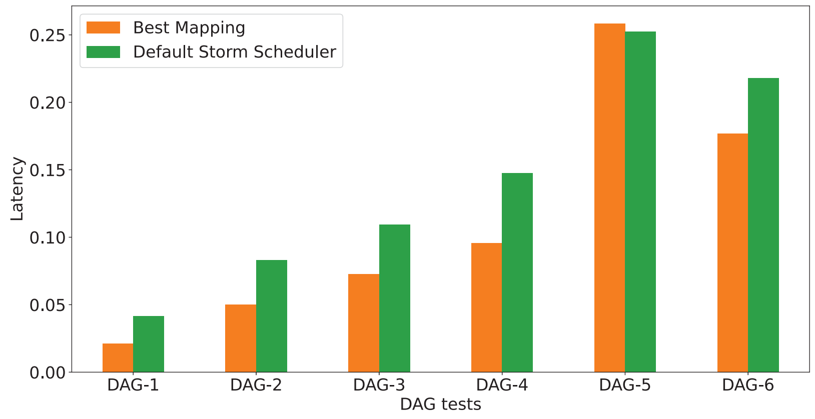


c) Results for simulator prediction (3)

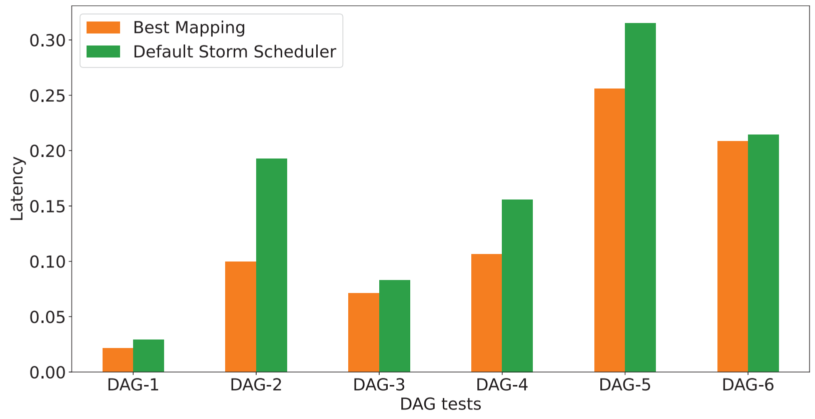
Figure 4.14 Results for simulator prediction for lower bandwidth experiment (experiment two)



a) Results for simulator prediction (1)



b) Results for simulator prediction (2)



c) Results for simulator prediction (3)

Figure 4.15 Results for simulator prediction for increasing the in-rate and out-rate experiment (experiment three)

the second section of Figure 4.16, our simulator does not provide the prediction for the default Storm scheduler in DAG-3 since the bandwidth constraint is not met in such deployment.

4.6.5 Experiment five: Reducing the message size

In this test, the message size is reduced (between 10000 bytes and 20000 bytes). As a result, the network latency also decreases. The results of our prediction is shown in Figure 4.17. In this experiment, our simulator successfully find the better placement in 100% of tests.

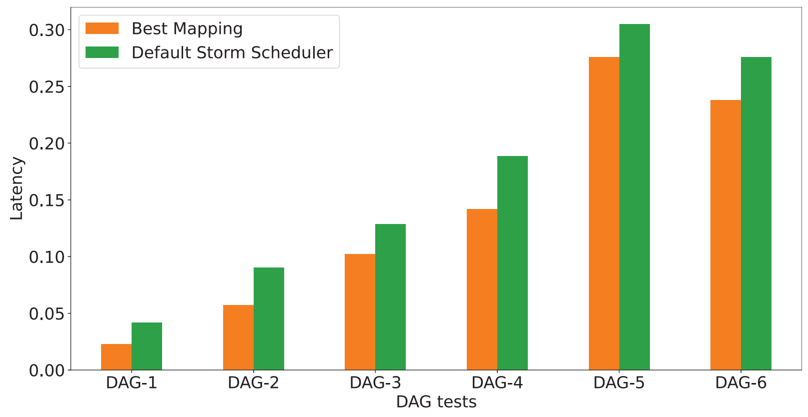
4.6.6 Experiment six: Reducing the CPU usage

We can reduce the processing time by reducing the CPU usage. So we have lower processing time at the operators. In this test, the fake processing range is between 5000 loops and 10000 loops. The prediction results of our simulator are presented in Figure 4.18. In three tests (DAG-3 and DAG-5 of section a and DAG-5 of section c in Figure 4.18), our simulator does not provide the prediction result for the default scheduler because the bandwidth constraint is not met. In 100% of tests mapping of EdgeDAG beats the default Storm scheduler mapping.

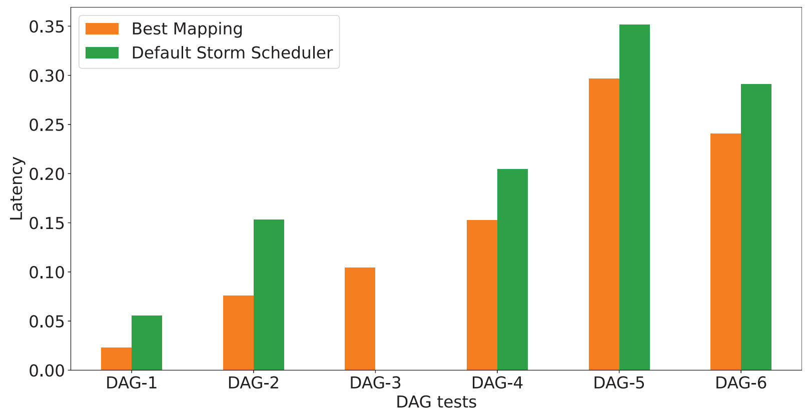
4.7 Discussion

In this section, we review the results in more details. By comparing the real deployment results in experiment one (decreasing the bandwidth) and experiment three (increasing the latency) with experiment four (reducing the message size), we can observe and how the network latency affects the results (in Figures 4.8, 4.10 and 4.11). We have a very high latency when we restrict the capabilities of the edge network (bandwidth and latency), while the latencies are much lower when the network is more available (lower message size and consequently lower throughput).

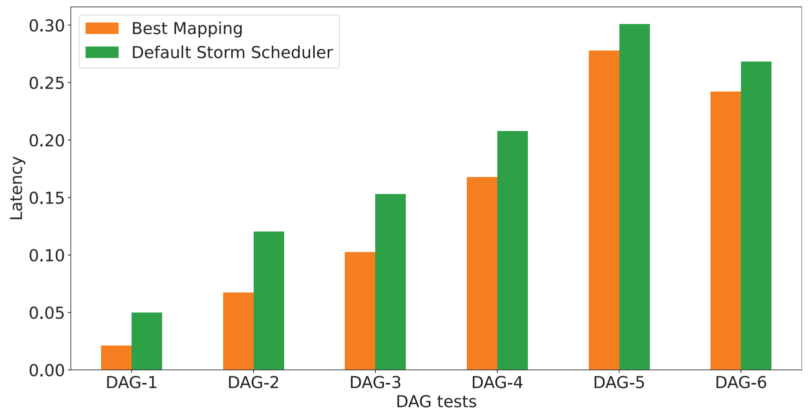
As we can observe in Table 4.4, in most cases, our scheduler provides better results than the cloud and baseline (Storm scheduler) deployments. Similarly, our simulator finds a better placement in most cases. In complex applications, we have a larger search space, so finding a good solution may be more difficult. The GWO meta-heuristic algorithm can find a better result if we run it



a) Results for simulator prediction (1)

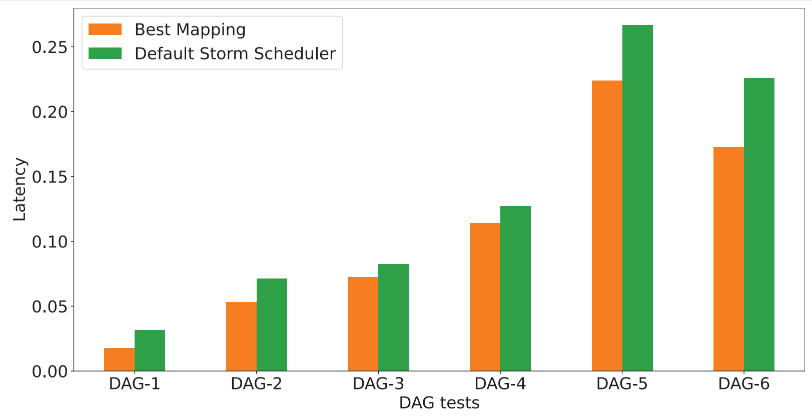


b) Results for simulator prediction (2)

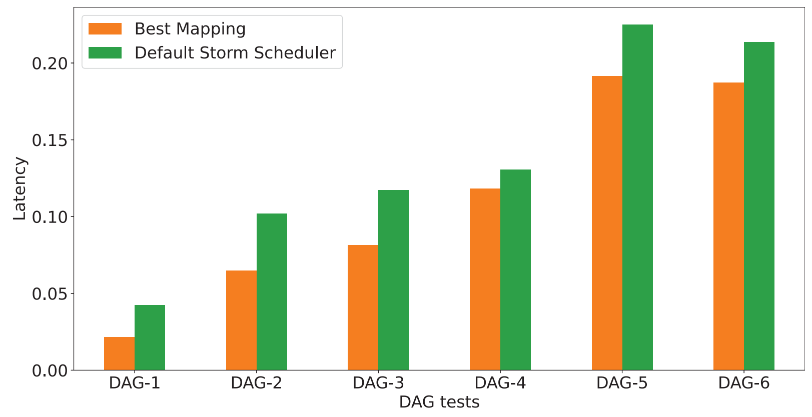


c) Results for simulator prediction (3)

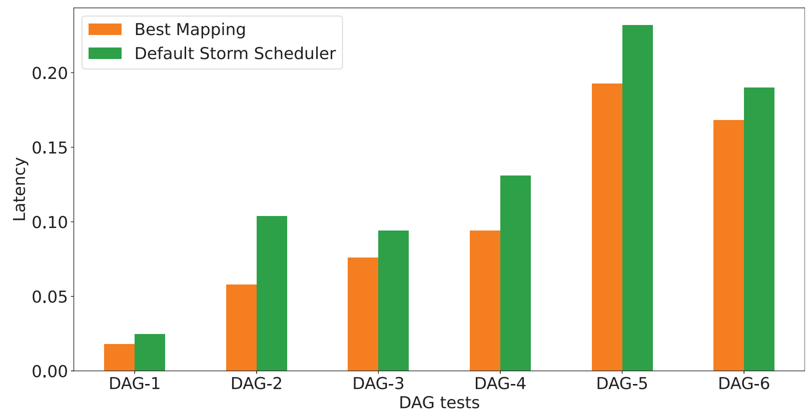
Figure 4.16 Results for simulator prediction for higher latency experiment (experiment four)



a) Results for simulator prediction (1)

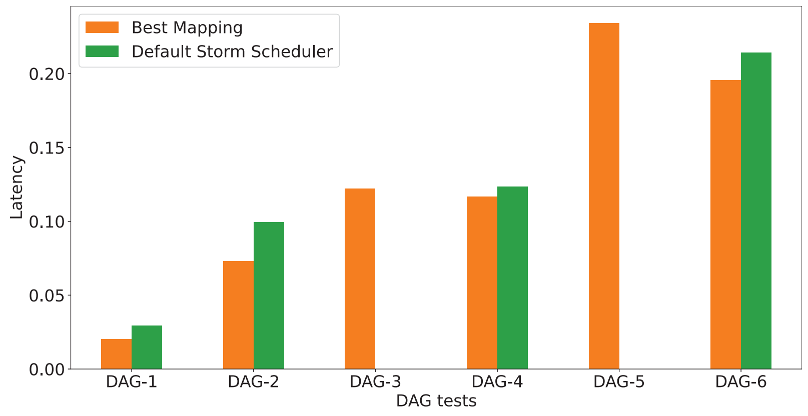


b) Results for simulator prediction (2)

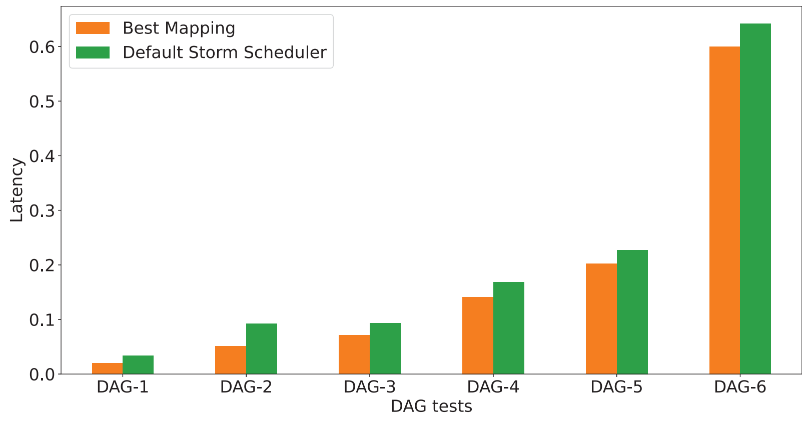


c) Results for simulator prediction (3)

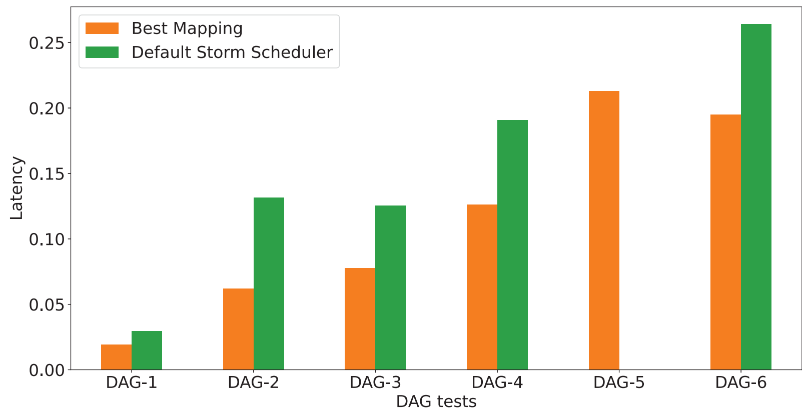
Figure 4.17 Results for simulator prediction for lower message size experiment (experiment five)



a) Results for simulator prediction (1)



b) Results for simulator prediction (2)



c) Results for simulator prediction (3)

Figure 4.18 Results for simulator prediction for lower CPU usage experiment (experiment six)

with a higher number of iterations and population. Also, we believe that other factors, such as buffering size, operating system context switching and packet delivery to the application layer may have affected the problem in cases where our scheduler does not deliver better results. Working with such factors is tricky and it is hard to include them in the modelling of the system. Moreover, there might be latency overhead in the network interfaces that causes the error in the accuracy of the latency. On the other hand, the time is not synchronized completely. Since we track the time in the source and sink separately, it can affect the results even though we use an NTP server.

Table 4.4 Experiment results

	EdgeDAG acts better than			Simulator acts better than
Experiment	Cloud	Storm	Both	Storm
1	77%	88%	72%	100%
2	94%	88%	83%	100%
3	83%	94%	77%	88%
4	77%	83%	66%	100%
5	94%	94%	88%	100%
6	94%	94%	88%	100%

Due to the high number of messages entering the operators, we sometimes experienced a connection error in Apache Storm in some experiments. It can affect the latency for processing the data. When these errors occur often, the Nimbus may treat the supervisor as a dead node. Then, the Nimbus uses another supervisor and continues running the application. Nevertheless, in the cloud experiments, we may see dead nodes because of the cloud's high processing speed and the sources' bandwidth limitations might increase the number of error connections. In other words, due to a high pressure on some Apache Storm supervisors at the edge, we might lose the connection, and the Nimbus might bring up new supervisors as an alternative. Buffering size at Apache Storm can be another reason for losing the connections to the edges. We might also face buffer overflow issues on the edge nodes because of the high rate of sending the data from the cloud to the edge.

In real deployment experiments, we can see that sometimes we have very different results among three runs of each experiment in the default Storm scheduler and the cloud deployment. The reason is the changes in the random deployments of the operators over the edge nodes. Moreover, the bandwidth of the edges nodes varies between each run. As a result, we might see different results in each run, hence the reason why we independently report results for several runs. That being said, in most cases, the EdgeDAG results are consistent since it finds a good placement. Therefore, we do not have major changes between EdgeDAG deployments. In 2% of the tests (part three of Figure 4.9, part two of Figure 4.10 and part two of Figure 4.12), we face higher latencies for the EdgeDAG deployments in comparison with the other two runs. We believe that in these cases, the random values of bandwidth and latency of the edge nodes are not good since in most cases, a better placement is offered in comparison with the cloud deployment and the default Storm scheduler. As it is mentioned, other factor such as buffering size can also affect the high latency results.

4.7.1 General DAG-based review

In this section, we review the results for each DAG separately. According to the results provided in Table 4.5, for DAG-1, EdgeDAG finds a better placement in 61% of the cases. For DAG-1, we have a very small search space and as the prediction results show, EdgeDAG predict a better placement. But in a real deployment, the results are not very good as we expected. We are not fully sure, but we pose the hypothesis that because of the small number of edges in this DAG (5 edges and 5 vertices), the network latency does not significantly influence the results. Consequently, the effect of EdgeDAG would not be significant, and the variations due to the other factors (e.g., CPU, buffering, operating system context switching) affect the results more than EdgeDAG. It is worth mentioning that since the CPU power of the devices is the same, EdgeDAG is more effective in finding better results when the end-to-end latency is increased (higher number of edges), while a lower number of edges in this DAG causes EdgeDAG to not find a significantly better placement in real deployments. On the other hand, for DAG-2, we obtain a better placement for 94% of the time. In this DAG, we have more edges. As a result, EdgeDAG has more chances to find a better placement in comparison to DAG-1. For DAG-2, we find a better placement for all the tests (100% of tests) in the simulation results which is almost same as the results of the real deployment (94% of tests).

Table 4.5 Results for each DAG

DAG number	EdgeDAG acts better than both approaches
1	61%
2	94%
3	100%
4	33%
5	100%
6	88%

In 100% of the tests, EdgeDAG provides a better placement for DAG-3 while the prediction results show that in one case, a better placement is not predicted. DAG-5 shows the same behaviour. In DAG-5, although the simulator does not predict a better placement for one test case out of 18 tests cases, we obtain a better placement in 100% of the test cases. We believe that we need to consider an error rate for the provided results since sometimes the results are very close together (0.1 ms differences). Moreover, by increasing the GWO search parameters (e.g., population number, iteration number), the simulator can provide better prediction results. In general, the simulation results and deployment results are close for these two DAGs.

In DAG-4, EdgeDAG is not as effective, as it beats the cloud only in 33% of cases. However, it manages to provide better results rather than the default Storm scheduler (as it is shown in the simulation result) in 94% of the test cases. We have the highest rate of edges to vertices in this DAG. The number of edges is three times higher than the number of vertices, and we do not have this rate in other DAGs (DAG-4 is more complex). In this DAG, the network latency is significantly increased because of the high number of edges, while in the cloud deployment, there is not a lot of network latency since most of the operators are run over the cloud. In this DAG, the number of sources and sinks is similar to that of DAG-3. The latency in the cloud deployment is increased whenever where we have more sources and sinks. In DAG-4, the number of edges is higher in comparison to DAG-3, so the network latency is consequently higher. Therefore, we might see better result in the cloud deployment strategy. As it is mentioned in Section 4.4, for DAG-4, we search the problem space more in comparison to DAG-3, but the results are not good. We believe that in some cases, even the best placement of search space might not be able to beat the cloud deployment because of the high number of edges.

For DAG-6, in 88% of cases, a better result is provided. Since the simulator provides a better placement in all the tests, it is possible that other factors (e.g., buffering, operating system context switching) could have affected the results of the real deployment.

4.7.2 Real deployment results and simulation results comparison

As it is shown in the real deployment and simulator experiments, it is not possible to directly compare the results of the real deployment with the results of the simulation. Building a system that can predict the real deployment latency was another goal of this thesis, but we found out it was much more complex than expected. After comparing the results, we observe that factors such as buffering size, context switching of operating systems, packet preparation and delivery to the application layer can affect the results. On the other hand, estimating these factors is very tricky. As a result, we do not focus on them. Therefore, we do not provide a prediction model in this thesis. We believe that accurately predicting the latency in a manner that is comparable to the real deployment latency is an interesting area for future work.

CONCLUSION AND RECOMMENDATIONS

The main goal of this thesis is to provide a tool that can offer a good initial placement for operators of distributed stream processing applications at the edge of the network. Our goal is to find a good placement in terms of latency while considering the constraints of the environment, namely the bandwidth and CPU usage. We can use the proposed tool to find a good initial placement for the operator placement problem. Our tool offers a good mapping between the operators and the edge nodes by analyzing the application characteristics. We implement our simulator as a Java application, and we use it to find a good placement of operators over the edge nodes and validate the proposed approaches.

Our experiments prove that our tool is more effective than two baseline approaches: deploying the operators in the cloud, and the Default Apache Storm scheduler. Also, we consider highly complex DAG applications with many operators and edges, and we deploy these applications on the edge network, which allows us to validate our results in a real test-bed environment. To the best of our knowledge, this is the first study that validates a methodology for complex DAG applications since most of the previous approaches are tested on very simple applications. The proposed tools allow us to find a better placement for most of the cases.

Future work could extend this work in different dimensions. In the following sections, we review some of them.

Considering more metrics

In this thesis, we considered a limited set of metrics. In a real deployment, other metrics can affect the results. Some metrics that we would like to consider in future work are characteristics like the buffer sizes and the memory usage.

Dynamic balancing

The initial placement of operators is only applicable for the first deployment. Due to the dynamic changes that can happen in the edge environment, considering the load balancing of operators is an important step. Our tool is applicable for the initial placement of operators, but it would be possible to extend our approach to use the proposed method for the dynamic placement of operators.

Edge-cloud infrastructures

Some applications can require a significant amount of resources (e.g., power and memory usage). In such cases, sometimes sending the data to the cloud and retrieving the result can be an option. Proposing an approach that works for the combination of edge and cloud resources would be helpful for the operator placement problem. The proposed tools in this thesis also can be extended and can work with the combined infrastructures.

Operator replication

Operator replication is a feature that can be helpful to reduce the latency of the operator placement problem. It is possible to study the possibility of replicating operators based on the volume of requests sent to them. It can help balance the loads for the specific operators.

Multiple operators on one edge node

We can run multiple operators on one edge node since the current edge nodes have multiple cores. Thus, we can reduce the network latency between the operators that generate the largest message sizes. Currently, the simulator supports this feature, but it is not currently used – and it is possible to extend this work to deploy multiple operators on one edge node.

BIBLIOGRAPHY

Ai, Y., Peng, M. & Zhang, K. (2018). Edge computing technologies for Internet of Things: a primer. *Digital Communications and Networks*, 4(2), 77–86.

Amarasinghe, G., De Assuncao, M. D., Harwood, A. & Karunasekera, S. (2018). A data stream processing optimisation framework for edge computing applications. *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 91–98.

Benoit, A., Dobrila, A., Nicod, J.-M. & Philippe, L. (2013). Scheduling linear chain streaming applications on heterogeneous systems with failures. *Future Generation Computer Systems*, 29(5), 1140–1151.

Cai, X., Kuang, H., Hu, H., Song, W. & Lü, J. (2018). Response time aware operator placement for complex event processing in edge computing. *International Conference on Service-Oriented Computing*, pp. 264–278.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. & Buyya, R. (2011). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1), 23–50.

Canali, C. & Lancellotti, R. (2019). GASP: genetic algorithms for service placement in fog computing systems. *Algorithms*, 12(10), 201.

Cao, Y., Chen, H., Jiang, J. & Hu, F. (2018). TaSRD: Task Scheduling Relying on Resource and Dependency in Mobile Edge Computing. *2018 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pp. 287–295.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. & Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4).

Cardellini, V., Grassi, V., Presti, F. L. & Nardelli, M. (2015). On QoS-aware scheduling of data stream applications over fog computing infrastructures. *2015 IEEE Symposium on Computers and Communication (ISCC)*, pp. 271–276.

Cardellini, V., Grassi, V., Lo Presti, F. & Nardelli, M. (2016). Optimal operator placement for distributed stream processing applications. *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 69–80.

Cardellini, V., Grassi, V., Lo Presti, F. & Nardelli, M. (2017). Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Performance Evaluation Review*, 44(4), 11–22.

Cardellini, V., Lo Presti, F., Nardelli, M. & Russo Russo, G. (2018). Optimal operator deployment and replication for elastic distributed data stream processing. *Concurrency and Computation: Practice and Experience*, 30(9), e4334.

Cheng, B., Papageorgiou, A. & Bauer, M. (2016). Geelytics: Enabling on-demand edge analytics over scoped data sources. *2016 IEEE International Congress on Big Data (BigData Congress)*, pp. 101–108.

da Silva Veith, A., de Assuncao, M. D. & Lefevre, L. (2018). Latency-aware placement of data stream analytics on edge computing. *International Conference on Service-Oriented Computing*, pp. 215–229.

da Silva Veith, A., de Assuncao, M. D. & Lefevre, L. (2021). Latency-aware strategies for deploying data stream processing applications on large cloud-edge infrastructure. *IEEE transactions on cloud computing*.

Dias, M., Assunção, D., Veith, S. & Buyya, R. (2018). Distributed data stream processing and edge computing : A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103(November 2017), 1–17. doi: 10.1016/j.jnca.2017.12.001.

Eidenbenz, R. & Locher, T. (2016). Task allocation for distributed stream processing. *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9.

Elgamal, T., Sandur, A., Nguyen, P., Nahrstedt, K. & Agha, G. (2018). DROPLET : Distributed Operator Placement for IoT Applications Spanning Edge and Cloud Resources. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 1–8. doi: 10.1109/CLOUD.2018.00008.

Eskandari, L., Mair, J., Huang, Z. & Eyers, D. (2021). I-Scheduler: Iterative scheduling for distributed stream processing systems. *Future Generation Computer Systems*, 117, 219–233.

Gedeon, J., Stein, M., Wang, L. & Muehlhaeuser, M. (2018). On scalable in-network operator placement for edge computing. *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–9.

Ghosh, R. & Simmhan, Y. (2018). Distributed scheduling of event analytics across edge and cloud. *ACM Transactions on Cyber-Physical Systems*, 2(4), 1–28.

Ghosh, R., Komma, S. P. R. & Simmhan, Y. (2018). Adaptive energy-aware scheduling of dynamic event analytics across edge and cloud resources. *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 72–82.

Hiessl, T., Karagiannis, V., Hochreiner, C., Schulte, S. & Nardelli, M. (2019). Optimal placement of stream processing operators in the fog. *2019 IEEE 3rd International Conference on Fog and Edge Computing, ICFEC 2019 - Proceedings*, 1–10. doi: 10.1109/CFEC.2019.8733147.

Janßen, G., Verbitskiy, I., Renner, T. & Thamsen, L. (2018). Scheduling stream processing tasks on geo-distributed heterogeneous resources. *2018 IEEE International Conference on Big Data (Big Data)*, pp. 5159–5164.

Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I. & Ahmed, A. (2019). Edge computing: A survey. *Future Generation Computer Systems*, 97, 219–235.

Khare, S., Sun, H., Gascon-Samson, J., Zhang, K., Gokhale, A., Barve, Y., Bhattacharjee, A. & Koutsoukos, X. (2019). Linearize, predict and place: minimizing the makespan for edge-based stream processing of directed acyclic graphs. *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pp. 1–14.

Kim, Y., Son, S. & Moon, Y.-S. (2019). SPMgr: Dynamic workflow manager for sampling and filtering data streams over Apache Storm. *International Journal of Distributed Sensor Networks*, 15(7), 1550147719862206.

Lin, L., Li, P., Xiong, J. & Lin, M. (2018). Distributed and Application-aware Task Scheduling in Edge-clouds. *2018 14th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pp. 165–170.

Liu, J., Mao, Y., Zhang, J. & Letaief, K. B. (2016). Delay-optimal computation task scheduling for mobile-edge computing systems. *2016 IEEE international symposium on information theory (ISIT)*, pp. 1451–1455.

Liu, P., Da Silva, D. & Hu, L. (2021). {DART}: A Scalable and Adaptive Edge Stream Processing Engine. *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 239–252.

Madej, A., Wang, N., Athanasopoulos, N., Ranjan, R. & Varghese, B. (2020). Priority-based Fair Scheduling in Edge Computing. 1–10.

Mahmud, R., Ramamohanarao, K. & Buyya, R. (2019). Edge affinity-based management of applications in fog computing environments. *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pp. 61–70.

Mirjalili, S., Mirjalili, S. M. & Lewis, A. (2014). Grey wolf optimizer. *Advances in engineering software*, 69, 46–61.

Nardelli, M., Cardellini, V., Grassi, V. & Presti, F. L. (2019). Efficient Operator Placement for Distributed Data Stream Processing Applications. 30(8), 1753–1767.

Oussous, A., Benjelloun, F.-Z., Lahcen, A. A. & Belfkih, S. (2018). Big Data technologies: A survey. *Journal of King Saud University-Computer and Information Sciences*, 30(4), 431–448.

Peng, Q., Xia, Y., Wang, Y., Wu, C., Luo, X. & Lee, J. (2019). Joint operator scaling and placement for distributed stream processing applications in edge computing. *International Conference on Service-Oriented Computing*, pp. 461–476.

Pfandzelter, T. & Bermbach, D. (2019). Iot data processing in the fog: Functions, streams, or batch processing? *2019 IEEE International conference on fog computing (ICFC)*, pp. 201–206.

Rajaraman, V. (2014). Cloud computing. *Resonance*, 19(3), 242–258.

Renart, E. G., Veith, A. D. S., Balouek-Thomert, D., De Assunção, M. D., Lefevre, L. & Parashar, M. (2019). Distributed operator placement for IoT data analytics across edge and cloud resources. *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 459–468.

Satyanarayanan, M. (2017). The emergence of edge computing. *Computer*, 50(1), 30–39.

Scoca, V., Aral, A., Brandic, I., De Nicola, R. & Uriarte, R. B. (2018). Scheduling Latency-Sensitive Applications in Edge Computing. *Closer*, pp. 158–168.

Shi, W., Cao, J., Zhang, Q., Li, Y. & Xu, L. (2016). Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5), 637–646.

Shi, W., Pallis, G. & Xu, Z. (2019). Edge Computing [Scanning the Issue]. *Proceedings of the IEEE*, 107(8), 1474-1481. doi: 10.1109/JPROC.2019.2928287.

Shoro, A. G. & Soomro, T. R. (2015). Big data analysis: Apache spark perspective. *Global Journal of Computer Science and Technology*.

Shukla, A., Chaturvedi, S. & Simmhan, Y. (2017). RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms. *Deloitte University Press*, 54. doi: 10.5120/19787-1571.

Sonmez, C., Ozgovde, A. & Ersoy, C. (2018). Edgecloudsim: An environment for performance evaluation of edge computing systems. *Transactions on Emerging Telecommunications Technologies*, 29(11), e3493.

Stephens, R. (1997). A survey of stream processing. *Acta Informatica*, 34(7), 491–541.

Taneja, M. & Davy, A. (2017). Resource aware placement of IoT application modules in Fog-Cloud Computing Paradigm. *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 1222–1228.

Varshney, P. & Simmhan, Y. (2020). Characterizing application scheduling on edge, fog, and cloud computing resources. *Software: Practice and Experience*, 50(5), 558–595.

Wei, X., Wei, X., Li, H., Zhuang, Y. & Yue, H. (2018). Topology-aware task allocation for distributed stream processing with latency guarantee. *Proceedings of the 2nd International Conference on Advances in Image Processing*, pp. 245–251.

Yin, F., Li, X., Li, X. & Li, Y. (2019). Task Scheduling for Streaming Applications in a Cloud-Edge System. *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*, pp. 105–114.

Yu, W. E. I., Liang, F. A. N., He, X., Hatcher, W. G., Lu, C., Lin, J. I. E. & Yang, X. (2018). *A Survey on the Edge Computing for the Internet of Things*. IEEE Access. IEEE.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S. & Stoica, I. (2012). Resilient Distributed Datasets: A {Fault-Tolerant} Abstraction for {In-Memory} Cluster Computing. *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 15–28.