PROGRAMMING AND MANAGING DATA-DRIVEN APPLICATIONS BETWEEN THE EDGE AND THE CLOUD

BY

EDUARD GIBERT RENART

A dissertation submitted to the

School of Graduate Studies—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Manish Parashar

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2020

ABSTRACT OF THE DISSERTATION

Programming and Managing Data-Driven Applications between the edge and the cloud

by Eduard Gibert Renart

Dissertation Director: Manish Parashar

Due to the proliferation of the Internet of Things (IoT), the number of devices connected to the Internet is growing. These devices are generating large volumes of data at the edge of the infrastructure. According to International Data Corporation (IDC) predictions by 2025 the worldwide data will reach 180 zettabytes (ZB), and more than half of that data will come from IoT sensors. Although the generated data provides great potential for science and society, identifying and processing relevant data points hidden in streams of unimportant data, and doing this in near real-time, remains a significant challenge. The prevalent model of moving data from the edge to the cloud of the network is becoming unsustainable, resulting in an impact on latency, network congestion, storage cost and privacy. These observations can be leveraged to design hybrid architectures that can leverage both the edge and the cloud resources to process the data in a timely manner. Although the cloud is better suited to perform heavier (resource intensive) analysis, such as processing historical events and very large datasets, edge devices can support real-time analytics that consider the temporal and spatial characteristics of IoT data. While edge processing can benefit IoT applications, edge resources are typically constrained in their capabilities. In addition integrating edge computing can also add complexity to applications, especially when they need to

include policies that govern what kind of data is processed and analyzed at the edge and what is sent to cloud.

To address these challenges, this dissertation presents an IoT Edge Framework, called R-Pulsar, that extends cloud capabilities to local devices and provides a programming model for deciding what, when, where and how data get collected and processed. This thesis makes the following contributions: (1) A content- and location-based programming abstraction for specifying **what** data gets collected and **where** the data gets analyzed. (2) A rule-based programming abstraction for specifying **when** to trigger data-processing tasks based on data observations. (3) A programming abstraction for specifying **how** to split a given dataflow and place operators across edge and cloud resources. (4) An operator placement strategy that aims to minimize an aggregate cost which covers the end-to-end latency (time for an event to traverse the entire dataflow), the data transfer rate (amount of data transferred between the edge and the cloud) and the messaging cost (number of messages transferred between edge and the cloud). (5) Performance optimizations on the data-processing pipeline in order to achieve real-time performance on constrained devices. The applicability of this work to real-world IoT applications is validated through a series of experiments in which shows that R-Pulsar can reduce the bandwidth consumption between the edge and the cloud by up to 82% and obtain results 40% faster than the traditional approach of moving all the data to the cloud.

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Manish Parashar, for his constant support and guidance during my PhD journey. I would like to thank Dr. Daniel Balouek-Thomert for all his mentorship, feedback and support during my PhD journey. I would like to thank the members of my committee, Dr. Ulrich Kremer, Dr. Srinivas Narayana Ganapathy and Dr. Otto Anshus, for reading, reviewing and offering great advice on my dissertation. I would like to thank my colleagues at Rutgers Discovery Informatics Institute (RDI^2) for the good advice and for supporting each other through our respective academic journeys. Finally, I would like to thank my mother and my sister for always being there when I needed them and for supporting me unconditionally. This wouldn't be possible without your support. Last but not least I would like to thank my girlfriend Sasha for always believing in me, helping me get through this journey and always being my number one fan.

Dedication

To my mother and late father.

Thanks for always supporting me and believing in me, this would not be possible without your unconditional love and guidance.

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

List of Tables x

List of Figures xi

1. Introduction 1

1.1. Motivation 3

1.2. Problem Description 4

1.3. Contributions 5

1.4. Outline 6

2. Motivating Applications and Requirements 8

2.1. Emergence, Benefits and Limitations of Using Edge Computing 8

2.1.1. Advantages 8

2.1.2. Disadvantages 9

2.2. Motivating Applications 9

2.2.1. Smart City 10

2.2.2. Disaster Recovery 11

Stage 1: Data generation - Pre-processing Stage 13

Stage 2: Change detection - Post-processing stage 14

2.2.3. Scientific Observatory 15

2.2.4. Video Analytics 15

2.2.5. Observe Orient Decide Act Loop 16

3. Background and Related Work . . . 17
3.1. Introduction . . . 17
3.2. Edge-based Middleware Architecture . . . 18
3.2.1. Resource Management Layer . . . 20
Design Goals . . . 20
Resource Discovery . . . 20
Resource Monitoring . . . 22
Resource Mobility . . . 22
3.2.2. Data Processing Layer . . . 23
Design Goals . . . 24
Data Ingestion . . . 24
Data Analysis . . . 25
Data Storage . . . 26
Data Query . . . 27
3.2.3. Service Layer . . . 27
Design Goals . . . 28
Rule Engine . . . 29
Programming Models . . . 29
Workflow Orchestrator . . . 30
3.2.4. Security Layer . . . 31
Data privacy . . . 31
End-to-End Security . . . 32

4. Associative Rendezvous (AR) . . . 44
4.1. Introduction . . . 44
4.2. R-Pulsar Associative Rendezvous . . . 45
4.2.1. AR Message . . . 45
4.2.2. Reactive Behaviors . . . 46
4.2.3. Illustrative Example . . . 47

5. Enabling Data-driven IoT Applications 50
5.1. Infrastructure Layer 51
5.2. Federation Layer 51
Location-aware Overlay Network Component 51
Content-based Routing Component 53
5.3. Streaming Layer 54
5.3.1. Serverless Messaging Component 54
5.3.2. Memory-mapped Streaming Analytics Component 55
Data Collection 56
Data Processing 56
Data Storage and Query 57
5.3.3. Rule-based Programming Abstraction Component 57
Content-Driven Rules 58
Data Quality Rules 60
5.4. Application Layer 62

6. R-Pulsar, an edge-based middleware 63
6.1. Design 63
6.2. Implementation 64
6.3. API Examples 66
6.4. Location-aware Overlay Network Component Evaluation 70
6.4.1. Overhead and Scalability Over Cloud And Edge Systems 70
6.4.2. Overhead and Scalability of the Data Replication 71
6.5. Content-based Routing Component Evaluation 72
6.5.1. Profile Matching Overhead Over Cloud Systems 72
6.5.2. Routing Overhead and Scalability Over Edge Systems 73
6.5.3. Information Propagating through the System 75
6.6. Memory-mapped Streaming Analytics Pipeline Evaluation 76
6.6.1. Performance of the Data Collection Layer Over Edge Systems . . 77

6.6.2. Scalability of Store/Query Operations Over Cloud Systems . . . 78
6.6.3. Performance of the Query and Store Layer Over Edge Systems . 80
6.7. Rule-based Programming Abstraction Evaluation 81
6.7.1. Scalability and Overhead Over Edge and Cloud Systems 82
6.8. End To End Evaluation 83
6.8.1. Disaster Response Use Case 83

7. Application to the Distributed Operator Placement Problem 89
7.1. Introduction 89
7.2. Problem Description 89
7.3. R-Pulsar Framework Extension 94
7.3.1. R-Pulsar Nodes 95
7.3.2. Placement Strategy 96
7.3.3. R-Pulsar API 100
7.4. Evaluation 100
7.4.1. Setup 101
7.4.2. End-to-end Tuple Latency Evaluation 102
7.4.3. Data Transfer Rate Evaluation 104
7.4.4. Messaging Cost Evaluation 105
7.4.5. Model Evaluation 107

8. Conclusion 110
8.1. Summary 110
8.2. Contributions 110
8.3. Perspectives 111
References 113

List of Tables

2.1. Edge Computing use cases, current limitations, and imperatives 10

3.1. Design goals of the resource management layer components of the cited papers in this survey. 19

3.2. Design goals of the data processing layer components of the cited papers in this survey. 23

3.3. Design goals of the service layer components of the cited papers in this survey. 28

3.4. Design goals of the security layer components of the cited papers in this survey. 31

3.5. Four Layer and Components for all the commercial and academic edge middleware available. 40

3.6. Resource management layer and design goals for all the commercial and academic edge middleware available. 41

3.7. Data processing layer and design goals for all the commercial and academic edge middleware available. 42

3.8. Service layer and design goals for all the commercial and academic edge middleware available. 43

4.1. R-Pulsar Reactive behaviors 46

5.1. Available R-Pulsar operators. 55

5.2. Measurements of Disk I/O vs RAM memory performance on a Raspberry Pi. 57

5.3. Available R-Pulsar rule operators. 58

7.1. Main notation adopted for the problem description. 90

7.2. Azure IoT Hub and Amazon IoT Core messaging pricing. 105

List of Figures

1.1. Thesis Organization. 6
2.1. Disaster recovery decision stages and its associated reactions based on the LiDAR images. 12
2.2. RIoTBench IoT high-level logical interactions between different sensors, applications and users. 16
3.1. Edge-based middleware reference architecture consisting of four layers, each of them with their respective components. 18
4.1. An illustration of the pub/sub model. 44
4.2. An illustration of the pub/sub model. 44
4.3. An example illustrating the operation of associative rendezvous. 47
4.4. One-to-many interactions using associative rendezvous. 48
5.1. Schematic overview of the R-Pulsar Architecture 50
5.2. R-Pulsar quadtree geographical organization to multi layer P2P overlay network. 51
5.3. R-Pulsar space filling curve routing using simple (a) and complex (b) profiles. 53
5.4. A time duration based sliding window with length 10 secs and sliding interval of 5 seconds. 60
5.5. A time duration based tumbling window with length 5 secs.. 60
5.6. Storm topology linewise representation 60
6.1. Sequence diagram of R-Pulsar, when a sensor wants to register itself in the R-Pulsar network. 63
6.2. R-Pulsar main classes interaction. 65
6.3. Trigger topology reaction rule definition. 68

6.4. Store reaction rule definition. 69
6.5. Rule engine window bolt definition. 69
6.6. Tuple QoS rule definition 70
6.7. Location-based query overhead for different number of matches per query. 71
6.8. Time neccessary to update information of all RP nodes in the system after multiple topologies are registered. 71
6.9. Profile matching overhead. Results are grouped based on the matching ratio expressed as a percentage of the total number of stored profiles. . . 73
6.10. Evaluation of R-Pulsar space filling curve routing overhead and scalability as the number of messages and the complexity of profiles increase over the Android system. 74
6.11. Evaluation of R-Pulsar space filling curve routing overhead and scalability as the number of messages and the complexity of profiles increase over the Raspberry Pi system. 74
6.12. Time neccessary to update information of all RP nodes in the system after multiple new RP nodes join. 75
6.13. Single broker and producer throughput performance as message sizes grow. Comparison of R-Pulsar, Kafka, and Mosquitto systems deployed on a single Raspberry Pi. 77
6.14. Single broker and producer throughput performance as message sizes grow. Comparison of R-Pulsar and Mosquitto on the Android system. . 78
6.15. Evaluation of R-Pulsar store query operations as the number of nodes increases on Chameleon cloud. 79
6.16. Evaluation of R-Pulsar exact query operations as the number of nodes increases on Chameleon Cloud. 80
6.17. Storage performance of R-Pulsar, SQLite and Nitrite DB as the number of elements to be stored increases over 10 Raspberry Pi. 81
6.18. Exact query performance of R-Pulsar, SQLite and Nitrite DB as the number of exact queries increases over 10 Raspberry Pi. 82

6.19. Wildcard query performance of R-Pulsar, SQLite and Nitrite DB as the number of wildcard queries increases over 10 Raspberry Pi. 83
6.20. Rule engine overhead for different number of rules. 84
6.21. Data quality rule-based system 20 second deadline. 85
6.24. Disaster response workflow 50 images edge speed-up. 85
6.22. Disaster response workflow edge vs core no QoS. 86
6.25. Disaster response workflow 100 images edge speed-up. 86
6.23. Disaster response workflow 10 images edge speed-up. 87
6.26. Disaster response workflow 100 images with breakdown. 87
7.1. Example of four operators and their respective queues placed on two resources. 91
7.2. Phases to determine the final placement using split points, where red means placed on edge, blue represents placed on cloud, and green delimits forks and joins. 97
7.3. End-to-end tuple latency optimization with 3 self injected failures affecting edge and cloud nodes, while comparing it with Cloud, Random and LB approaches. 103
7.4. End-to-end tuple latency optimization cumulative distribution function (CDF) comparison with Cloud, Random and LB approaches. 103
7.5. End-to-end data transfer rate optimization cumulative distribution function (CDF) comparison with cloud, Random and LB approaches. 104
7.6. Multi optimization evaluation, end-to-end tuple latency and data transfer rate comparison with cloud, Random, and LB approaches. 105
7.7. Messaging cost savings evaluation based on the Microsoft Azure IoT Hub pricing model, for four different setups. 106
7.8. Messaging cost savings evaluation based on the Amazon IoT pricing model, for four different setups. 107
7.9. Evaluation of the scalability of the operator placement problem algorithm as the number of operators to place increase over edge and cloud resources. 107

7.10. Evaluation of the cost of redeploying a subset of operators over edge and cloud resources. 108
7.11. Evaluation of the actual deployment vs the modeled time as the number of operators increases. 108

Chapter 1

Introduction

The Internet of Things paradigm (IoT) fosters the connection of large numbers of sensors to the network. According to Cisco systems [1], 500 billion IoT devices are expected to be connected to the Internet by 2030, and mearly 50% of the data produced worldwide will be generated by IoT sensors [2] IoT devices produce important and timely data that can lead to new and transformative applications that are important to science and society, such as:

- Precision medicine applications that benefit from runtime actuation based on continuous monitoring by scientific instruments.
- Urban mobility applications that rely on processing data from sensors to identify and alleviate traffic congestion.
- Healthcare applications that infer lifestyle patterns based on behavioral information obtained from wearables.

Making such applications a reality requires collecting data from sensors and instruments, processing this data individually or collectively in a timely manner, and making decisions based on the results.

Stream processing frameworks (SPFs) have proven to be very effective at processing large amounts of data in a timely manner, especially when combined with the elasticity and scalability of the cloud. Nonetheless, existing solutions were developed keeping in mind Big Data streams generated at the core of the infrastructure, such as those associated with web analytics. As a result, applying these solutions to IoT data stream requires transferring data from the edges to a data center located at the core of the

infrastructure for processing. This model of moving data is quickly becoming unsustainable [3], due to the resulting impact on latency, network congestion, storage cost, and privacy, limiting the potential impact of IoT.

However, in recent years, non-trivial computational capabilities have proliferated across the computing service landscape [4]. In particular, edge services are emerging closer to the data sources and can provide potential data-processing capabilities[5, 6].

Edge computing extends the traditional cloud infrastructure with additional computing resources enabling the execution of applications close to end-user. Edge computing uses edge nodes, which may range from IoT embedded devices featuring limited storage, memory, and processing capacity to whole data centers (i.e. "local clouds") which are deployed closer to end-users and physical infrastructures. Overall, the edge computing paradigm extends the cloud paradigm to the edge of the network. In this way, users can benefit from computing, storage and communication resources at their vicinity, instead of interfacing with the centralized cloud.

IoT data feature certain characteristics, which distinguish them radically from other types of data sources and respective applications. The special characteristics and related challenges for IoT data processing applications can be listed as follows:

- **Geographically Distributed:** IoT data streams are produced in a geographically distributed fashion manner.
- **Time and Location:** The data streams contains temporal and spatial dependencies, which are directly related with the value. For that reason IoT applications need to process data in a timely fashion and from the proper location, in order to extract its maximum value.
- **Real-Time:** IoT data streams are produced in high velocities (machine speed) and requires applications to process the data in real-time.
- **Security:** The majority of the IoT data produced by sensors contains personal and sensitive data.

- **Mobility:** IoT applications can involve sensors that moves around such as: connected vehicles, autonomous cars, etc... and requires the communication to local resources (computing, storage) residing in their vicinity.

While edge computing can help achieve all the new requirements that IoT applications need, edge resources are typically constrained in their capabilities. Therefore, edge computing can be leveraged to complement the computing capabilities of the cloud-centric approach. The use of edge and cloud architecture poses several challenges:

- Deciding how to split IoT applications among the edge and cloud resources, in order to meet the requirement of the application; Where an IoT application is a sequence of operators from a source to a sink.
- Exploring heterogeneous infrastructure for deploying data flow applications has proved to be NP-hard [7]. Due to the fact that there are so many possible combinations (many edge devices and many operators to place).
- Moving operators from cloud to edge devices is challenging due to the devices' limitations with respect to memory, CPU, and often network bandwidth [8].

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

1.1 Motivation

The popularity and proliferation of the Internet of Things (IoT) paradigm is resulting in a growing number of devices connected to the Internet. These devices are generating and consuming unprecedented amounts of data at the edges of the infrastructure, and are enabling new classes of applications, however, current approaches typically rely on cloud platforms located at the core of the infrastructure to process data. As the number of devices and the amount of data they generate and consume increases, such

core-centric approaches are becoming increasingly inefficient as they need to transfer data back and forth between the edge and the core. Furthermore, not all the data produced is interesting or relevant, and only a part of it may need to be processed in the context of an application. These observations can be leveraged to design hybrid architectures that can effectively leverage both the edge and the cloud resources to process the data in an effective and timely manner[9, 10].

To address these limitations, we propose R-Pulsar, an architecture with a content- and location-based programming abstraction to perform and orchestrate data analytics between the edge and the cloud. The programming abstraction enables developers to address the **what**, **where**, and **when** data needs to be processed by specifying content and action descriptors. We also propose a programming model to provide developers with the ability to define **how** to automatically split the dataflow across the edge and the cloud by specifying a set of dataflow constraints. In addition we present an optimized data-processing pipeline for achieving timely data analytics on constrained devices.

1.2 Problem Description

With the increasing number of connected IoT devices, providing efficient and effective streaming analytics across the edge and the cloud for IoT applications is non-trivial due to the characteristics of IoT streams.

What data to consume

Due to the large number of IoT devices/sensors that are currently online, not all the data produced by the sensors is interesting or relevant, or only a part of it may need to be processed in the context of an application. Requiring all the data to be transported to the cloud for processing, will result in latencies that can prevent timely decision making or may reduce the amount of data processed. As a result there is a need for a programming model to help developers decide what data they want to consume.

Where to perform the computations

As mentioned earlier IoT streams have time and location dependencies, so in order to satisfy the requirements, computations needs to be placed between the edge and the cloud. Due to the hardware heterogeneity of edge and cloud resources, there is a need for a programming model that allows developers to reason about where they want the computations to be placed. Since edge resources are typically constrained in their capabilities and cloud resources are suited to perform heavier (resource intensive) analysis.

When to perform computations

Edge computing helps reduce the latency and the total amount of data and the load that is sent to the cloud by pre-processing, filtering and analyzing at the edge of the network. Once the data is pre-processed at the edge of the network a decision need to be made to either discard it or forward it to the cloud for post-processing. For those reasons there is a need for a rule programming abstraction to allow the ability to decide when and where to perform post-processing computations.

How to split IoT applications across the edge and the cloud

Cloud-based architectures often centralize storage and processing, generating high data movement overheads that penalize timely applications. Edge and Cloud architecture pushes computation closer to where the data is generated, reducing the cost of data movements and improving the application response time. The heterogeneity among the edge devices and cloud servers introduces an important challenge for deciding how to split and orchestrate the IoT applications across the edge and the cloud. For those reasons there is a need for a programming model to provide developers with the ability to define how to automatically split the dataflow across the edge and the cloud by specifying a set of dataflow constraints.

1.3 Contributions

The primary contributions of the research in this thesis are a programming model for deciding **what**, **when**, **where** data needs to be processed by specifying content and action descriptors and **how** computations get distributed across the edge and the cloud.

The detailed contributions are presented as follows.

- A content- and location-based programming abstraction for specifying **what** data gets collected and **where** the data gets analyzed.
- A rule-based programming abstraction for specifying **when** to trigger data-processing tasks based on data observations.
- A programming abstraction for specifying **how** to split a given dataflow and place operators across edge and cloud resources.
- An operator placement strategy that aims to minimize an aggregate cost which covers the end-to-end latency (time for an event to traverse the entire dataflow), the data transfer rate (amount of data transferred between the edge and the cloud) and the messaging cost (number of messages transferred between edge and the cloud).
- Performance optimizations on the data-processing pipeline in order to achieve high performance on constrained devices.
- An implementation of the above capabilities as part of the R-Pulsar architecture and its evaluation using embedded devices (Raspberry Pi and Android phone).

1.4 Outline

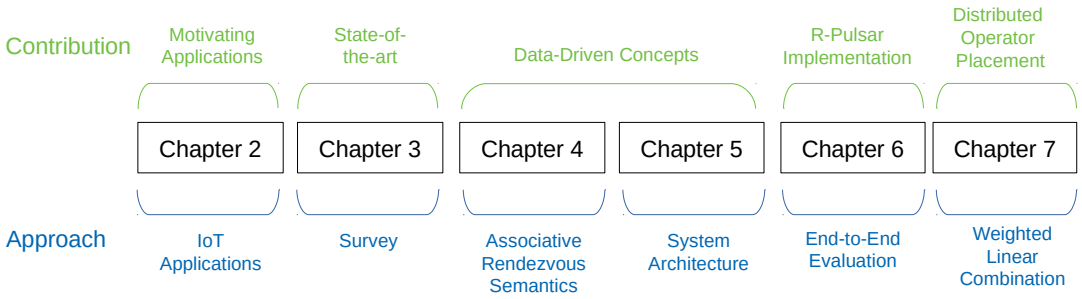

Figure 1.1: Thesis Organization.

The core chapters of the thesis are structured as shown in Figure 1.1 and are deviated

from articles and journals published during the PhD. The remaining of the thesis is organized as follows:

- Chapter 2 shows some of the IoT application that where used in order to motivate and validate the build of R-Pulsar.
- Chapter 3 presents an extensive literature review of all the commercial and academic edge-based middleware currently available.
- Chapter 4 presents the Associative Rendezvous programming abstraction that R-Pulsar builds upon.
- Chapter 5 presents the system concepts and all the layers on what R-Pulsar was build upon.
- Chapter 6 presents the implementation and evaluation details of all the layers that R-Pulsar consists.
- Chapter 7 introduces the operator placement problem, to solve the how to split IoT applications dynamically across the edge and the cloud.
- Chapter 8 concludes the dissertation by outlining future research work.

Chapter 2

Motivating Applications and Requirements

2.1 Emergence, Benefits and Limitations of Using Edge Computing

Edge computing complements the cloud in IoT by filling in the gap between the cloud and the internet of things by providing computing in the continuum. In this section we describe the advantages and disadvantages of Edge computing and later propose some IoT applications that will benefit from using edge computing.

2.1.1 Advantages

Allows to Minimize Data Explosion and Network Traffic: Due to the large number of connected devices the volume of data that they will generate will increase exponentially. Another concern with increasing data generation is the increase of network traffic to the cloud. By using edge computing data can stay local and only send the important data to the cloud, reducing the volume and network bandwidth.

Allows to Achieve Low Latency: Emerging IoT scenarios involve gathering and processing large volumes of streaming data using complex workflows in a timely manner to support decision making. Traditionally, data processing has been done at large data centers in the core of the network, however, as the data volumes and rates grow, and the application scenario becomes increasingly time sensitive, such an approach is quickly becoming infeasible, and it is becoming essential to also leverage resources closer to the edge. By applying edge computing applications are capable of supporting time-sensitive applications.

Allows to Achieve Real-Time Computations: Edge computing improves the overall performance by offloading computations near the data source and reducing the unnecessary costs of moving the data to the cloud.

Increased Security: Edge computing allows to process and filter sensitive data locally and transmit only the aggregate data over to the cloud. Therefore avoiding the need to send personal and sensitive data to the cloud.

2.1.2 Disadvantages

Limited Computational Resources: While leveraging edge resources can alleviate costs associated with cloud data transfers, edge resources tend to be constrained in their capabilities. In addition state-of-the-art data analytics pipelines are known to be computationally intensive tasks, resulting in the inability to performing timely data analytics when deployed on constrained devices.

Added complexity: Integrating edge computing adds complexity to applications, especially when they need to include policies that govern what kind of data is processed and analyzed at the edge and what is sent to cloud. In addition exploring heterogeneous infrastructures such as edge and cloud for deploying dataflow applications has proved to be NP-hard [7].

2.2 Motivating Applications

IoT applications are present in several domains: Precision medicine, Urban mobility, and Healthcare. In this section, we highlight four different use cases described in both industry and academia that benefits from the IoT paradigm. Table 2.1 summarises the scenario, limitations and requirements of those use cases.

Table 2.1: Edge Computing use cases, current limitations, and imperatives

Applications	Example Use Case	Limitations	Requirements
Smart City	Help autistic people navigate through large crowded spaces.	Hard to provide timely directions due to the need to send large volumes of data to the cloud.	Low Latency, Security, Geographically Distributed, Mobility, Scalability Reliability and Robustness
Disaster Recovery	Need to quickly determine whether building conditions are safe for evacuees to return after a natural disaster has struck.	Hard to perform timely decision due to the need to send large volumes of data to the cloud.	Low Latency, Geographically Distributed, Orchestration and Management
Distributed Observatories	Large networked system of under water instruments to collect real-time data from the ocean.	Hard to deliver near real-time data to the end user due to the need to send large volumes of data to the cloud.	Low Latency, Security, Geographically Distributed, Multi-Tenancy, Scalability
Video Analaytics	Video analytics for safety and security from public video cameras.	Hard to perform timely analytics due to the need to send large volumes of data to the cloud.	Low Latency, Security, Geographically Distributed, Scalability
Observe Orient Decide Act Loop	Refers to the decision-making cycle ofobserve, orient, decide, and act, developed by military strategists and the United States Air Force	Hard to deliver near real-time data to the end user due to the need to send large volumes of data to the cloud.	Low Latency, Security, Geographically Distributed

2.2.1 Smart City

The first use case is smart cities for people with disabilities. Large cities are difficult to navigate, especially for people with special needs such as those with visual impairment, Autism Spectrum Disorder (ASD), or simply those with navigational challenges. The primary objective of this application usecase is to explore the use of IoT capabilities to transform cities around the world into smart cities capable of providing location-aware services (e.g., finding buildings and streets, improving travel experience, obtaining security alerts) [11]. In order to create smart cities that can support reliable navigation services to people with special needs, researchers are creating complex workflows integrating a number of novel IoT elements, including video analytics, Bluetooth beacons, mobile computing, and LiDAR-scanned 3D semantic models. For example, we may have a streaming application workflow that analyzes video feeds from the surveillance cameras of the streets in a timely manner to evaluate the density of crowds in different parts of the city to help select path choices. Specially, ASD individuals may prefer to choose paths that have less dense crowds due to psychological factors; people with visual impairment try to avoid large open spaces due to the difficulty of finding references

for localization; and people in wheelchairs can navigate along paths with fewer crowds far more conveniently than along those with large crowds. This information is then combined with a 3D model and the location of the user to calculate the best path to reach the desired destination. Additionally, we need to continuously monitor the user (e.g., using the Bluetooth beacons), and the streets (e.g., using surveillance cameras) to adapt to changes.

Data-driven workflow, such as the one described above, are very latency sensitive. In our use case, the navigation path needs to be computed in a timely manner to improve the quality of experience and allow users to meet planned schedules (for example, arrive in time to take a specific bus). In some cases we might need to adapt the path based on users' feedback. For example, if an ASD user gets stuck and panics at a certain location, the data-streaming application has to react following pre-defined or learned strategies such as re-route the path to avoid a current crowd, or move them to certain intermediate location to make them wait until the crowd passes.

Supporting workflows that require analyzing video analytics from a public space requires the need of transferring all the raw data to the cloud. This can lead to extra latencies that can affect users' quality of experience and may also result in privacy concerns.

2.2.2 Disaster Recovery

Our second use case is a disaster response use case.Disaster management is a process that involves four phases: *mitigation*, *preparedness*, *response*, and *recovery*. Mitigation efforts attempt to prevent hazards from developing into disasters altogether or to reduce the effects of disasters when they occur. In the preparedness phase, emergency managers develop plans of action when the disaster strikes and analyze and manage required resources. The response phase executes the action plans, which include the mobilization of the necessary emergency services and dispatch of first responders and other material resources in the disaster area. Finally, the aim of the recovery phase is to restore the affected area to its previous state.

This paper focuses on the response phase and use a multi-stage generic response

workflow that will be executed at the edge and at the core of the network. We start by capturing data from the affected zones (e.g LiDAR, photogrammetry, etc.) and we perform a preprocessing stage at the edge of the network. In our case, a minivan or a drone with networking and computational capabilities will be used to determine the content of the data and if any further post-processing is needed. If further processing is needed, data will be either sent to the cloud to perform a change detection with previously recorded historical data, store data into the cloud, or notify agencies to determine if building conditions are safe.

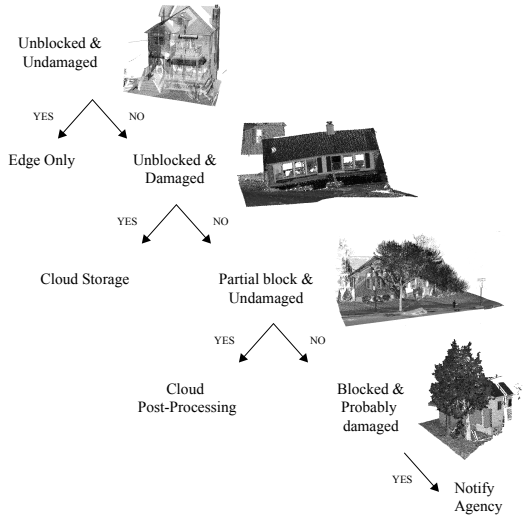

Figure 2.1: Disaster recovery decision stages and its associated reactions based on the LiDAR images.

This workflow presents a content-driven stage where the stream-processing engine needs to perform decisions based on the content of the data that is being processed. Figure 2.1 depicts all the different content-driven stages that our workflow presents in a decision tree way and its associated reactions.

The workflow consists of multiple different stages driven by the content of the data (more data content variety means more stages), but only two of them need stream-processing capabilities. The first stage with stream-processing needs is where data gets generated and pre-processed; this stage is performed at the edge of the network so we can quickly and efficiently determine whether the building conditions are safe or not for evacuees to return. Depending on the results from this stage, we trigger the rest of the

stages. The second stage, that also needs stream processing, is the change detection application which is executed in the cloud.

Stage 1: Data generation - Pre-processing Stage

The following are the actual pre-processing stages that we implemented in Apache Storm to determine the conditions of the buildings.

Noise Filtering: flags or removes noise points in the LiDAR data.
Ground Point Classification: it classifies the LiDAR points into ground object points and non-ground object points.
Classification: it classifies the point features according to the geometry information.
Visual inspection: is designed to provide visual inspection, which allows experts or agencies to access the data and perform visual inspections and help perform more informed decisions.
Content-driven stage: the last stage on the workflow is a content-driven stage where, depending on the results of the data, we might need to react and trigger further processing to determine if a building is safe or not for evacuees to return. The decision of whether or not data needs post-processing is based on two metrics obtained from the first stage of the pre-processing workflow. The two metrics are: data quality and computation intensity.

Measurement of data quality:

$$DQ = \frac{Spacing}{step\_para} \tag{2.1}$$

where *Spacing* represents the average edge length (2D) to all neighbor points of the original LiDAR data and *step_para* represents to the cell size (or the grid size) parameter of the grid-based interpolation.

Measurement of computation intensity:

$$CI = \frac{FileSize}{(X\_lowe - X\_upper) \cdot (Y\_lowe - Y\_upper)} \tag{2.2}$$

If both the data quality and the computation intensity do not satisfy the specified threshold then further processing is required and the second stage will be performed.

Since this first stage will be performed at the edge of the network with limited computation capabilities, the user needs to have the ability to specify QoS metrics. In this case the user has the ability to specify a deadline that all the tuples have to meet, as one wants to get the results as fast as possible.

The following mathematical expressions are the task optimization model proposed for this workflow:

$$\begin{aligned} \max \sum x_{ij} \\ \sum t_{ij} \cdot x_{ij} \leq t_{constraint}, \forall i \\ x_{ij} \in 0, 1, \forall i, j \end{aligned} \tag{2.3}$$

Where x_{ij} detonates the jth tasks at the processing level i. $x_{ij} \in 0, 1$ where $x_{ij} = 1$ indicates the execution of the process, while $x_{ij} = 0$ represents not executing the process. t_{ij} denotes the estimated runtime of task x_{ij}. $t_{constraint}$ is the total workload budget for all tasks at level i.

Stage 2: Change detection - Post-processing stage

The following are the actual post-processing stages that we implemented and are triggered based on the content of the data.

Historical data: the first decision process to determine whether the data that needs further processing has any geo-spatial overlaps with the historical data that is currently stored in the system.

Change detection: the process that involves comparing changes between LiDAR photographs taken over different time periods that cover the exact same geographic area to understand how a given area has changed between two time periods.

Content-driven stage: this stage will notify agencies if the results produced are alarmingly atrocious.

To simulate this workflow we used real LiDAR images that were taken right after Hurricane Sandy struck back in 2012 in the NY and Long Island area, with a total of 741 images and 3.7 GB in size, with the biggest image size of 33.8 MB, and the smallest of 1.8 KB. For the historical data in stage 2 we used a bigger data set of pre-Hurricane Sandy. Supporting workflows that generate such large amounts of information at the edge of the network and having to transfer data back and forth between the edge and the core can prevent an effective reaction to an emergency situation and/or target application objectives.

2.2.3 Scientific Observatory

Our third use case is focused on scientific observatories, in particular the Ocean Observatory Initiative (OOI) [12]. OOI is a networked ocean research observatory with arrays of sensors and autonomous underwater vehicles. This networked system of instruments provide scientists the means to collect data sets, and enables the examination of complex cyber-physical processes. The scientific observatory use case presents similar timely constraints that prevents the sending large data products to the Cloud. This particularly affects timely delivery and transformation of data products into scientific insights.

2.2.4 Video Analytics

Our fourth use case is the use of video analytics for safety and security [13]. A standard video camera produces between 553 Mbps and 1.24 Gbps for a minute of video recording. The ability to record 4K video on cameras will push that number to grow exponentially in the upcoming years. The traditional model to send all the data to the Cloud is not efficient enough to support such video data analytics [14]. Video Analytics pipelines need to be performed using edge, in-transit and cloud resources in order to cater to low latency requirement for large-scale video streams [15].

2.2.5 Observe Orient Decide Act Loop

Out last use case is the OODA loop. The Observe Orient Decide Act (OODA) loop refers to the decision-making cycle of observe, orient, decide, and act, developed by military strategists and the United States Air Force [16]. OODA is a decision-making cycle to process data streaming from sensors in real time, becoming an essential design characteristic for IoT applications.

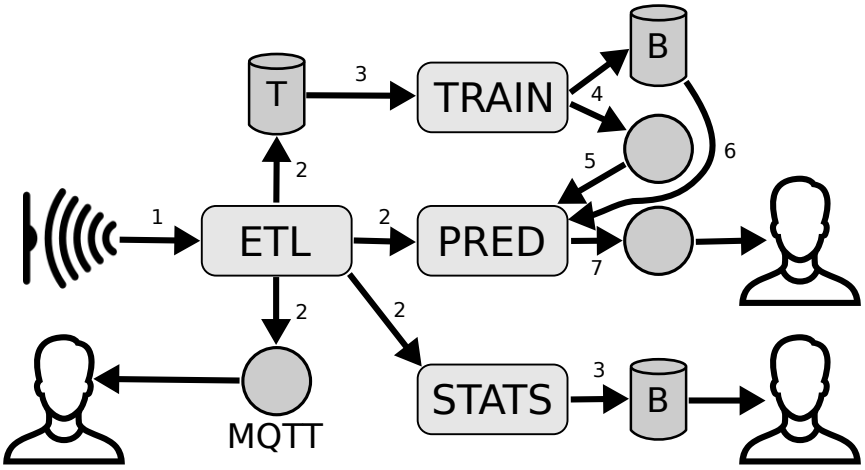

Figure 2.2: RIoTBench IoT high-level logical interactions between different sensors, applications and users.

Anshu *et al.* [17] offer a suite of IoT applications that follows the closed-loop OODA cycle. The applications are based on common IoT patterns for data pre-processing, statistical summarization, and predictive analytics. These are coupled with workloads sourced from real IoT observations. A high-level overview of the logical interaction of the IoT applications is depicted in Figure 2.2.

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

Chapter 3

Background and Related Work

3.1 Introduction

Cloud computing was introduced a decade ago with the promise of seemingly infinite computing resources available on demand [18]. This model has proved to be effective for scaling up search engines[19], social networks[20], and content service providers[21] to billions of users around the world. However, this centralized model is being challenged by the emergence of a new computing paradigm and associated technologies i.e. Internet of Things (IoT).

The Internet of Things paradigm (IoT) fosters the connection of large numbers of sensors to the network. As the volume of data generated from the devices increases, moving data from the edge of the network to the Cloud might not be feasible due to bandwidth constraints [22]. Furthermore, as low latency and location-aware applications emerge [23], transfering all the data to the Cloud will not satisfy the low latency or location-aware constraints that the IoT applications expect. In addition, some applications, deal with sensitive and personal data, making it not possible to send the data to the Cloud due to privacy concerns [24]. For example, Toyota estimates that the amount of data flowing between vehicles and servers will reach 10 exabytes per month by 2025 [25]. Another example is commercial jets, which generate 10 TB of data for every 30 minutes of flight, making it impractical to transport all the data from the edge to the Cloud [26].

Edge computing has emerged as a potential approach for handling the large quantity of data generated by connected devices. It leverages the ability to execute computations and process data at the edge of the network, closer from the location of data producers.

Edge computing leverages smaller servers or single board computers that are widely distributed close to the edge to improve delays. Edge middleware is the essential software stack/architecture that serves as an interface between the Cloud and the IoT devices, supporting data discovery, communication and processing between edge devices and cloud services.

The realization of edge-based middleware platforms presents several conceptual and technical challenges. We believe that the seamless integration of edge and Cloud systems is one of the main challenges that prevent the efficient utilization of IoT. Without such an integration, developers must explicitly manage the platform as a unified set of resources to orchestrate computations, coordinate devices, and deliver data to users. As a result, there has been a substantial amount of research towards building edge-based middleware, addressing key crosscutting challenges, such as device discovery, scalability, and privacy and security. It is therefore important to understand the current state-of-the-art edge-based middleware and identify the gaps that may exist.

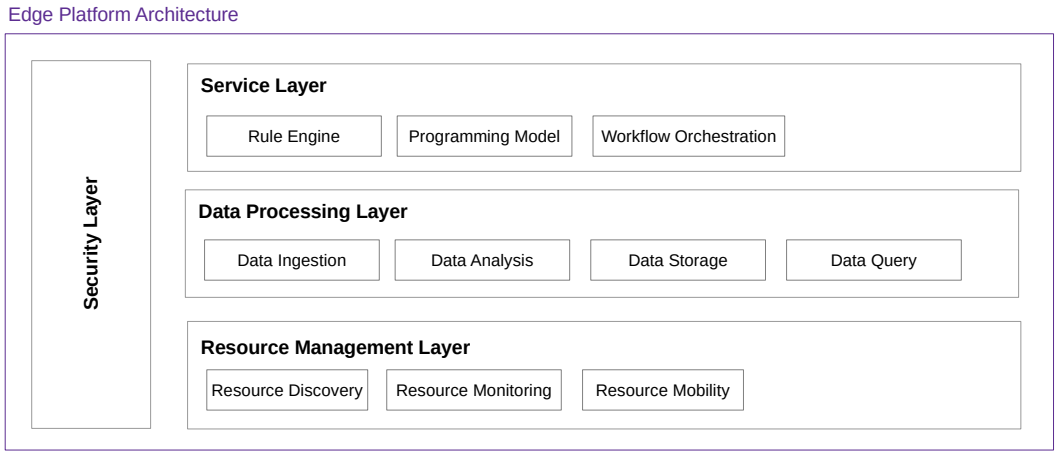

Figure 3.1: Edge-based middleware reference architecture consisting of four layers, each of them with their respective components.

3.2 Edge-based Middleware Architecture

Extensive research and development have been put into creating edge-based middleware systems. There are currently more than 100 edge-based middleware platforms in the market today and the number is continuously growing [27]. However, not every platform

is designed with the same capabilities or architecture. Despite the diversity and large number of edge-based middleware systems, two common architectures emerge:

The majority of IoT platform's architecture follows the Cloud-centric approach. They are built on the premise that ingestion, management, and processing of IoT data can be done in the Cloud, without any edge computing capabilities. Some examples are: Particle Cloud [28], Salesforce IoT Cloud [29] and If This Then That [30].

The other approach is the end-to-end architecture or edge-based middleware architecture built on the premise that edge-processing can save huge costs to clients.

In this survey, we focus on the end-to-end or edge-based middleware architecture, since the Cloud-centric approach will not be able to satisfy the requirement of the IoT applications presented in section 2.2. In order to compare and contrast all the existing state-of-the-art edge-based middlewares, we carefully studied the requirements and limitations of the IoT applications and came up with a four-layer edge-based middleware framework that satisfies all the requirements and limitations of the IoT applications, and each of the middlewares should consist. Figure 3.1 presents the layers and components that need to be included in an edge-computing solution. The edge-based middleware architecture is composed of four separate layers: resource management, data processing, service , and security.

Table 3.1: Design goals of the resource management layer components of the cited papers in this survey.

	Resource Discovery		Resource Monitoring		Resource Mobility	
Paper	Distributed	Low Overhead	Distributed	Low Overhead	Distributed	Low Overhead
Paganelli et al. [31]	✓	✓				
Liu et al. [32]	✓					
Cirani et al. [33]	✓	✓				
Jara et al. [34]		✓				
Zhou et al. [35]		✓				
Tanganelli et al. [36]			✓	✓		
Mäenpää et al. [37]			✓	✓		
SEGUE [38]					✓	✓
Chaufournier et al. [39]					✓	✓
Farris et al. [40]					✓	✓

3.2.1 Resource Management Layer

The resource management layer is dedicated to the discovery, identification and allocation of available resources. The challenge of the resource management layer is in managing these limited and geo-distributed resources efficiently. The resource management layer consists of the following components: resource discovery, resource monitoring, and resource mobility. Table 3.1 summarizes the design goals of each of the works focused on the resource management layer.

Design Goals

The following are the design goals that need to be taken into consideration when developing any of the components of the resource management layer.

Low Overhead: The algorithms and protocols of the resource management layer need to offer low runtime overhead when deployed in performance-limited hardware platforms.

Distributed: The resource management components also need to be designed in a distributed fashion in order to scale with the number of applications running in the system and with the number of IoT and edge devices.

Resource Discovery

The resource discovery component is responsible for efficiently identifying and discovering the geo-distributed IoT sensors. The following are some of the work focused on the resource discovery component.

Paganelli et al. [31] present a service for discovering Internet of Things resources. The service uses a peer-to-peer approach along with distributed hash table (DHT) techniques to support the discovery of distributed resources, the system guarantees scalability, robustness, and maintainability. Paganelli et al. meet both design goals since they use a distributed architecture by the means of a P2P architecture to support

the number of growing devices and offers a low-overhead algorithm.

Liu et al. [32] propose a distributed architecture for resource discovery, designed to be used in Machine-to-Machine applications. The architecture uses an overlay network composed of peer nodes to distribute workload,and eliminating the single point of failure. Liu et al. resource discovery mechanism only meet the distributed goal since the system is build using a peer-to-peer architecture to efficiently discover the resources in a decentralized manner. It does not meet the low overhead since it used HTTP to communicate and discover resources [41]. The reason being that HTTP runs on TCP, therefore it incurs all TCP connection overheads for connection establishment and closing [41].

Cirani et al. [33] also present a Peer-to-Peer architecture for service and resource discovery that can be applied for the Internet of Things applications. Cirani et al. resource discovery mechanism satisfies both goals since it used a distributed P2P architecture for discovering resources and it used CoAP a lightweight messaging system that uses UDP [41] for keeping track of all the resources.

Jara et al. [34] presents a centralized mechanism for discovering devices based on context and location. Jara et al. only satisfy the low overhead goal of the resource discovery mechanism since it uses a centralized architecture for discovering devices. It is well-known that centralized architectures have a single point-of-failure and present some scalability concerns when the number of IoT devices grows [32].

Zhou et al. [35] presents a service discovery algorithm and architecture designed for the Internet of Things. The work focuses on the context and location aware discovery. They first present an architecture called "Digcovery" to support the large number of IoT devices. And finally they present a search engine to offer query, look-up and filtering support. Zhou et al. only satisfy the low-overhead goal since the resource discovery mechanism claims that the algorithm has good scalability, and it can be applied to different fields. Only the domain ontology needs to be replaced.

Resource Monitoring

The resource monitoring component is responsible for controlling and managing hardware and software infrastructures. It also provides information and performance indicators for both platforms and applications to assist in the decision of allocating the resources. In addition, it monitors the state of the resources in the event of failure. The following are some of the work focused only on the resource monitoring component.

Tanganelli et al. [36] propose an edge-centric architecture that uses the CoRE Resource Directory interface and the CoAP protocol to enable resource monitoring and discovery for IoT applications. This approach is able to satisfy both design goals since it is distributed, in the means of a P2P network, and achieves low overhead since they run their experiments on emulated embedded devices and achieve millisecond latencies.

Mäenpää et al. [37] propose an architecture that focuses on the resource discovery and monitoring of wide area sensors and actuators. The architecture enables a federation of geographically distributed Wireless Sensor Networks (WSNs) using a peer-to-peer network. This approach satisfies both design goals.

Resource Mobility

The resource mobility component is responsible for moving computations between edge nodes in order to achieve the requirements of the IoT applications. The following are some of the work focused only on the resource mobility component.

SEGUE [38] is a migration system, that achieves optimal migration decisions by using the Markov Decision Process (MDP) to perform migration decisions. SEGUE meets all the design goals since it was carefully evaluated to showcase the real-time performance, scalability, and dynamicity by using real mobility trace of 320 taxis in Rome.

Chaufournier et al. [39] relies on multi-path TCP, an effort to use multiple paths to maximize resource usage and increase redundancy. This techniques aims at improving the migration time of virtual machines. Chaufournier et al. resource mobility approach also achieves all the goals since it proposed the uses of multi-path TCP and claims that

increases the migration throughput by 6x and reduces the time by 50% in some cases.

Farris et al. [40],presents two Integer Linear Problem optimization schemes, with the pourpus of reducing the quality of service when performing migrations at the edge of the network. Farris et al. resource mobility also meets all the goals since it was designed to cope with the limitation of resource-constrained edge nodes and showcased the scalability in terms of users and the dynamicity of the algorithm.

Table 3.2: Design goals of the data processing layer components of the cited papers in this survey.

	Data Ingestion			Data Analysis			Data Storage			Data Query		
Project	Real-Time	Distributed	Scalable	Real-Time	Distributed	Scalable	Real-Time	Distributed	Scalable	Real-Time	Distributed	Scalable
Apache Kafka [42]		✓	✓									
Mosquitto [43]		✓	✓									
RabbitMQ [44]		✓										
ActiveMQ [45]		✓										
Heron [46]					✓	✓						
Storm [47]					✓	✓						
Flink [48]					✓	✓						
MillWheel [49]					✓	✓						
Spark [50]					✓	✓						
ApacheEdgent [51]				✓	✓	✓						
LMC [52]				✓		✓						
DataFlog [53]							✓	✓	✓	✓	✓	✓
FogStore [54, 55]							✓	✓	✓	✓	✓	✓
Moon et. al. [56]											✓	✓
IOTMDB [57]											✓	✓

3.2.2 Data Processing Layer

The data processing layer is in charge of the consolidation of data from multiple producers, along with its processing and delivery. Current approaches in data processing are known to be data-intensive process. The frequent operations on disk results in the inability to perform real-time data analytics when executed on edge constrained devices. The data processing layer consists of four components: ingestion, analysis, storage, and query. Table 3.2 summarizes the design goals of each of the works focused

on the data processing layer.

Design Goals

The following are the design goals that need to be taken into consideration when designing any of the components of the data processing layer.

Distributed: The data processing components should support distributed information processing, distributed computing capabilities, distributed storage and query. In order to address the variable number of devices, services, and users at any given point in time.

Scalable: The scalable term refers to the ability of the data processing component to handle a growing number of clients. The data processing components also need to be scalable also to address the needs of a variable number of devices, services, and users.

Real-Time: The real-time term refers to the ability to achieve low processing latency as the number of messages increases. The data processing components need to be able to process data in real-time either in edge constrained devices such as Raspberry Pis or smartphones and the cloud. Since IoT applications are latency sensitive as we described in section 2.2.

Data Ingestion

The data ingestion component aggregates data from multiple producers and sources in order to enable processing through pipelines. The following works focused solely on the data ingestion component.

Apache Kafka [42] is one of the most popular frameworks available, it is an open-source framework used for building real-time data pipelines and streaming apps. Apache Kafka meets three of the four design goals, the reason Apache Kafka does not offer real-time processing at the edge of the network, because it was not designed to be deployed in constrained devices, as it was demonstrated in this work[58].

Mosquitto [43] is a lightweight open-source publish/subscribe messaging broker designed for the Internet of Things. It implements the lightweight MQTT protocol, to transport the messages, making it suitable for low-power devices. Even though Mosquitto was created for the need to achieve real-time message handling, Scalagent published a survey where they stress test the Mosquitto and show that it can succeed at handling 60,000 publishers but it requires high transmission latency and high CPU usage [59]. For those reasons Mosquitto only satisfies the scalability and distributed and scalable design goals.

RabbitMQ [44] is also a lightweight publish/subscribe messaging broker, designed to be deployed in the cloud.Similarly to Mosquitto, RabbitMQ in the scaleagent tests shows that it can only handle 8,000 publishers producing 8,000 messages per second, and is not able to achieve real-time analytics [59]. In this case RabbitMQ only satisfies the distributed goal since it can only support 8,000 publishers, and as mentioned earlier, current city-scale experimental research facilities envision the deployment of 20,000 to 40,000 sensors [60].

ActiveMQ [45] its an open-source messaging broker, that supports numerous industry-standard protocols. ActiveMQ also suffers from the same problems as RabbitMQ since it has high message transmission latency and cannot handle more than 20.000 publishers [59].

Data Analysis

Data analysis is the process of analyzing large volumes of data to discover useful information and perform informed decisions. The following are some of the work focused on the data analysis component.

Heron [46] is a real-time analytics platform developed by Twitter. It is designed for speedy performance, low latency, isolation, and reliability. Heron meets all the design goals except for the real-time data analytics at the edge because it was designed to be deployed in large clusters at the core of the network.

Apache Storm [47] is an open-source distributed real-time stream processing system. Similarly, Storm was also designed to be deployed in the Cloud.

Flink [48] is a distributed processing engine for performing stream processing applications over unbounded and bounded data streams. Flink was also designed to be deployed in the Cloud and not for the edge.

MillWheel [49] is a framework for building low-latency data-processing applications that was designed and build by Google. MillWheel, just like Heron, Storm, and Flink, was designed to be deployed in large clusters in the Cloud.

Spark [50] is an open-source distributed general-purpose stream processing and batch processing framework witch allow to perform in-memory analytics. Spark, just like Heron, Storm, and Flink, was designed to be deployed in large clusters in the Cloud.

Apache Edgent [51] is a micro-kernel framework designed to be deployed in small footprint edge devices, enabling local, real-time analytics at the edge of the network. Apache Edgent is a stream processing engine that was designed to be deployed on edge devices, allowing it to achieve all the design goals.

LMC [52] enables cross-platform code execution on constrained IoT devices. LCM meets the real-time and the scalable design goals since it was designed to constrained devices , but it doesn't meet the distributed goal since there is no currently not supported.

Data Storage

Due to the ever-increasing deployment of bandwidth-intensive IoT platforms (especially cameras), there is an increasing pressure on the bandwidth to transport data back and forth between the edge and the Cloud. There is a need for a more efficient management and computation of the data at the edge of the network. Building a storage system on an edge computing infrastructure has its own set of particular challenges. The wide geo-distribution and heterogeneous and constrained natures of this infrastructure require data-partitioning and replication policies that are commensurate with the latency requirements of the applications. The following are some of the work focused on the data storage component.

DataFlog [53] is a distributed indexing mechanism that performs data placement (both among edge nodes, and between the edge and the Cloud) based on spatiotemporal

attributes to support efficient queries involving multiple edge nodes. DataFlog is able to achieve all the design goals for the data storage layer since it uses distributed indexing mechanism, it supports efficient queries and it can scale since it uses a P2P network and can be deployed in any environment.

FogStore [54] [55] is distributed key-value storage system tailored for the edge of the network. FogStore uses a fog-aware replica placement, and a context-sensitive differential consistency strategies to satisfy the requirements of the Edge and the Fog. FogStore was designed by the same authors of DataFog and it also meets all the design goals, just like DataFlog.

Data Query

Similarly to the data storage, once data has been stored it needs to be accessed as well. The following are some of the research work on creating edge query systems. The following are some of the work focused on the data query component.

Moon et al. [56] propose a data management and searching system based on blockchain which ensures security. Moon et al. are able to meet all the goals for the data query layer except for the real-time design goal since they are using the blockchain Proof-of-Work consensus algorithm and it is known that the time to perform a computation does not increase linearly as the number of nodes increases.

IOTMDB [57] is an IoT storage solution based on NoSQL (Not Only SQL), to solve the storage and management problems of large volumes of IoT data. IOTMDB is able to satisfy all the design goals except for the real-time, since storing 1,000 records can take up to 2 seconds since other frameworks such as RocksDB can store 1,000 records in less than 60 ms [58].

3.2.3 Service Layer

The service Layer defines an application's set of available operations to the end user. The service layer is composed of three components: rule engine, programming model, and workflow orchestrator. Table 3.3 summarizes design goals of each of the works focused on the service layer.

Table 3.3: Design goals of the service layer components of the cited papers in this survey.

	Rule Engine		Programming Model		Workflow Orchestrator		
Paper	Scalable	Low Overhead	Expressive	Extensible	Dynamic	Scalable	Low Overhead
Chui et. al. [61]	✓	✓					
Lica et. al. [62]	✓						
Mobile-Fog [63]			✓	✓			
Rabel [64]			✓	✓			
Fabryq [65]			✓	✓			
FogFlow [66]			✓	✓			
Eidenbenz et al. [67]							✓
Taneja et al. [68]						✓	✓
DROPLET [69]					✓	✓	✓
Ghosh et al. [70]					✓		

Design Goals

The following are the design goals that need to be taken into consideration when creating any of the components of the service layer.

Low Overhead The service layer components need to be able to process data in real-time, i.e. providing fast analysis and data queries when deployed in performance-limited hardware platforms.

Scalable The service layer components need to offer good scalability, since there is going to be a large number of rules and a large number of operators that need to be placed.

Dynamic This design goal only applies to the workflow orchestrator. The workflow ochestrator needs to be able to orchestrate the workflows based on the runtime characteristics of the nodes.

Expressive This design goal only applies to the programming model. The programming model needs to be easy to express ideas, algorithms, tasks in an easy-to-read and succinct way.

Extensible This design goal also only applies to the programming model. The programming model needs to flexible enough that if new capabilities are needed, they can be added to the software without major changes to the underlying architecture.

Rule Engine

The Rule Engine makes it possible to evaluate data, perform decisions and trigger actions. The following are some of the work focused on the rule engine component.

Chui et al. [61] propose a rule-based system designed to support heterogeneous IoT devices. The rule-based system is based on Event-Condition-Action (ECA) rule mechanism with SOAP technology. Chui et al. rule engine satisfies all the design goals since they use an Event-Condition-Action (ECA) pattern which allows them to scale the system as the number of rules grows and achieve low overhead.

Lica et. al [62] propose a rule-based architecture that addresses the main issues involved in application management in the Internet of Things. Lica et al. approach satisfies the expressive and extensible goals since further developments are necessary to improve the architecture effectiveness before its final implementation is carried out.

Programming Models

Due to the high dynamicity of edge resource, heterogeneity of Cloud and edge resources deploying low latency and scalable applications can be tricky. For this reason, there is a need for high-level programming models that simplify the development of IoT applications across the edge and the Cloud. The following are some of the work focused on the programming model component.

Mobile-Fog [63] is a high-level programming model designed for applications that require large number of sensors and actuators and they are latency-sensitive. Mobile-Fog only satisfies both design goals since it uses a high-level API to program the sensors, making it easy to learn.

Ravel [64] proposes a programming model to program applications across embedded devices, edge nodes and cloud nodes by using an extension of the Model-View-Controller architecture. Ravel satisfies both design goals since it uses a high-level API.

Fabryq et al. [65] propose a proxy programming model to find and control sensors and actuators. Fabryq et al. approach also satisfies both design goals since it uses Javascript as the main programming language and also has a high-level API to program the sensors, making it easy to learn.

FogFlow [66] is a programming model that extends the dataflow programming model, allowing developers fast and easy development of edge and fog applications. FogFlow satisfies both design goals since it extends the Cloud dataflow programming model and makes it suitable for the edge environment, making it easy to learn.

Workflow Orchestrator

The Workflow Orchestrator consists of defining how to accommodate the application components (i.e., operators) on the available resources of the network topology to optimize one or more performance metrics [71]. The main challenge is to decide how to split the operators between the edge and Cloud in order to minimize the overall completion time. The workflow placement has been proved to be at least NP-Hard [7]. The following are some of the work focused only on the workflow orchestrator component.

Eidenbenz et al. [67] present an algorithm for the Series-Parallel-Decomposable Graphs (SPDG). Eidenbenz et al. only satisfy the real-time design goals since its only a theoretical approach.

Taneja et al. [68] propose an approach for deploying application across Cloud and edge resources by using a Module Mapping Algorithm. Taneja et al. meet all the design goals except for the dynamicity since the approach doesn't take into consideration network connectivity or failure of nodes.

DROPLET [69] is an algorithm, that partitions tasks across the edge and Cloud resources, while minimizing the total completion time. DROPLET achieves all the design goals since it is able to react and adapt to dynamic network events and is capable of performing real-time decisions and scale polynomially with increasing the number of operators to place.

Ghosh et al. [70] propose a Genetic Algorithm (GA) meta-heuristic for distributing analytics across edge and Cloud resources to support IoT applications. The main goal

of the genetic algorithm is to minimize the end-to-end latency. Ghosh et al. only meet one of the design goals since it takes between 1 - 26 seconds for placing 1 - 50 operators, making it not real-time or scalable when the number of operators grows.

R-Pulsar [71] we propose an operator placement strategy that aims to minimize an aggregate cost which covers the end-to-end latency, the data transfer rate and the messaging cost. The main differences between the R-Pulsar approach and other state of the art approaches is that, R-Pulsar focuses on optimizing three metrics 7, where the majority of the related work only focuses on end-to-end, bandwidth or both. Furthermore, the R-Pulsar approach implements a "knob" approach that allows the end-user to decide how much they want to optimize for each of the three techniques. Besides, all of that our R-Pulsar approach offers a machine learning model that constantly monitors the status of the operators, and if the requirements specified by the end-user are not met. a redeployment is performed. R-Pulsar meets all the design goals.

Table 3.4: Design goals of the security layer components of the cited papers in this survey.

Paper	End-to-End Security	Data Privacy
Lu. et al. [72]		✓
Shi et al. [73]		✓
Behrens et al. [74]	✓	
Mukherjee et al. [75]	✓	
Kothmayr et al. [76]	✓	

3.2.4 Security Layer

The fourth and last layer is the Security Layer, which consists of keeping the data generated by thousands of IoT devices private and secure. The following are some of the work focused on the end-to-end security component. Table 3.4 summarizes the work focused on the security layer.

Data privacy

Since the IoT produces large volumes of data easily available privacy protection in IoT its a challenge. The following are some of the work focused only on the data privacy

component.

Lu et al. [72] present a lightweight privacy-preserving data aggregation scheme designed to be used in constrained devices. The proposed aggregation schema uses the homomorphic Paillier encryption, Chinese Remainder Theorem, and one-way hash chain techniques to aggregate data.

Shi et al. [73] propose an algorithm that allows users to upload encrypted data to an untrusted aggregator, and allows the aggregator to decrypt statistics for each time interval.

End-to-End Security

In this section, we analyze the similarities and differences amongst all the currently available edge-based middleware systems that implement one or more of the layers of our edge-based middleware architecture. To do so we use the proposed edge platform architecture and the goals of each of the layers described in the previous section. Tables 3.5,3.6,3.7,3.8 summarize and offer more details on all the edge middleware surveyed systems, including the design goals that each component satisfies.

AWS Greengrass [77] is a software stack that allows to locally run computations, messaging, data caching, sync, and Machine Learning capabilities on devices in a secure way. AWS Greengrass consists of all the four layers presented in section 3.2. The main limitations of AWS Greengrass are the centralized architecture of the resource management layer, the lack of storage and query of the data processing layer, the use of a similar MQTT broker to Mosquitto for data ingestion violating the real-time design goal for the data processing layer, and the lack the workflow orchestration component in the service layer, leaving it to the end-user for the management and provisioning of the workflows.

Azure IoT Edge [78] is a collection of services designed to create end-to-end IoT applications on Azure Cloud. This service is meant for analyzing data at the edge of the network, instead of in the Cloud. Azure IoT, similarly to AWS, takes security very seriously, and uses certificate-based authentication as the primary mechanism for authentication for the Azure IoT Edge platform. Azure IoT also implements all four

layers proposed in section 3.1. Azure IoT only misses two components: the first one is the workflow orchestrator from the service layer and the second one is the data privacy at the security layer. Azure IoT uses a similar MQTT broker to Mosquitto making it not able to achieve real-time analytics.

EAaaS [79] is an analytics service that enables real-time edge analytics in IoT scenarios. The main focus of the EAaaS is the uses of a unified rule-based analytic model to simplify the user's programming efforts. In addition, they put a great amount of attention on making the system as lightweight and scalable as possible. EAaaS implements two of the four layers; it does not implement the resource management layer or the security layer and misses some components on the layers that it implements. The first components missing are from the data processing layer: EAaaS does not allow the storage or query of data at the edge of the network. From the service layer, EAaS does not implement the workflow orchestrator, forcing the end-user to decide where to place computations to achieve optimal performance.

Google Cloud IoT Edge [80] is a collection of services that allows users to manage, and consume IoT data from distributed devices at a large scale, and take actions as needed. Google Cloud IoT Edge follows the same path as the AWS Greengrass, implementing all four layers but missing some critical components on some of the layers. Google Cloud IoT Edge does not support the ability to store or query at the edge of the network. In addition, just like all the commercial systems surveyed so far, it also implements a similar broker to Mosquitto, violating the real-time design goal. Google Cloud IoT Edge does not offer the ability to orchestrate application between the edge and the Cloud.

Everyware IoT [81] is a comperical platform that provides an end-to-end IoT platform with propriotory software and hardware solutions. Everyware IoT implements all four layers but misses some critical components in all the layers. Similar to AWS, Azure, and Google, it lacks the query and storage support at the edge of the network and uses a similar Mosquitto broker for the data ingestion. Additionally, Everyware IoT lacks the rule engine of the service layer making it not possible to trigger or react to events that happen at the edge of the network.

Predix [82] is General Electric's commercial software platform for the collection and analysis of data from industrial machines. Predix implements all four layers but misses some critical components. Predix does not offer resource monitoring, storage, or query. In addition, it also doesn't offer the ability to orchestrate workflows between the edge and the cloud.

Bosch IoT [83] is a commercial end-to-end IoT platform that consists of multiple Cloud-enabled services and software packages. Bosch IoT implements all four layers but misses some components. Bosch IoT does not offer the rule engine or the workflow orchestrator of the service layer, and just like all other commercial systems it also implements a similar broker to Mosquitto.

Yanzi [84] is a commercial IoT platform designed to optimize office costs and productivity. Yanzi implements three of the four layers, lacking the resource management layer and some critical components on other layers. In the service layer, Yanzi misses the rule engine and the workflow orchestrator, and in the security layer, it misses the data privacy component.

R-Pulsar [85, 86] is an academic architecture that lets you run local analytics, messaging, data storage, and data querying capabilities on edge devices. R-Pulsar is the only one that satisfies all four layers with the most design goals. In addition, is the only architecture that has a full memory-mapped pipeline making it truly real-time. Also, it's one of the few that offers a unified architecture between the edge and the core, allowing it to seamlessly program the edge and the core. A limitation that the majority of the software stacks/architectures present is a split platform architecture between the edge and the core, leaving the end user to manage the scalability, replication, and distribution to the end user. For platforms that use a single architecture such as R-Pulsar, the system takes care of it so the user can focus on developing the application. R-Pulsar is also the only one to offer any application objectives, all the other software do not any application objectives.

FogHorn [87] is a commercial software platform that enables to run advanced analytics and machine learning applications at the edge of the network. FogHorn implements all four layers but misses some critical components in some of the layers. In the data

processing layer it misses the data storage and query components, not allowing the storage or query of data at the edge of the network. In the service layer, it misses the workflow orchestrator making the end user responsible for the management and provisioning of the resources and workflows. In the security layer, it misses the data privacy component.

GeeLytics [88] is an academic platform, which can perform real-time analytics either at the edge edge, or in the Cloud in a dynamic manner. Geelytics was designed to emphasize the service layer, in particular, the workflow orchestration component. Geelytics enables developers to run stream processing applications across the edge and the Cloud, without the need to consider where each task is located. GeeLytics implements two of the four layers, not implementing the security and the resource management layer. In addition, GeeLytics lacks the rule engine in the service layer and makes use of Mosquitto or Apache Kafka as the data ingestion data processing layer making it hard to scale or perform real-time analytics at the edge of the network.

Fogflow [66] is the evolution of GeeLytics, an academic framework that orchestrates workflows over the Cloud and the edge based on various context, including system context. For this second iteration they improved their workflow orchestration mechanism, added the missing rule engine component, and implemented the resource management layer. Some of the drawbacks existing on the previous version still have not been addressed, such as the use of Mosquitto or Kafka as the data ingestion component, limiting the scalability and the performance at the edge of the network.

OpenMTC [89] is a commercial open-source implementation of an IoT/M2M middleware with the focus on providing a standard-compliant platform. OpenMTC implements three of the four layers, missing the resource management layer. In the data processing layer it does not allow the storage or query of data at the edge of the network, and just like any other commercial approach, it uses a similar MQTT broker for the data ingestion. In addition in the service layer, it misses the workflow orchestration.

SiteWhere [90] is an industrial open-source platform, that uses a multi-tenant microservice-based infrastructure. SiteWhere implements all four layers and only misses very few

components on some of the layers. In the service layer, it lacks the workflow orchestration and in the security layer, it lacks data privacy.

SmartThings [91] is a commercial IoT platform designed for the smart houses. SmartThings implements three of the four layers missing the resource management layer and lacks some major components in some layers. In the data processing layer lacks the ability to store or query data at the edge of the network. In the service layer, it also lacks the workflow orchestration.

Kaa [92] is a commercial-grade IoT platform that is fully customizable. Kaa is one of the commercial systems more complete, implementing all four layers and missing very few components in some layers. The main drawback of Kaa is the lack of workflow orchestration between the edge and the Cloud, and the lack of data privacy in the security layer.

Samsung Artik [93] is a commercial IoT platform that focuses on unifying hardware, software, the cloud and the edge as a single ecosystem. Samsung Artick implements three of the four layers, missing the resource management layer. The main drawback is the lack of two of the key components in the data processing layer: the storage and query components. In addition, like all other commercial systems, Artick uses an MQTT broker similar to Mosquitto for the data ingestion component.

Ayla Network [94] is a commercial end-to-end IoT platform that includes a completely managed Cloud service. Ayla implements all the layers except for the resource management layer. In the data processing layer, it lacks the data storage and query and it uses an MQTT broker for the data ingestion layer. In addition, it also lacks the workflow orchestration component.

Altair SmartWorks [95] is a commercial platform designe as a Platform as a Service (PaaS) for Internet of Things projects, to collect data from objects, store it and build applications. Altair SmartWorks consists of three of the four layers, missing the data management layer. In the data processing layer, it lacks the ability to store and query data at the edge of the network. In also does not offer the ability to orchestrate workflows between the edge and the Cloud.

EdgeX [96] is a commercial open-source IoT microservice framework that allows

end uses to chose their sensors from a large ecosystem of 3rd party offerings. EdgeX implements all four layers but lacks some of the components in most layers. In the data processing layer, EdgeX does not support the data storage or query. In addition like all other commercial systems, EdgeX uses an MQTT broker for the data ingestion violating the real-time design goal. In the service layer, it lacks the workflow orchestration.

PiCasso [97] is an academic orchestration engine that deploys services based on specifications and resources availability. PiCasso implements all the layers except for the security layer. PiCasso puts a lot of emphasis in the service layer more, in particular, the workflow orchestration component. PiCasso lacks the storage and query components of the data processing layers.

Hua-Jun Hong et al. [98] is an academic fog computing platform that that focuses on the task distribution between the edge and the cloud. Hua-Jun Hong et al. approach implements three of the four layers, missing the security layer. In addition, it misses most of the components in all layers, since the main focus of this platform is to make deployment decisions to maximize the number of satisfied IoT analytics (operator deployment problem). In the data processing layer lacks the ability to store and query data at the edge of the network.

Cloud4IoT [99] is an academic platform that focuses on automatically deploying and orchestrating IoT applications. Cloud4IoT implements all the layers except for the security layer. Cloud4IoT to ease the code interoperability between the edge and the Cloud, to do that relies on commercial software that was designed to be deployed on a large cluster, making it hard to achieve real-time analytics at the edge of the network.

Nebulae [100] is a commercial end-to-end IoT platform, which the main focus in interoperability and inter-portability. Nebulae implements three of the four layers missing the resource management layer. In the data processing layer, it lacks the ability to store and query data at the edge of the network. In the service layer, it lacks the rule engine and the workflow orchestrator.

FogGIS [101] is an academic framework for improving throughput and reducing latency for analysis of geospatial data. FogGIS implements all the layers except for the

resource management layer. In addition, FogGIS data processing layer relies on a commercial system designed to be deployed on the Cloud not at the edge with constrained devices, making it hard to achieve real-time analytics.

FOG-engine [102] is an academic end-to-end platform for processing real-time analytics of data near where it is generated. FOG-engine implements two layers, not implementing the resource management and security layers, missing some key components on most layers. In the service layer, it lacks the orchestration and management of resources and workflows.

CEFIoT [103] is an academic end-to-end fault-tolerant architecture that reuses Cloud technologies at the edge of the network. CEFIoT implements two of the four layers, missing the resource management and security layers. In the service layer, it lacks the rule engine.

SAVI-IoT [104] is an academic self-managing programmable IoT platform that leverages both Hybrid Virtual Machines (HVV) and container isolation techniques to manage IoT applications. SAVI-IoT, just like CEFIoT, misses the same layers. The main difference is that SAVI-IoT does not offer a rule engine or a workflow orchestration. Another drawbacks of SAVI-IoT uses Kafka as the data ingestion component and Spark for the data analyses layer making them violate the real-time analytics at the edge of the network when deployed on constrained devices.

Foggy [105] is an academic architectural framework and software platform based on open-source technologies. Foggy main focus is the orchestration of application across the edge and the cloud. Foggy implements three of the four layers, missing the resource management layer. One of the main drawbacks of Foggy is the use of containers for orchestrating resources between the federated resource, making it no able to perform real-time analytics at the edge of the network. In addition, it lacks the ability to support storage and query at the edge of the network.

ISYMPHONY [106] is an academic orchestration framework designed for scaling real-time and on-demand IoT services. ISYMPHONY implements three of the four layers missing the security layer. ISYMPHONY focuses on the service layer in particular in the workflow orchestration layer. In the data processing layer, it lacks the data

storage and query components. In addition, it lacks the rule engine in the service layer.

Macchina.io [107] is a commercial IoT SDK that allows to connect sensors, actuators, Cloud services, mobile devices, and humans. Macchina.io implements three of the four layers, missing the resource management layer. In the service layer, it doesn't offer a rule-based engine or the workflow orchestration. In addition, Macchina.io relies on an MQTT broker similar to Mosquitto for the data ingestion layer.

Clearblade [108] is a commercial IoT platform to build scalable, secure enterprise IoT solutions. Clearblade implements three of the four layers, missing the resource management layer, and just like every other system it implements Mosquitto as their data ingestion component.

IBM Watson IoT Platform [109] is a commercial IoT platform that can connect and control IoT sensors, appliances, homes, and industries. The IBM Watson IoT Platform relies on the cloud to distribute and manage the edge analytics. IBM Watson IoT Platform implements all four layers proposed but misses the data storage and data query components of the data processing layer. Just like every other commercial systems surveyed above, it uses Mosquitto as their data ingestion component making it not scalable and real-time.

Table 3.5: Four Layer and Components for all the commercial and academic edge middleware available.

	Data Processing Layer				Resource Management Layer			Service Layer			Security Layer	
System	Data Ingestion	Data Analysis	Storage	Data Query	Resource Discovery	Resource Monitoring	Resource Mobility	Workflow Orchestrator	Rule Engine	Programming Model	Data privacy	End-to-End Security
AWS Greengrass [77]	✓	✓				✓			✓	✓		✓
Azure IoT Edge [78]	✓	✓	✓	✓		✓			✓	✓		✓
EAaaS [79]	✓	✓							✓	✓		
Google Cloud IoT [80]	✓	✓			✓	✓			✓	✓		✓
Everyware IoT [81]	✓	✓			✓	✓				✓		✓
Predix [82]	✓	✓			✓	✓			✓	✓		✓
Bosch IoT [83]	✓	✓	✓		✓	✓				✓		✓
Yanzi et. al. [84]	✓	✓								✓		✓
R-Pulsar [85, 86]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
FogHorn [87]	✓	✓				✓			✓	✓		✓
GeeLytics [88]	✓	✓								✓		
Fogflow [66]	✓	✓	✓	✓	✓	✓		✓	✓	✓		✓
OpenMTC [89]	✓	✓							✓	✓		✓
SiteWhere [90]	✓	✓	✓	✓	✓	✓			✓	✓		✓
SmartThings [91]	✓	✓							✓	✓		✓
Kaa [92]	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓
Samsung Artik [93]	✓	✓							✓	✓		✓
Ayla Network [94]	✓	✓							✓	✓		✓
Altair SmartWorks [95]	✓	✓							✓	✓		✓
EdgeX [96]	✓	✓			✓	✓			✓	✓		✓
PiCasso [97]	✓	✓			✓	✓		✓		✓		
Hua-Jun Hong et. al. [98]	✓	✓	✓	✓	✓	✓				✓		
Cloud4IoT [99]	✓	✓			✓			✓		✓		
Nebulae [100]	✓	✓								✓		✓
FogGIS [101]	✓	✓	✓	✓						✓		
FOG-engine [102]	✓	✓	✓	✓				✓		✓		
CEFIoT [103]	✓	✓	✓	✓				✓		✓		
SAVI-IoT [104]	✓	✓	✓	✓						✓		
Foggy [105]	✓	✓						✓		✓	✓	✓
ISYMPHONY [106]	✓	✓						✓		✓		
Macchina.io [107]	✓	✓	✓	✓						✓		✓
Clearblade [108]	✓	✓	✓	✓					✓	✓		✓
IBM Watson IoT [109]	✓	✓			✓	✓			✓	✓		✓

Table 3.6: Resource management layer and design goals for all the commercial and academic edge middleware available.

	Resource Discovery		Resource Monitoring		Resource Mobility	
System	Distributed	Low Overhead	Distributed	Low Overhead	Distributed	Low Overhead
AWS Greengrass [77]		✓		✓		
Azure IoT Edge [78]		✓		✓		
Google Cloud IoT [80]		✓		✓		
R-Pulsar [85, 86]	✓	✓	✓	✓	✓	✓
Fogflow [66]		✓	✓	✓		
SiteWhere [90]		✓		✓		
EdgeX [96]		✓		✓		
PiCasso [97]			✓	✓		
Hua-Jun Hong et. al. [98]				✓		
Cloud4IoT [99]	✓		✓			
CEFIoT [103]				✓		
Foggy [105]				✓		
ISYMPHONY [106]				✓		

Table 3.7: Data processing layer and design goals for all the commercial and academic edge middleware available.

	Data Ingestion			Data Analysis			Data Storage			Data Query		
System	Real-Time	Distributed	Scalable	Real-Time	Distributed	Scalable	Real-Time	Distributed	Scalable	Real-Time	Distributed	Scalable
AWS Greengrass [77]		✓	✓	✓	✓	✓						
Azure IoT Edge [78]		✓	✓	✓	✓	✓		✓	✓		✓	✓
EAaaS [79]	✓	✓	✓	✓	✓	✓						
Google Cloud IoT [80]		✓	✓		✓	✓						
Cisco IoT Cloud Connect [110]		✓	✓	✓	✓	✓						
Everyware IoT [81]		✓	✓		✓	✓						
Predix [82]		✓	✓		✓	✓						
Bosch IoT [83]		✓	✓		✓	✓						
Yanzi et. al. [84]		✓	✓		✓	✓						
R-Pulsar [85, 86]	✓	✓	✓				✓	✓	✓	✓	✓	✓
FogHorn [87]		✓	✓		✓	✓						
GeeLytics [88]		✓	✓		✓	✓	✓	✓	✓		✓	✓
Fogflow [66]		✓	✓		✓	✓	✓	✓	✓		✓	✓
OpenMTC [89]		✓	✓		✓	✓	✓	✓	✓		✓	✓
SiteWhere [90]		✓	✓		✓	✓						
SmartThings [91]		✓	✓		✓	✓						
Kaa [92]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Samsung Artik		✓	✓		✓	✓						
Ayla Network [94]		✓	✓		✓	✓						
Altair SmartWorks [95]		✓	✓		✓	✓						
EdgeX [96]		✓	✓		✓	✓						
PiCasso [97]					✓	✓						
Hua-Jun Hong et. al. [98]		✓	✓		✓	✓		✓	✓		✓	✓
Cloud4IoT [99]		✓	✓		✓	✓		✓	✓			
Nebulae [100]		✓	✓		✓	✓						
FogGIS [101]		✓	✓		✓	✓		✓	✓		✓	✓
FOG-engine [102]		✓	✓		✓	✓						
CEFIoT [103]		✓	✓		✓	✓						
SAVI-IoT [104]		✓	✓		✓	✓						
Foggy [105]		✓	✓		✓	✓						
ISYMPHONY [106]		✓	✓		✓	✓						
Macchina.io [107]		✓	✓		✓	✓		✓	✓		✓	✓
Clearblade [108]		✓	✓		✓	✓						
IBM Watson IoT [109]		✓	✓	✓	✓	✓						

Table 3.8: Service layer and design goals for all the commercial and academic edge middleware available.

	Rule Engine		Programming Model		Workflow Orchestrator		
System	Scalable	Low Overhead	Expressive	Extensible	Dynamic	Scalable	Low Overhead
AWS Greengrass [77]	✓	✓	✓	✓			
Azure IoT Edge [78]	✓	✓	✓	✓			
EAaaS [79]	✓	✓	✓	✓			
Google Cloud IoT [80]	✓	✓	✓	✓			
Cisco IoT Cloud Connect [110]			✓	✓			
Everyware IoT [81]	✓	✓	✓	✓			
Predix [82]	✓	✓	✓	✓			
Bosch IoT [83]	✓	✓	✓	✓			
Yanzi et. al. [84]			✓	✓			
R-Pulsar [85, 86]	✓	✓	✓	✓	✓	✓	✓
FogHorn [87]	✓	✓	✓	✓			
GeeLytics [88]			✓	✓			
Fogflow [66]	✓	✓	✓	✓	✓	✓	✓
OpenMTC [89]			✓	✓			
SiteWhere [90]	✓	✓	✓	✓			
SmartThings [91]	✓	✓	✓	✓			
Kaa [92]	✓	✓	✓	✓			
Samsung Artik [93]			✓	✓			
Ayla Network [94]	✓	✓	✓	✓			
Altair SmartWorks [95]			✓	✓			
EdgeX [96]	✓	✓	✓	✓			
PiCasso [97]			✓	✓	✓	✓	✓
Hua-Jun Hong et. al. [98]			✓	✓			
Cloud4IoT [99]			✓	✓	✓	✓	✓
Nebulae [100]	✓	✓	✓	✓			
FogGIS [101]			✓	✓			
FOG-engine [102]	✓	✓	✓	✓	✓	✓	✓
CEFIoT [103]			✓	✓		✓	
SAVI-IoT [104]			✓	✓		✓	
Foggy [105]			✓	✓	✓		✓
ISYMPHONY [106]			✓	✓	✓	✓	✓
Macchina.io [107]			✓	✓			
Clearblade [108]	✓	✓	✓	✓			
IBM Watson IoT [109]	✓	✓	✓	✓			

Chapter 4

Associative Rendezvous (AR)

4.1 Introduction

In this chapter, we describe the semantics and mechanisms that our programming abstraction builds upon, which enable developers to decide **what** and **where** data are collected and analyzed.

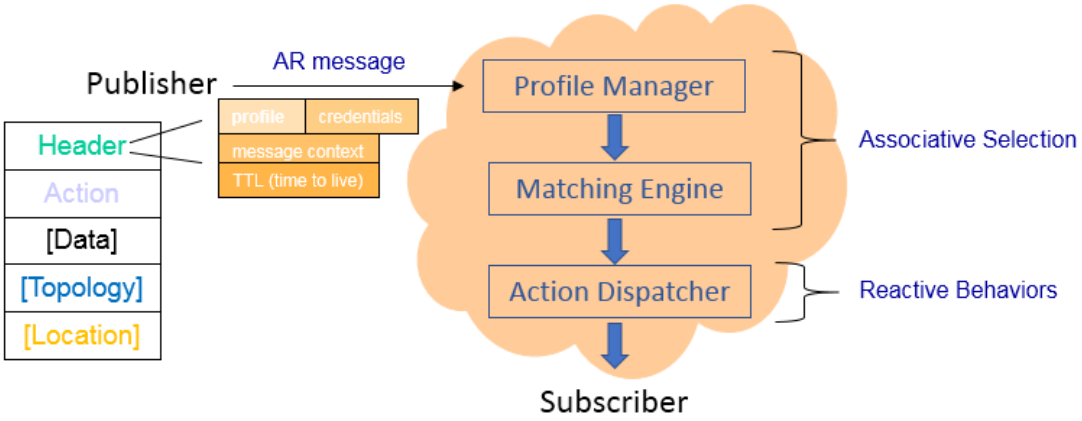

Figure 4.1: An illustration of the pub/sub model.

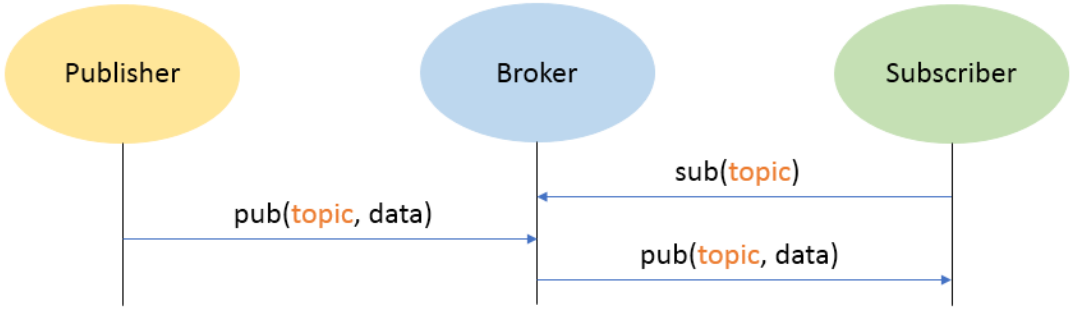

Figure 4.2: An illustration of the pub/sub model.

Publish/subscribe (pub/sub) systems are widely used in IoT applications, enabling

event-driven and asynchronous parallel processing while improving performance, reliability, and scalability. Pub/sub-protocols, however, make some common task in IoT more challenging to achieve, for instance, it is not straightforward to discover potential sensors of a particular type in a particular location. The lack of a discovery mechanism in the pub/sub its a challenge that needs to be addressed in order to enable the interoperability of different IoT data providers and producers. For those reasons, we chose the Associative Rendezvous (AR) paradigm. AR differs from the traditional publish/subscribe paradigms in that it does not rely on topics, it relays on well-defined combinations of keywords (i.e. keywords, partial keywords, wildcards, ranges) from a semantic information space to enable the discovery mechanism. More information about Associtive Rendezvous can be found in [111].

4.2 R-Pulsar Associative Rendezvous

R-Pulsar uses a custom implementation of the original AR semantics [112]. In our new implementation of the AR model, we modified two elements: the AR Message and the Reactive Behaviors.

4.2.1 AR Message

The AR message is now defined as a quintuplet instead of the original triplet: (header, action, data, location, and data-processing task). The location and data processing task fields have been added in to the already existing fields. The location field has been added so the AR message can be routed based on the location and the content, where the original version of AR only routes messages based on the content. The location coordinates represent the physical location of the sensors or the physical location of where to deploy data-processing tasks and the tag helps decide where to deploy data-processing tasks, either the edge or the cloud, allowing to pick from multiple cloud or edge geographically distributed resources.

Table 4.1: R-Pulsar Reactive behaviors

Actions	**Semantics**
store	Store data in the RP queue.
profiles	Notify sender all the interest_profiles stored in the RP point.
store_topology	Store the new topology in the RP.
start_topology	Start the topology in the RP.
stop_topology	Stop the topology.
delete_topology	Delete the topology.
notify_data	Match with already existing data/interest profiles.
notify_interest	Notify sender if there is someone interested in the data.
query_data	Allows to perform SQL-like queries on stored data.
delete_data	Remove all the stored data from the RP.
delete_interest	Remove all interest profiles from the RP.

4.2.2 Reactive Behaviors

For the reactive behaviors are now classified in two two different classes: resource actions and function actions. Where in the original implementation of AR there where only one type of actions. Table 4.1 summarizes the available actions.

Resource actions are designated for discovering, starting, and stopping sensors from transmitting data. Basic resource reactive behaviors currently defined include *notify_interest, notify_data, query_data*, and *delete*. The *notify_interest* is used by sensors for advertising its data producing capabilities and that they want to be notified when there is someone interested in the data they can produce. The *notify_data* are used by the data-processing tasks that will consume the data produced by sensors. When a *notify_data* profile and a *notify_interest* profile match, sensors are notified to start streaming data to the consumer. The *query_data* action is for performing SQL-like queries on stored data. The *delete* action deletes all matching profiles from the system.

Function actions are designated for storing, triggering, and stopping data-processing tasks. Basic function reactive behaviors currently defined include *store_function, start_function and stop_function*. The *store_function* action allows users to submit and store user-defined data-processing tasks in the RPs, allowing to share and discover existing data-processing tasks previously uploaded by other users. This avoids the need to rewrite the same function multiple times and facilitates the reproducibility of the experiments.

The *start_function* allows users to trigger data-processing tasks on demand. If there is a match between two profiles, the data-processing task is executed. The *stop_function* allows users to stop data-processing tasks that are running.

4.2.3 Illustrative Example

This section illustrates two operation examples of the R-Pulsar AR model. The first example in Figure 4.3 illustrates the exchange of messages for subscribing to sensors in R-Pulsar. The second example in Figure 4.4 demonstrates the one-to-many interactions using the R-Pulsar associative rendezvous.

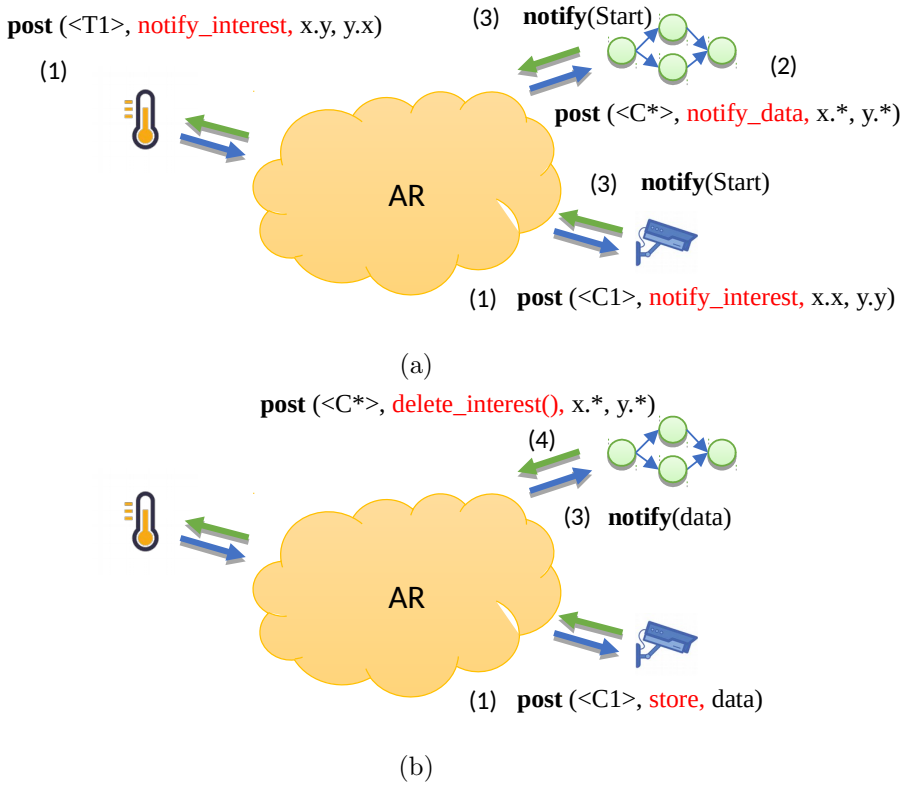

Figure 4.3: An example illustrating the operation of associative rendezvous.

For this example we have two different types of data producers/sensors in this case we have a temperature sensors and a CCTV camera as data producers. In step (1) both sensors perform is to register themselves into the system by advertising the type of data they can produce and where they are located, in the case of the temperature sensor its described by the profile $< T1 >$, x.y,y.x and $< C1 >$ x.x, y.y for the CCTV camera.

By doing that they are requesting to be notified if there are other clients interested in the type of data that they can produce. Both interest profiles are stored in the system, and matched against existing interest profiles. Since there is no interest profiles stored in the system nothing else happens. Both data producers/sensors publishes data in the system only if other clients need it. In step (2) the data consumer/computation is interested in consuming a very specific type of data, in this case it is interested in consuming data that matches the profile $< C* >$ and it is located in x.*,y.*., requesting to be notified if there are data stored in the system matching the profile. The interest profile of the computation is stored in the system and matched against the other profiles in the system. Since the notify data profile matches the profile of the CCTV camera, in step (3) a notification message is sent to the CCTV camera that someone is interested in its data and to start pushing the data into the system. In Figure 4.3b step (4) the camera starts publishing data in the system, the data published by the camera matches the data profile specified by the computation, resulting in step (5) the data being send to the computation for processing. After a few minutes the computation decide that the data the CCTV camera is producing is no longer valuable and decides the unsubscribe from it by sending a delete interest in step (6), pushing a notification to the CCTV camera to sop pushing data to the system.

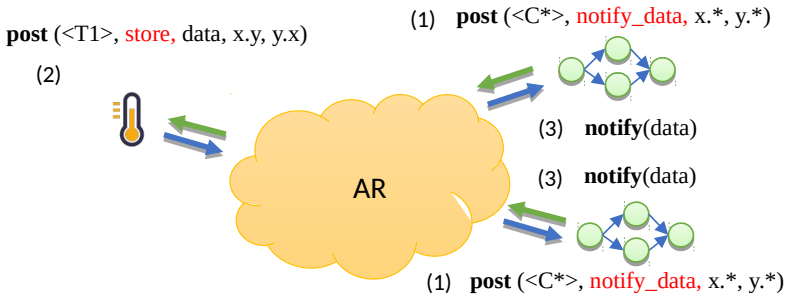

Figure 4.4: One-to-many interactions using associative rendezvous.

Figure 4.4 illustrates a one-to-many (e.g. multicast) interaction using AR. The example assumes that the temperature sensor has already been registered in the system with the notify interest profile $< T1 >$. In step (1) the two data consumers register in the system requesting to be notified if there are data stored in the system matching the

profile. In step (2) the interest profile of the computation is stored in the system and matched against the other profiles in the system. Since the notify data profile matches the profile of the temperature sensor, a notification message is sent to the temperature sensor that someone is interested in its data and to start pushing the data into the system. In step (3) the temperature sensor published data into the system. In step (4) the data published by the sensor matches the notify data profile of the two consumers so both consumers are notified with the data.

Chapter 5

Enabling Data-driven IoT Applications

In this chapter, we present our concepts in which R-Pulsar have been build upon. R-Pulsar is an architecture that extends cloud capabilities to edge devices, allowing to collect and analyze data closer to the source of information and react autonomously to local events. R-Pulsar consists of four layers: (1) the infrastructure layer, (2) the federation layer, (3) the streaming layer, and (4) the application layer. Each of the layers consists of multiple components.

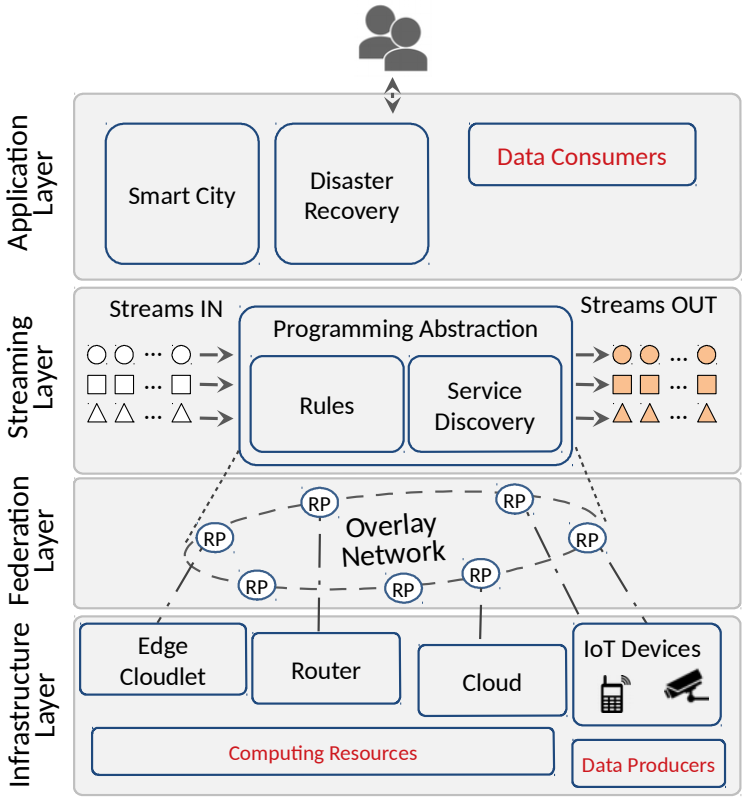

Figure 5.1: Schematic overview of the R-Pulsar Architecture

5.1 Infrastructure Layer

The infrastructure layer is composed of data producers and computational resources. The data producers are the various streaming sources that can generate data, including IoT devices (e.g., cameras, smart watch, and smart infrastructures, etc). The computational resources are a group of computers responsible for running the applications the user deploys. The resources are heterogeneous and distributed through the infrastructure, from the core to the edge of the network.

R-Pulsar uses a distributed architecture by the means of an overlay network, where each resource/node in the overlay network is called a Rendezvous Point (RP). RPs can be part of a public or private gateways located at the edge of the network or public or private server located in the cloud.

5.2 Federation Layer

The federation layer is responsible for orchestrating the geographically distributed resources composing the infrastructure. This layer is built using two main components: the location aware overlay component and the content based routing component.

Location-aware Overlay Network Component

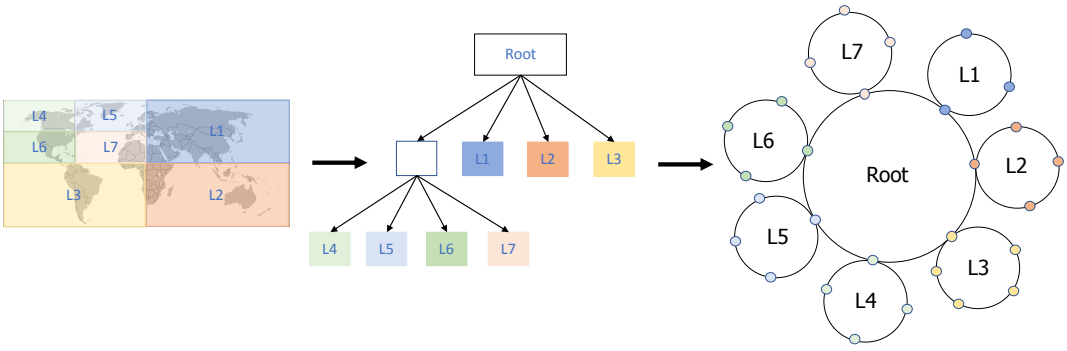

Figure 5.2: R-Pulsar quadtree geographical organization to multi layer P2P overlay network.

IoT data comes with temporal and spatial information, which is directly associated with their business value in a given context. Hence, IoT applications must process

data in a timely fashion and from proper locations. In order to process data from the right location R-Pulsar uses a location-aware overlay network in conjunction with a point quadtree to logically groups of RPs that are physically close together. A point quadtree is a tree data structure in which each internal node has exactly four children. Each node represents a 2D bounded box covering a specific part of the space to index, using a root node to cover the entire area.

The R-Pulsar overlay network is an n-dimensional self-organizing structured overlay composed of RP nodes. Peers in the overlay can join or leave the network at any time. Every node in the overlay is assigned a unique identifier that consists of a 160bit unique identifier. Each node stores the keys that maps to the segment of the curve between itself and its predecessor node.

During the initialization of the overlay network, the RP attempts to discover already existing RPs in the system and update its routing table. The joining RP sends a discovery message. If the message remains unanswered, the RP assumes that it is the first in the system and it becomes the master RP, creating a single overlay network (Peer-to-Peer network). Every time any other RP joins the overlay network and the master RP responds to the discovery message, the RP is added to the system by using the location of the RP and determining which quadrant the RP occupies. Once the initial P2P network has a sufficient number of RPs to guarantee that in case of multiple failures the P2P network will not disappear, the quadtree subdivides the overlay network into four additional P2P rings, plus an extra ring that will allow all the master RPs of each ring to communicate. Each RP master keeps a copy of the quadtree, so in the case of an RP failure the overlay network structure will never be lost. In the case of a master RPs failure, a master RP election is performed using the Hirschberg and Sinclair algorithm [113]. Figure 5.2 is a graphical representation of the quadtree and the logical organization of the P2P network.

In order to route a message the first strep is perform a quadtree query to decide in which of the P2P rings needs to be routed to. The locations in the AR message is used to perform a lookup in the tree and find the P2P ring closes to the given location.

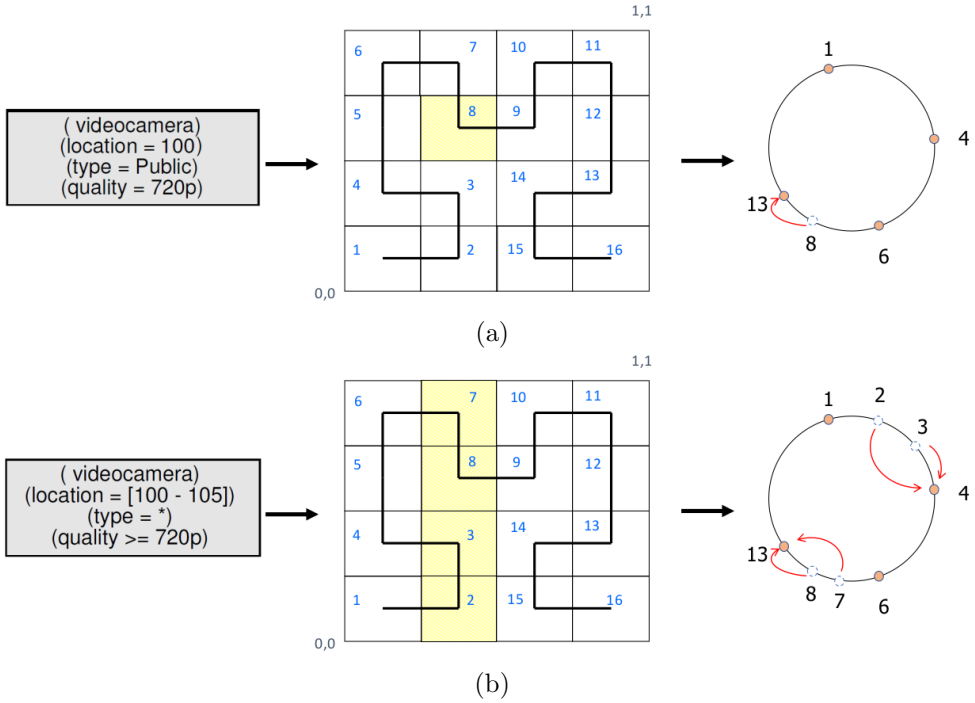

Figure 5.3: R-Pulsar space filling curve routing using simple (a) and complex (b) profiles.

Content-based Routing Component

This component builds on top of the location-aware overlay to route messages. The content-based routing component maps AR profiles onto single identifiers or clusters of identifiers to enable information discovery using partial knowledge, as described in [114].

Content-based routing is achieved by performing two steps: The first step is to encode the set of attributes and/or attribute-value pairs of the AR message, into a set of unique base 10 ids. The second step is to use the set of base 10 numbers produced in the previous step and pass them to the Hilbert Space Filling Curve (SFC). The SFC [115] is used to map the n-dimensional space of the AR profile to the one-dimensional space ID of the location-aware overlay network. Content based routing can be performed in two ways using simple keyword profiles or complex keyword profiles, the routing process is described below.

Routing using simple keyword profiles: The routing process consists of two steps. At the first step, the AR profile containing only exact attribute-value pairs is

encoded into a set of based 10 ID's, each id represents an exact attribute-value pair of the AR profile, then the set of base 10 ID's are passed to the SFC to obtain a single based 10 ID. This base 10 ID corresponds to 160bit unique identifier used by the P2P overlay network. Figure 5.3a illustrates this process.

Routing using complex keyword profiles: Similarly to the first step, an AR profile this time contains wildcards, ranges, or both is encoded and a set of base 10 ID's. The wildcard gets replaced for each of the possible value in the alphabet, producing multiple base 10 encoding sets. This sets of encoding are passed to the SFC to obtain a single based 10 ID, this step is repeated for each of the base 10 encoding sets. Then we end up with muntiple base 10 IDs that corresponds to several 160bit unique identifiers of the P2P overlay network. Figure 5.3b illustrates this process.

5.3 Streaming Layer

The streaming layer provides users and applications with efficient data-driven access to federated resources. This layer is composed of the serverless messaging component, memory-mapped streaming analytics component, the rule-based programming component and the operator placement component:

5.3.1 Serverless Messaging Component

Serverless computing is a cloud computing model that aims to abstract server management and low-level infrastructure decisions away from developers. Allowing to deploy and execute pieces of code in response to events without the need to specify IP addresses. The serverless messaging layer implements the AR interaction model described in Chapter 4 and it consists of two components: the matching engine and the profile manager.

The matching engine component is essentially responsible for matching profiles. If the result of the match is positive, then the action field of the incoming message is executed first, followed by the evaluation of the action field in the matched profiles.

The profile manager manages locally stored profiles and monitors message credentials and contexts to ensure that related constraints are satisfied.

Table 5.1: Available R-Pulsar operators.

Operators	Guarantees
bool **init()**	Connects to the existing P2P network and guarantees the discover all existing RPs
bool **post**(AR Message)	Routes the AR Message based on the profile and the location; Guarantees that all RPs responsible for that content in that location will receive the AR message.
bool **stream**(AR Message, String Peer Id)	Stream direct messages to a select RP. Guarantees that the responsible RP will always receive the message.
AR Message **poll**()	Retrieve messages from a select RP point. Guarantees that the consumer will always receive the message.

The serverless messaging component offers four different operators. Table 5.1 summarized all of the available operators.

The **init**() operator is used to join the R-Pulsar network and it guarantees that all the available RPs will be discovered.

The **stream**(AR Message, String Peer Id) operator is used to stream data directly to a specific RP, this operator bypasses the location and the content based routing component.

The **poll**() operator is used to retrieve the data other RPs are sending.

The **post**(AR message) operator is used to route the messages without the need of specifying the recipient of the message, the recipient of the message is resolved by using the AR Message.

5.3.2 Memory-mapped Streaming Analytics Component

The data pipeline is responsible for consolidating data, processing the data, and making them available to be used. State-of-the-art data pipelines are known to be data-intensive tasks, resulting in the inability to performing timely data analytics when deployed on constrained devices. The memory-mapped streaming analytics pipeline component is

motivated to overcome that issue. The streaming analytics pipeline comprises the following sub-components:

1. The data collection sub-component gathers data and brings them to the pipeline.
2. The stream processing sub-component processes the data and performs computations on the collected data.
3. The data storage and query sub-component reads and writes data to the main memory and disk.

Data Collection

Multiple data collection services are available, such as Apache Kafka [42], Google Pub-/Sub [80], Amazon Firehose [116], and Mosquitto [43]. Although some are designed to be deployed on edge devices, these services offer limited performance when deployed in constrained devices due to the limited read and write disk speeds.

We designed and implemented a custom data collection sub-component designed specifically for constrained devices using a memory-mapped queue. A memory mapped file is a segment of virtual memory that has been assigned a direct correlation with some portion of a file. This file is physically present on disk, which allows the operating system to ensure data access operations with better performance than standard file access. The core principle of the R-Pulsar queue system emerges from the observation that random memory read is about 3.5x faster than sequential disk read, as measured in Table 5.2. To perform the tests we used the Linux tool sysbench [117], a multi-threaded benchmark tool that allows to quickly get an impression about system performance.

The trade-off of using a memory mapped data collection system is that operating systems decides when to copy data from the main memory to disk.

Data Processing

R-Pulsar can be used on top of any data processing engine, allowing the end user to choose his or her favorite data-processing engine. The current release of R-Pulsar was

Table 5.2: Measurements of Disk I/O vs RAM memory performance on a Raspberry Pi.

Operation	Disk	RAM Memory
Sequential read	18.89 MB/s	631.34 MB/s
Sequential write	7.12 MB/s	573.65 MB/s
Random read	0.78 MB/s	65.96 MB/s
Random write	0.15 MB/s	65.88 MB/s

validated using Apache Edgent.

Data Storage and Query

This sub-component leverages the AR programming abstraction and a key-value database to offer SQL-like query capabilities. The storage sub-component uses the SFC of the content-based routing component to allow the ability to perform wildcard, range, or exact queries and allows the data to be horizontally partitioned among multiple RPs.

For storing data, R-Pulsar relies on RocksDB [118], an embedded key-value database optimized for fast and low-latency storage. The database keeps the most recently used data in the main memory and stores the least recently used data on disk.

5.3.3 Rule-based Programming Abstraction Component

The rule-based programming abstraction component makes it possible to build IoT applications and decide **when** data must be sent to the cloud for further post-processing without having to manage any infrastructure.

It consists of a rule engine that allows developers to specify IF-THEN rules that can trigger other data-processing tasks when a condition is satisfied. The THEN clause of the conditions sends an AR message with a custom profile to start and stop data-processing tasks on demand.

We created a rule-based system, which contains all of the appropriate knowledge encoded into a set of If-Then rules. The system examines all the rule conditions (IF) and determines a subset, the conflict set, of the rules whose conditions are satisfied based on the data tuples. Out of this conflict set, one of those rules is triggered (fired).

When a rule is fired, the action specified in its THEN clause is carried out. The loop for firing rules continues until one of two conditions are met: there are no more rules whose conditions are satisfied or a rule is fired. We allow to specify two different types of rules, ones that let you express data quality requirements which impose time constraints on the processing of the tuples, allowing the specification of a trade-off between the data quality and computational complexity. And the second one that offers the ability to express content-driven rules which complement the data quality requirements by triggering further stream-processing topologies either at the core or at the edge of the network if the data needs further processing due to quality of the data.

Content-Driven Rules

The content-driven rules consists of a single rule table that contains a set of rule entries installed by the developer. The rule entries contain a collection of conditions, a single action, and a single priority field.

The priority field is used in the event of having two or more rules that satisfy the condition, in order to brake the tie, only the one with highest priority will be executed. The condition field consists of one ore multiple antecedents (If clause). The antecedent of a rule consists of two parts: an object and its value. The object and its value are linked by an operator. The operator identifies the object and assigns the value. Table 5.3 summarized all the rule operators currently supported.

Table 5.3: Available R-Pulsar rule operators.

Operator	Definition
AND	Evaluates if two values or expressions are both true.
OR	Evaluates if at least one of multiple values or expressions is true.
NOT	Returns false for true and true for false.
>,<, >=,<= ==	Evaluates if a value is less than the value that follows this symbol. Evaluates if a value is greater than the value that follows this symbol. Evaluates if a value is less than or equal to the value that follows this symbol. Evaluates if a value is greater than or equal to the value that follows this symbol.
MIN	Returns true if the given number is the lowest number from a list of numbers.
MAX	Returns true if the given number is the highest number from a list of numbers.
AVG	Returns true if the given number is the avg number from a list of numbers.
STD	Returns true if the given number is the std number from a list of numbers.
IF	Determines if expressions are true or false. Returns a given value if true and another value if false.

The rule actions define what to do when the condition is satisfied, then the rule is fired and the action its performed. Each rule-entry has a single action associated with it; the three actions that are currently supported in our AR system are:

1. **Store** the results of the computation at the Edge or the Cloud. This allows the topology developers to seamlessly store the results of the topology based on the content of the data across a federated set of resources and have the ability to locate where those data results have been stored when needed.

2. **Trigger** a new Apache Storm topology, if it doesn't exist already, or route the tuples to an already running topology. Can be used to achieve multiple functionalities such as: split topologies/workflows across a set of participating Rendezvous Points that can be located at the core or at the edge of the network or make decisions based on the content of the data and triggering new computations.

3. **Notify** action allows to notify any node part of the overlay network and stream the results to them.

The rules can also be used to evaluate using a single tuple at a time or using window of tuples at a time. Windows is the concept in stream processing of splitting the infinite streams into finite chunks, and then apply computations to each chunk. There are two types of windows:

- **Sliding Window:** data elements are grouped in one or more windows that slides based on a specified interval.

- **Tumbling Window:** data elements are grouped in a single window based on the specified interval.

Windows can be specified accordingly to two different types of intervals: time and count.

- **Time:** can be used for both windows to group elements based on a period of time.

- **Count:** can be used for both windows to group elements in a defined size. In this case, all windows have the same size.

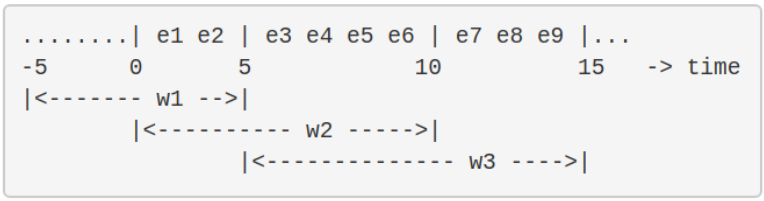

Figure 5.4: A time duration based sliding window with length 10 secs and sliding interval of 5 seconds.

Figure 5.4 depicts an example of a duration based sliding window with length of 10 secs and sliding interval of 5 seconds.

Figure 5.5 depicts a window is evaluated every five seconds and none of the windows overlap.

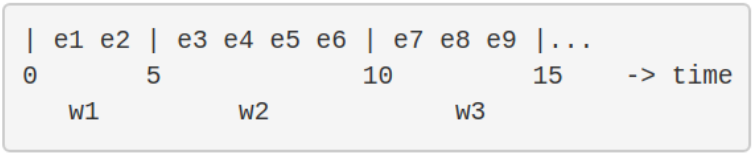

Figure 5.5: A time duration based tumbling window with length 5 secs..

Data Quality Rules

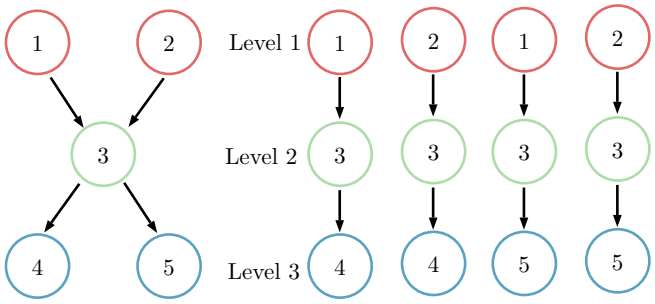

Figure 5.6: Storm topology linewise representation

The second type of rule are the data quality rules. Data quality rules are used in workflows that have multiple redundant computations, where each computations takes different amount of time and produces a different data quality. The rule system is

designed so users can specify time and data quality constraints so the QoS for the application can be meet.

The data quality rules are only supported when the Apache Storm engine is used, since it was build using Apache Storm. The content based rules consist of a custom Apache Storm stream grouping and the rule engine. In Apache Storm, part of defining a topology is to define how data is exchanged between components (how streams are consumed by the bolts). A Stream Grouping specifies which streams are consumed by each bolt. A stream grouping tells a topology how to send tuples between two components. The rule engine parses the given rule entries and computes all the possible paths that will satisfy the constraints specified by the rules. In order for the stream grouping to determine all the paths that will satisfy the constraints specified by the rules we developed in an offline training mode where the stream grouping collects execution information, transfers information, etc.. to determine all the paths that will satisfy the constraint. The data quality rule entries contain a tag filed and a deadline filed. The tag filed needs to be part of each of the tuples that needs to processed. The deadline filed specifies how much time each tuple with the corresponding tag has to go through the entire Storm topology. The algorithm consists of following steps: (1) breakdown the Storm topology into lines-paths from the task of the first level to the task of the last level through only one child on each level; (2) order lines according to their relative computing times T^l_{comp}. (3) iterate over the lines until one of the lines does not meet the deadline D and schedule the work.

The overall steps of the algorithm are depicted in Algorithm 1. In the first step of execution process the algorithm creates lines of the topology tasks and calculates the relative computing time T^l_{comp} for each line. T^l_{comp} is calculated as the sum of execution times T^l_{comp} of all tasks in the line. T^l_{comp} consists of task runtime T^t_{run} and total input data transfer time T^t_{data}. Then we order all the paths according to their relative computing times T^l_{comp} so we only need to check if T^l_{comp} satisfies the deadline D. Once we find one path that does not satisfy the deadline D, we have our set of paths that satisfies the constraint and we schedule the work. The algorithm is constantly running and reschedules the work if the computation conditions change.

Algorithm 1 Data quality rule algorithm

Input: workflow W
function LINES
    Break W into set of lines L. **return** L

Input: set of lines L
function REORDER
    for each l in L **Calculate** T^l_{comp}
    descending **order** L according to T^l_{comp}
    return L

Input: set of lines L
function COMPUTE
    Reorder(L)
While L is not empty:
    Take task L[1][1]
    if $T_{comp} \leq D$ **then return** l and Remove L[1][1] from L.

5.4 Application Layer

The application layer is composed of data consumers. Data consumers are the IoT applications described in Chapter 2. Data consumers process these the incoming data from the sensors/data producers and aggregate incoming data, send automatic alerts in a timely manner, or produce new streams of data that can be processed by other consumers.

Chapter 6

R-Pulsar, an edge-based middleware

6.1 Design

In this section we describe the design of R-Pulsar, Figure 6.1 depicts the interactions that R-Pulsar performs when a the Post primitive is used.

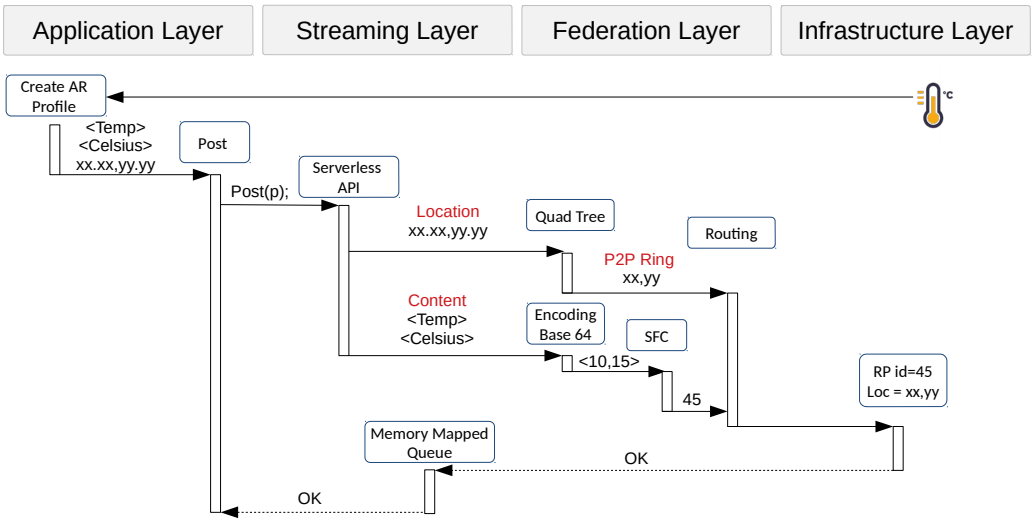

Figure 6.1: Sequence diagram of R-Pulsar, when a sensor wants to register itself in the R-Pulsar network.

The following are the steps that are taken when the post primitive is called:

- **Step 1:** The application layer defines an AR profile by specifying the type of sensor it is, and where is it located.
- **Step 2:** The application layer calls the post primitive with the profile specified in step 1.

- **Step 3:** The serverless API splits the profile passed by the post primitive in two separate calls: one for the content, and one of the location.
- **Step 4.1.1:** The location specified in the AR message is used for performing a location query in the quadtree. The result of the query is the closest P2P network given that point.
- **Step 4.2.1:** The content of the AR profile is used to select which RP will be responsible for the message. The content of the AR profile is encoded into a set of base 10 ids.
- **Step 4.2.2:** The set of base 10 ids are passed to the space filling curve (SFC) to go from n-dimensional space to 1-dimensional space id.
- **Step 5:** The base id of the SFC is used to route the message in to the P2P network obtained in step 4.1.1, and the message is forwarded to the right RP.

The current design of R-Pulsar uses a P2P architecture and a distributed hash table (DHT) which provides a decentralized key-value infrastructure for distributed applications.

6.2 Implementation

In this section we describe the implementation of R-Pulsar. Figure 6.2 depicts the dependencies of the main R-Pulsar classes. The current implementation of R-Pulsar is build using Java and builds upon the following open source projects: TomP2P for the P2P and DHT implementation and RocksDB for the memory mapped key value pair optimized for constrained devices. R-Pulsar consists of seven main packages that contain all the programming abstractions presented in chapter 5. Further documentation on the implementation and the code of R-Pulsar can be found on the GitHub [119].

- **Core Package:** The core package contains all the necessary code for starting and stopping the P2P rings, it holds the implementation the Associative Rendezvous programming abstraction and the memory mapped queuing system. The core

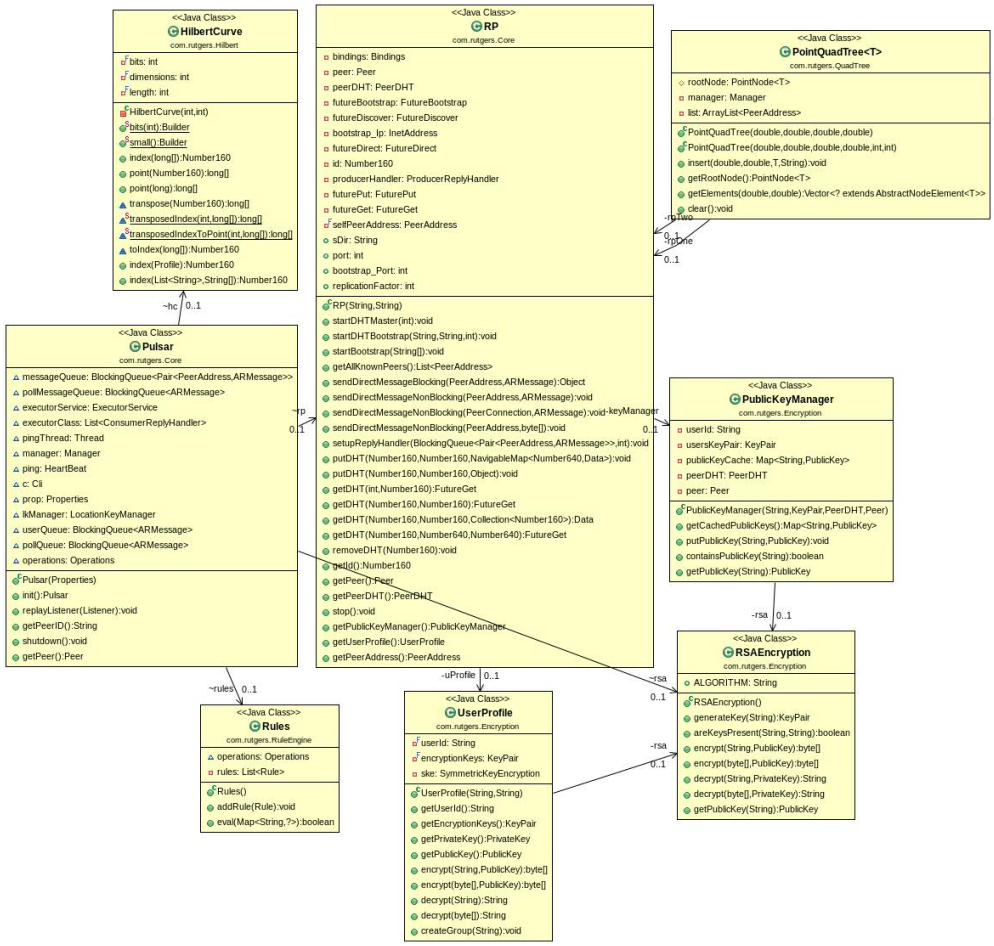

Figure 6.2: R-Pulsar main classes interaction.

package main responsibility is to abstract the AR messaging system and offering a simple API to the end user.

- **DB Package:** This package overwrites the database implementation of TomP2P to implement a memory mapped key value store. As mentioned earlier R-Pulsar storage system relies on RocksDB for storing data efficiently on constrained devices.

- **Encryption Package:** The encryption package offers RSA data encryption, so data can be encrypting so all the communications are safe.

- **Hilbert Package:** Contains the implementation of the space filling curve, R-Pulsar implements the Hilbert space filling curve. In addition the Hilbert package it contains all the necessary tools to encode the data to pass it to the Hilbert space

filling curve. For the encoding R-Pulsar uses the Base 64 for encoding the AR profiles so they can be passed to the Hilbert space filling curve.

- **Quadtree Package:** Contains the implementation of the quadtree which implements a point quadtree. The point quadtree has been created to accept latitude and longitude coordinates so RPs and messages will be delivered based on where they are located.

- **Rule Engine Package:** Contains a custom implementation of a rule engine with all the conditions that are currently supported. The rule engine consists of Operations and ActionDispatchers. The operations is a java class that allows to define evaluation conditions. The ActionDispatcher is another java class that allows to define the reactions of the rules.

- **Examples Package:** This package is not an essential package, but it simply contains a collection of examples demonstrating the usage of all the features that R-Pulsar offers.

6.3 API Examples

In this section, we present two sets of API examples, they are based on the disaster recovery use case presented in Chapter 2. The first set of API examples showcase the use of resource actions for sensor registration and discovery and the use of function actions for storing and triggering data-processing tasks. The second set of examples showcase the rule based programming abstraction.

The first set of examples uses the resource actions for enabling the exchange of data, without prior knowledge between devices. In Listing 6.1, an advertising profile is specified by the drone sensor with the type of data it can produce, and requests to be notified when someone is interested in such data.

```
1 profile.addSingle("Drone").addSingle("LiDAR");
2 ARMessage msg = ARMessage.newBuilder()
  .setAction(ARMessage.NOTIFY_INTEREST)
  .setLatitude(40.0583).setLongitude(-74.4056))
```

```
   .setProfile(profile);
3  producer.post(msg);
```

Listing 6.1: Data producer resource profile sample code.

Respectively, a data consumer declares the type of content of its interest. Listing 6.2 presents a data consumer with interest for any LiDAR sensor data that match the profile "Drone" and "Li*", located within the specified range (40*, 70*). As this profile matches the previous profile, the sensor from Listing 1 is notified that there is a consumer interested in its data, then the sensor starts streaming.

```
1  profile.addSingle("Drone").addSingle("Li*")
   .addSingle("lat:40*").addSingle("long:-74*");
2  ARMessage msg = ARMessage.newBuilder()
   .setProfile(profile).setAction
   (ARMessage.NOTIFY_DATA);
3  producer.post(msg);
```

Listing 6.2: Data consumer resource profile sample code.

As mentioned previously, profiles can be used in two different ways: for discovering resources or subscribing to data publishers (resource actions) and for deploying data-processing tasks across the edge and the cloud (function actions). Listing 6.3 showcases the deployment of a function (post_processing_func) in the system. This allows the developer to specify where/on which set of resources it should be deployed.

```
1  profile.addSingle("post_processing_func");
2  ARMessage msg = ARMessage.newBuilder()
   .setProfile(profile).setLocationTag("Cloud");
   .setAction(ARMessage.STORE_FUNCTION);
3  producer.post(msg);
```

Listing 6.3: Store post-processing task in the R-Pulsar overlay network.

Consequently, a profile and a decision (the IF-THEN rule) can be created to decide **when** to trigger the data-processing function (post_processing_func). In Listing 6.4,

the resulting action is created and attached to the function profile from Listing 6.5, which is sent when the rule is satisfied.

In addition, Listing 6.4 defines a rule that is constantly evaluated for every data element. If the condition of this rule is met, then the function profile from Listing 6.5 is forwarded, resulting in the execution (trigger) of the data-processing task previously stored.

```
1 Action topo1 = bluenew Reaction(T-profile);
2 Rule rule1 = bluenew Rule.Builder()
  .withCondition("IF(RESULT >= 10)")
  .withConsequence(topo1).withPriority(0);
```

Listing 6.4: Rule based programming abstraction for deploying the post-processing task.

```
1 T-profile.addSingle("post_processing_func");
2 ARMessage msg = ARMessage.newBuilder()
  .setAction(ARMessage.START_FUNCTION)
  .setProfile(T-profile);
3 producer.post(msg);
```

Listing 6.5: Profile for deploying the post-processing task.

The second set of examples showcase the rule based programming abstraction. The following snippet of code shows how developers will express the content-driven rules to trigger a new topology in the cloud:

```
Rule functional1 = new Rule.Builder()
.withCondition("IF(DATA_QUALITY < 5 OR
COMPUTATION_INTENSITY < 1)")
.withConsequence(new
TriggerTopologyReaction("PostProcTopo","cloud"))
.withPriority(1)
.build();
```

Figure 6.3: Trigger topology reaction rule definition.

The 'withCondition' is the IF rule expression that has to be satisfied and the 'withConsequence' is the action that will be triggered when the rule is satisfied. The action has two parameters: the first one specifies the name of the topology that needs to triggered if it is the first time executing the action or route to that topology if its already running. The second parameter specifies where it needs to be triggered. In this case it will be triggered on the cloud nodes that are part of the overlay network since its really computationally intensive.

Another brief code example of how developers will express the content-driven rules to store the matching results at the edge of the network:

```
Rule functional2 = new Rule.Builder()
.withCondition("IF(DATA_QUALITY >= 5 OR
COMPUTATION_INTENSITY >= 1)")
.withConsequence(new
StoreReaction("edge","batch"))
.withPriority(2)
.build();
```

Figure 6.4: Store reaction rule definition.

The action of this second rule also has two parameters: the first parameter specifies where to store the results of the streaming computation, which in this case, is at one of the edge nodes that are part of the overlay network since we want to be able to get our results quickly. The second parameter specifies how to store the results either one by one (streaming fashion) or batch (several at a time).

Figure 6.5 depicts the API to add a rule window with length 10 secs and sliding interval of 5 seconds.

```
builder.setBolt("RuleWindow", new
RuleEngineWindowBolt(rules.getRules(), bindings)
.withWindow(new Duration(10), new Duration(5)),
1);
```

Figure 6.5: Rule engine window bolt definition.

Figure 6.6 presents a brief code example of how developers will express the data quality rules:

```
Rule non-functional = new Rule.Builder()
.withTimeCondition("IF(TUPLE_TAG = '20x20')")
.withDeadlineInSeconds(30)
.build();
```

Figure 6.6: Tuple QoS rule definition

6.4 Location-aware Overlay Network Component Evaluation

In this section we evaluate the overhead and scalability of the location-aware overlay network component of R-Pulsar. The tests are evaluated using cloud and edge resources:

- Edge System: Chameleon Cloud [120], a configurable experimental environment for large-scale cloud research [120] with 100 instances of type m1.small (1 CPU and 2 GB RAM) to simulate the computation capabilities of a Raspberry Pi.
- Cloud System: Using Chameleon Cloud [120] with 100 instances of type m1.medium (2 CPU and 4 GB RAM).

6.4.1 Overhead and Scalability Over Cloud And Edge Systems

This experiment measures the overhead involved in performing a location-based query to identify relevant resources around the location of a client node. In our experiments, the location-based query used the GPS coordinates of each RP node and the client to identify the RP around the client. Not that for this experiment we did not deploy 500k RPs, we only used the location of 50k RPs to perform the queries.

Figure 6.7 shows that the overhead of performing a location-based query increases as the number of RP nodes in the system increases. Results show a maximum overhead of 10 milliseconds, which occurred when 50k RPs are in the system.

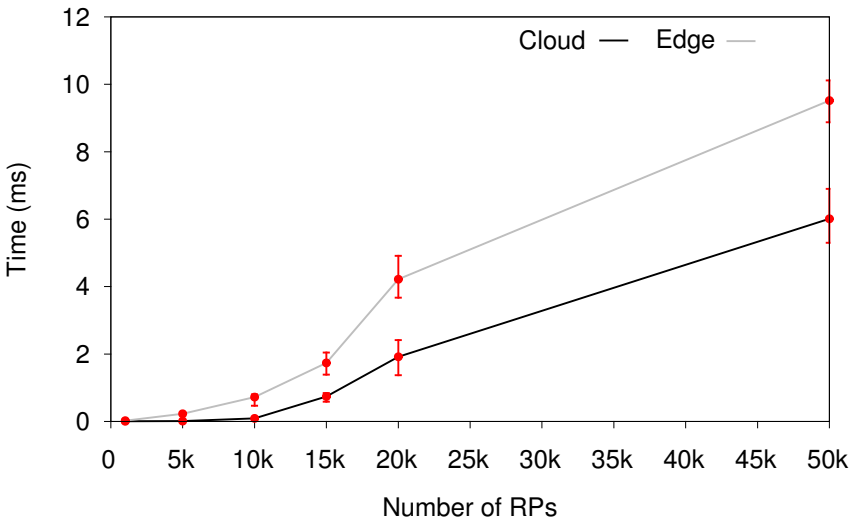

Figure 6.7: Location-based query overhead for different number of matches per query.

6.4.2 Overhead and Scalability of the Data Replication

Next, we measured the time required to replicate the information stored across a set of RP nodes in the system. In these experiments, we considered several scenarios involving different number of RP nodes as well as different number of replication factors. For this experiment we did deploy 25, 100 and 500 RPs, for the 5 RPs where deployed in the same instance in order to achieve 500 RP instances. Figure 6.8 collects the results.

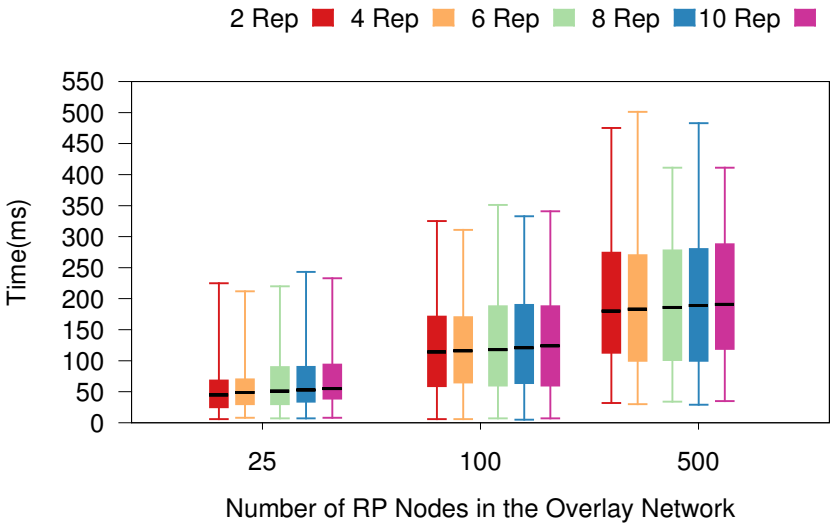

Figure 6.8: Time neccessary to update information of all RP nodes in the system after multiple topologies are registered.

We can observe that in Figure 6.8 that the time required to replicate the information

across all the RP nodes in the system is relatively low. In particular, it can vary from a few milliseconds to around 200 milliseconds on average when having a large system with 500 RP nodes. The main factor affecting the time required to propagate the information is the number of RP nodes, since the time to route a message increases. The time it takes to update the information on all RPs increases along with the error margins due to the fact that the cost of routing messages in a P2P network increases as the number of peers increases.

6.5 Content-based Routing Component Evaluation

In the content-based routing component we evaluate three aspects: The first aspect is the overhead of matching profiles, The second aspect we evaluate is the overhead and scalability of the routing, and finally we evaluate the time it takes to propagate information through the system when a new RP joins the system. The tests are evaluated using the same cloud resources as described in the previous section. For the edge resources of this set of experiments we used two different setups:

- Raspberry Pi System: Using 2 Raspberry Pi's 3 with 4x ARM Cortex-A53 1.2GHz, 1GB LPDDR2 of RAM and 10/100 Ethernet.
- Android System: Using 1 Motorola Moto G5 Plus with a Qualcomm Snapdragon 625 processor with 2.0 GHz octa-core CPU, 3GB of RAM

6.5.1 Profile Matching Overhead Over Cloud Systems

This first experiment measures the overhead involved in the profile matching operation at an RP node. We used a *notify_data* action message for these experiments. We considered different scenarios in which the system had different number of data profiles (i.e. subscriptions) and the request returned a different number of profile matches. The experiment was conducted using profiles that contained complex keyword tuples, wildcards and/or ranges. Figure 6.9 collects the results of these experiments.

In Figure 6.9 we can observe that, for a moderate database size of up to 10^4 profiles, the profile-matching operation incurs in an overhead between five and 10 milliseconds.

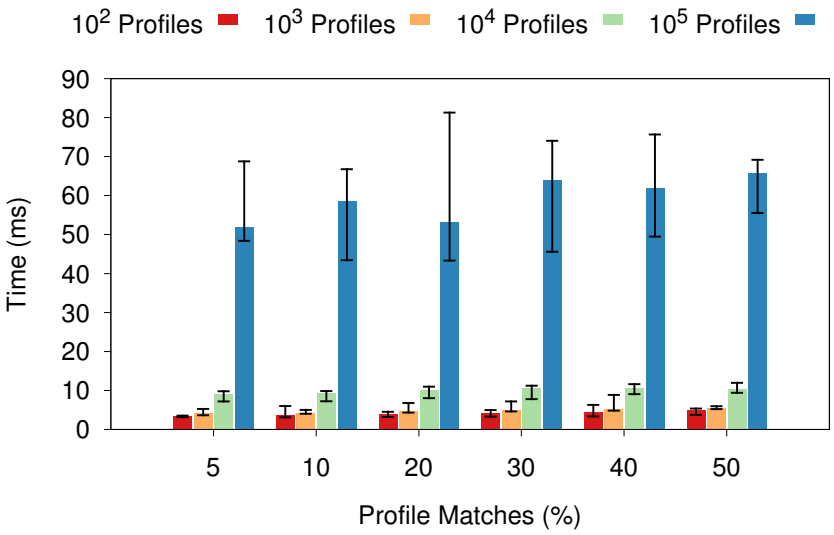

Figure 6.9: Profile matching overhead. Results are grouped based on the matching ratio expressed as a percentage of the total number of stored profiles.

Nonetheless, when we increase to 10^5 profiles, the overhead increases significantly, an this is due to the fact that we are increasing the memory and data access times required to identify and retrieve such a large number of profiles.

6.5.2 Routing Overhead and Scalability Over Edge Systems

This second set of experiments measure the routing overhead, which represents the time interval between an AR message is created until it is forwarded to the recipient of the message. It is important to maintain the routing overhead in the order of milliseconds to achieve timely analytics on constrained devices. The routing overhead was evaluated over Android and Raspberry PIs by simulating the storage or retrieval of data, as the number of RPs on a given region grows, and also as the AR profiles complexity grows. The profile complexity is defined in terms of the number of attribute-value pairs that make up the profile. For example, a 2D profile is composed of two properties, such as data type and location.

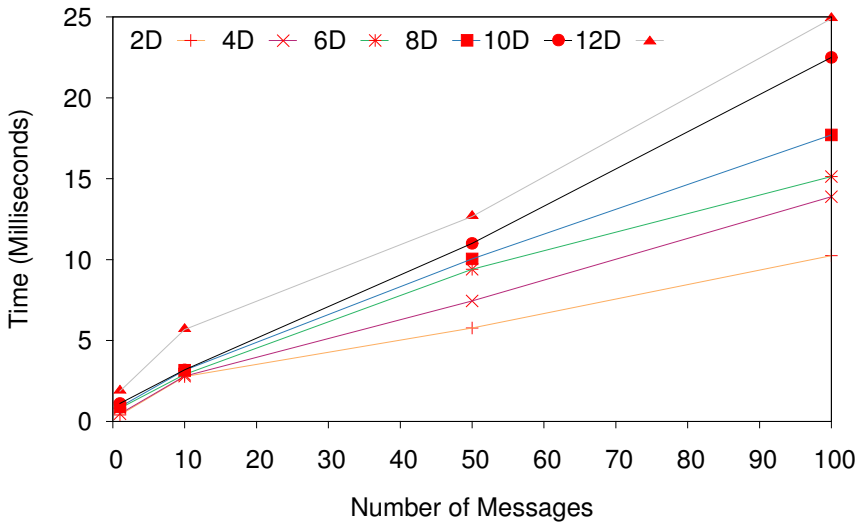

Figure 6.10: Evaluation of R-Pulsar space filling curve routing overhead and scalability as the number of messages and the complexity of profiles increase over the Android system.

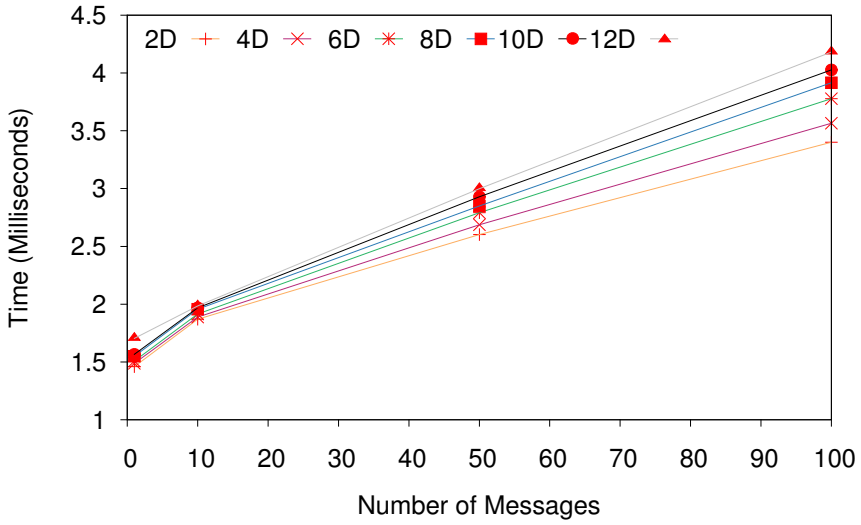

Figure 6.11: Evaluation of R-Pulsar space filling curve routing overhead and scalability as the number of messages and the complexity of profiles increase over the Raspberry Pi system.

In Figure 6.10 shows that when the profile complexity increases by a factor of 6, the time required to route messages increases by 2.5. Similarly, when the system increases the number of messages sent by a factor of 100, the time required to route one message

increases by a factor of 25. It shows that the routing overhead scales efficiently in both cases, as messages become increasingly complex and as the number of messages sent increases when using the Android system.

However, Figure 6.11 shows that when the profile complexity increases by a factor of 6, the time required to route messages increases by about 1.2, when using the Raspberry Pi system. Likewise, when the system increases the number of messages sent by a factor of 100, the time required to route one message increases by about 2.5. Demonstrating that the routing overhead scales more efficiently on a Raspberry Pi system than on an Android system.

6.5.3 Information Propagating through the System

This experiment is performed to understand the behavior of the system when multiple RP nodes request to join an existing deployment of our system. Specifically, we measured the time from when the RP nodes sent their joint message until all RP nodes existing in the system were updated. In our experiments, we varied both the number of RP nodes wanting to join and the number of existing RP nodes in the system. Figure 6.12 collects the results.

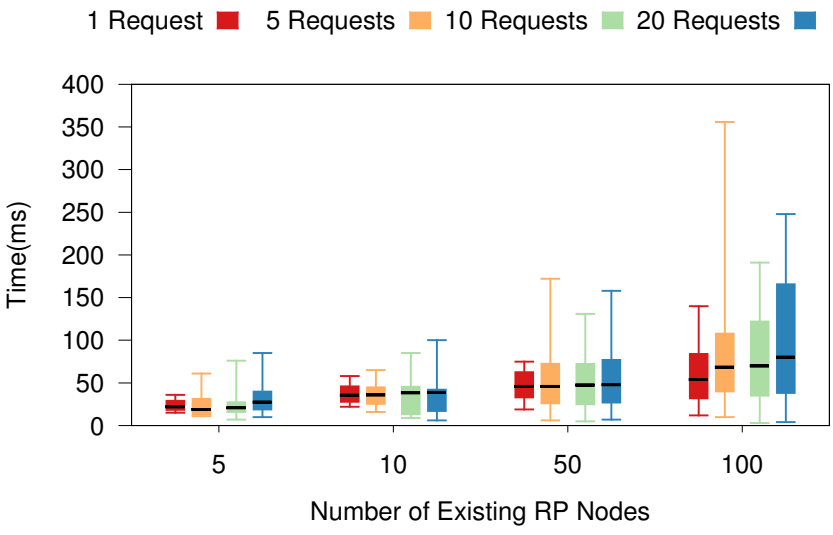

Figure 6.12: Time neccessary to update information of all RP nodes in the system after multiple new RP nodes join.

Figure 6.12 shows that the time required to have a consistent view of all the new RP

nodes after the requests are placed is relatively low, and that it scales properly when increasing the number of existing RP nodes. In the case of a small system with only five existing RP nodes the overhead is around 20 milliseconds on average. The overhead increases to around 70 milliseconds in the case of having a large system with 100 existing RP nodes. Additionally, we can observe that varying the number of requests does not significantly affect the time required to propagate the information in a system with up to 50 RP nodes. In the case of having 100 RP nodes, we observe that increasing the number of requests, increases the overhead as well as the dispersion around the average. Although it is important to maintain an updated view of the system to ensure that we can efficiently process clients' requests, it is not critical for the regular operations of the system. Our system has mechanisms to adapt to changes in the availability of RP nodes to improve data processing efficiency. Therefore, we consider that having an eventually consistent system is sufficient. Similar to the experiment 6.8 in this experiment we also observe that the time it takes to update small P2P rings is smaller than larger P2P rings and once again that is due to the fact that the more P2P nodes we have the more time it will take to route a message and the more potential for message collisions.

6.6 Memory-mapped Streaming Analytics Pipeline Evaluation

In this section we evaluate two of the sub-components that the memory-mapped streaming analytics pipeline consists, the Data collection and the storage and query sub-component. The tests are evaluated using the same cloud resources as in the location-aware overlay network component evaluation section. For the edge resources this set of experiments use the following resources:

- Raspberry Pi System: Using 2 Raspberry Pi's 3 with 4x ARM Cortex-A53 1.2GHz, 1GB LPDDR2 of RAM and 10/100 Ethernet.
- Android System: Using 1 Motorola Moto G5 Plus with a Qualcomm Snapdragon 625 processor with 2.0 GHz octa-core CPU, 3GB of RAM

6.6.1 Performance of the Data Collection Layer Over Edge Systems

The first experiment aims to evaluate the throughput of R-Pulsar messaging layer. This experiment was carried out using two Raspberry Pi's one as a producer and the other as the broker. The workload sizes are chosen based on the current limitations imposed by existing IoT services, such as AWS [121] and Azure [122]. The inner memory-mapped queue is compared to Apache Kafka and Mosquitto. Apache Kafka (the *de facto* standard for cloud and edge data analytics) [123, 124, 125], Mosquitto (a broker that can be found on edge frameworks, such as Azure IoT or the AWS Greengrass), and the R-Pulsar memory-mapped queue, using four different message sizes. The main difference between them is the way they store information: Apache Kafka and Mosquitto store messages on the disk while R-Pulsar stores them in the main memory and disk.

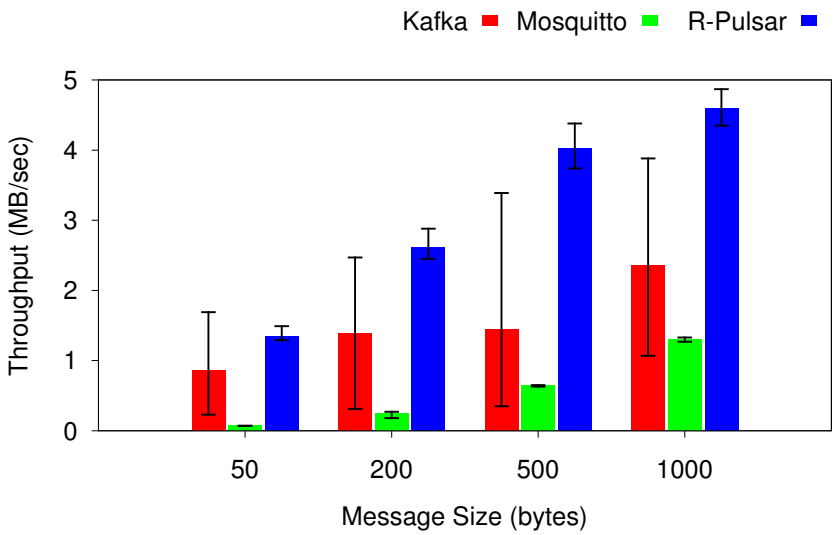

Figure 6.13: Single broker and producer throughput performance as message sizes grow. Comparison of R-Pulsar, Kafka, and Mosquitto systems deployed on a single Raspberry Pi.

Results in Figure 6.13 show that R-Pulsar's throughput scales as the message size increase. This experiment is representative of a traditional IoT scenario with small messages streamed at a high rate arrival. We observed that R-Pulsar pub/sub messaging system outperforms Kafka by a factor of 3x and Mosquitto by a factor of 7x. Also, Apache Kafka exhibits a high variability of throughput performance. This is explained

by the fact that Kafka continuously stores messages on the disk overwhelming the filesystem and producing this unpredictable throughput. The use of a memory-mapped queue allows R-Pulsar to obtain higher throughput, but also steadier and more predictable throughput.

In this second experiment, we want to demonstrate that R-Pulsar can be deployed on Android Phones. The experiment setup consists of an Android device as a data producer and a single Raspberry Pi as the RP. In Figure 6.14, the throughput comparison of R-Pulsar and Mosquitto [43] shows similar performance with larger messages. For smaller messages, R-Pulsar exhibits a better performance (factor of 10x). Also, Mosquitto presents a larger variability of performance (unpredictable throughput).

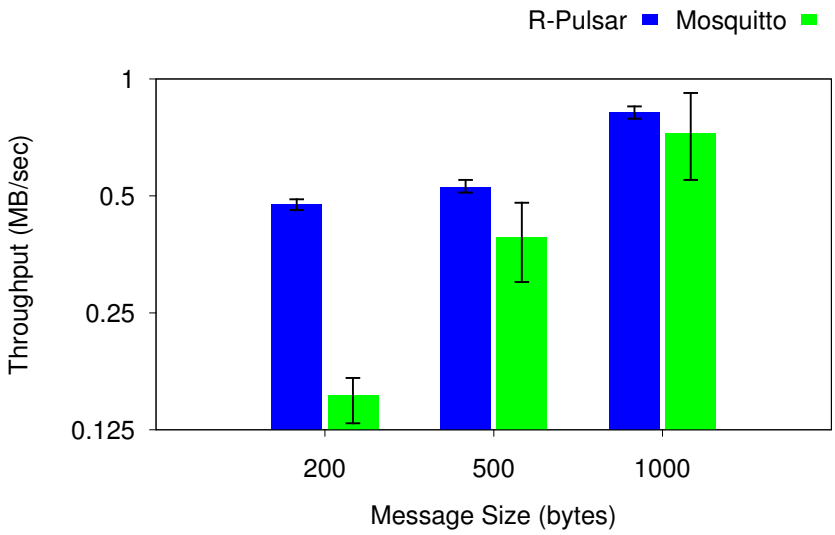

Figure 6.14: Single broker and producer throughput performance as message sizes grow. Comparison of R-Pulsar and Mosquitto on the Android system.

6.6.2 Scalability of Store/Query Operations Over Cloud Systems

These experiments are aimed to stress the system and evaluated the storage and query scalability of R-Pulsar using multiple workload sizes. The following workloads were used for the tests: Workload 1 (W1) stored/queried one element, Workload 2 (W2) stored/queried 10 different elements, Workload 3 (W3) stored/queried 50 different elements, and Workload 4 (W4) stored/queried 100 different elements. For this test, all RP nodes were part of the same P2P network and the same geographic region to

evaluate how R-Pulsar scales as the number of RP increases in each region.

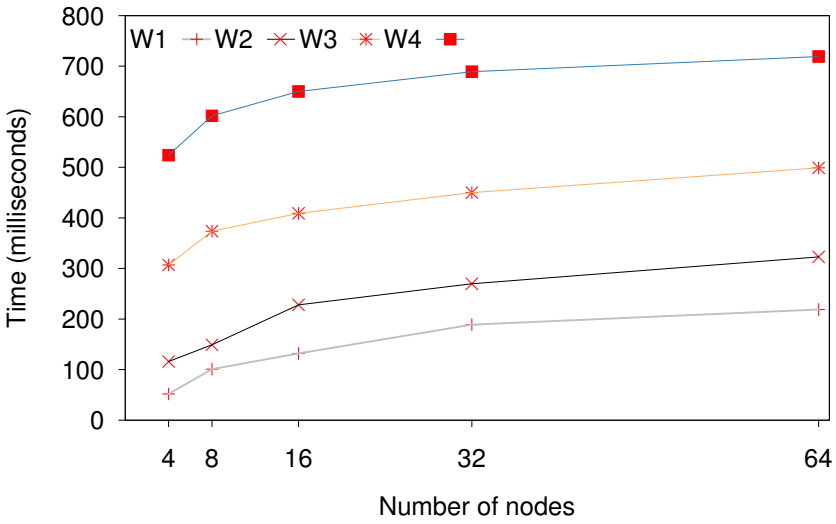

Figure 6.15: Evaluation of R-Pulsar store query operations as the number of nodes increases on Chameleon cloud.

Figure 6.15 presents the scalability evaluation of the R-Pulsar store operation. The figure shows that for storing a single element (W1), the runtime increased by a factor of ∼4 when the system size increased by a factor of 16 (from 4 nodes to 64 nodes). As the system expands, the number of intermediary nodes involved in routing the query grows, causing an increase in the runtime. The storage of 100 different elements (W4) forces the system to store elements in multiple destinations.

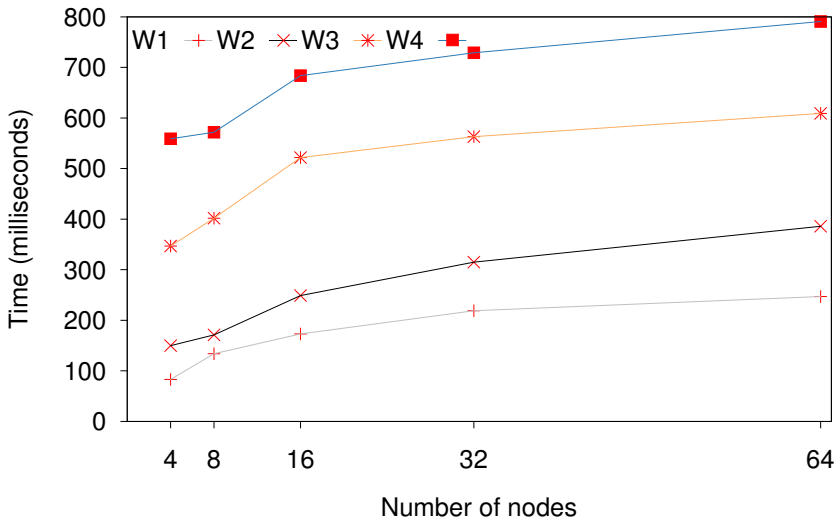

Figure 6.16: Evaluation of R-Pulsar exact query operations as the number of nodes increases on Chameleon Cloud.

Figure 6.16 presents an evaluation of the exact query operations. It shows that the query of a single element (W1), the runtime increases by a factor of 2.8 when the system size increases by a factor of 16 (from 4 nodes to 64 nodes).

6.6.3 Performance of the Query and Store Layer Over Edge Systems

The next set of experiments explores the performance of the R-Pulsar's storage and query layer as compared to self-contained, embedded and lightweight data storage systems. We compare R-Pulsar with lightweight SQL (SQLite) and non-SQL (NitriteDB) storage systems. We choose SQLite and Nitrite DB because both systems are designed to be deployed in constrained devices [126] [127]. In addition, Nitrite DB is a key-value store like R-Pulsar.

The experiments were deployed in 10 Raspberry Pis and grouped them together in the same R-Pulsar group. A client then issued requests for data to be stored and queried. As neither Nitrite DB nor SQLite supports horizontal data partitioning, the client directly queried a single DB for these systems. In the case of R-Pulsar, the content-based routing layer is responsible for determining where the data should be stored or queried.

We present three different results for these experiments. Figure 6.17 shows the time required for each system to store different sets of elements. R-Pulsar presents a steady performance as the number of elements grows. It also outperforms Nitrite DB (factor of 30x).

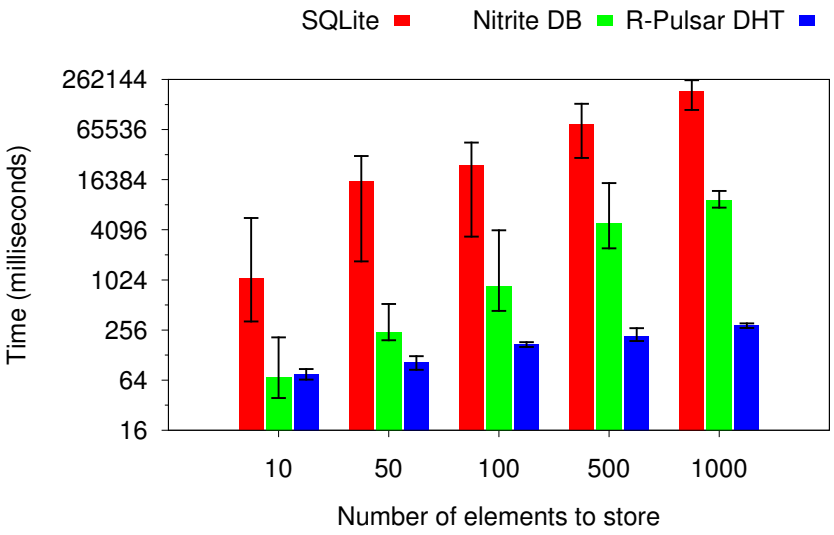

Figure 6.17: Storage performance of R-Pulsar, SQLite and Nitrite DB as the number of elements to be stored increases over 10 Raspberry Pi.

Figure 6.18 and 6.19 present the results for experiments using exact and wildcard queries (a profile containing wildcards, ranges or both). In the last two experiments, we can observe that Nitrite DB and SQLite are both faster when the number of consecutive reads is small, but R-Pulsar outperforms both when the workload increases. Overall, R-Pulsar has a higher performance because it takes advantage of the distributed storage of data over multiple RPs so that queries can be performed in parallel.

6.7 Rule-based Programming Abstraction Evaluation

In this section we performed two sets of experiments to evaluate the rule-based programming abstraction. The tests are evaluated using edge and cloud virtual nodes.

- Edge System: Consists of two different Chameleon Cloud setups: The first node is our "drone" with computational capabilities with 1 vCPU and 2 GB of memory. The second node is the "minivan" with 8 vCPU and 32 GB of memory.

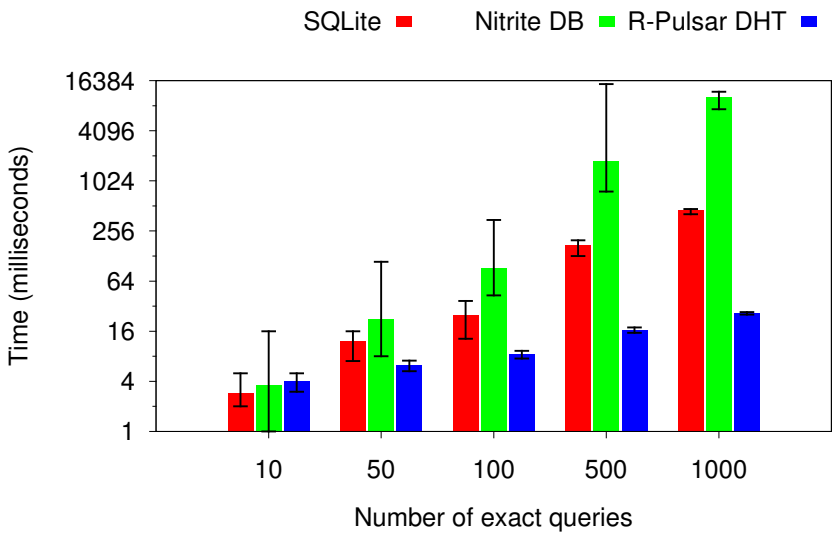

Figure 6.18: Exact query performance of R-Pulsar, SQLite and Nitrite DB as the number of exact queries increases over 10 Raspberry Pi.

- Cloud System: Consists of three Chamelon Cloud instances of type m1.large (4 CPUs and 8 GB of memory)

6.7.1 Scalability and Overhead Over Edge and Cloud Systems

To evaluate the scalability and overhead of the rule engine system we added different rule amounts and streamed our regular workload, forcing the evaluation of each of the tuples. The plotted graph in Figure 6.20 shows the scalability and overhead of our rule engine for two different machines described above. We can observe that the overhead is very minimal, we can also observe that as we add rules the overheads do not increase exponentially making it suitable to handle hundreds of rules.

In this experiment we wanted to show the importance of being able to programmatically express a trade-off between data quality and computational performance. Figure 6.21 we demonstrate that if we do not allow the to specify QoS rules to set a deadline in this case a deadline of 20 seconds per tuple was specified, depending on the workload and the underlying computing capabilities most of the tuples will not satisfy the deadline. We can also see that our algorithm can't guarantee 100% completion in time in small computational resources, but that is due to the fact that the CPU can't drain the storm bolt queues fast enough, so some tuples will experience some extra delay. The

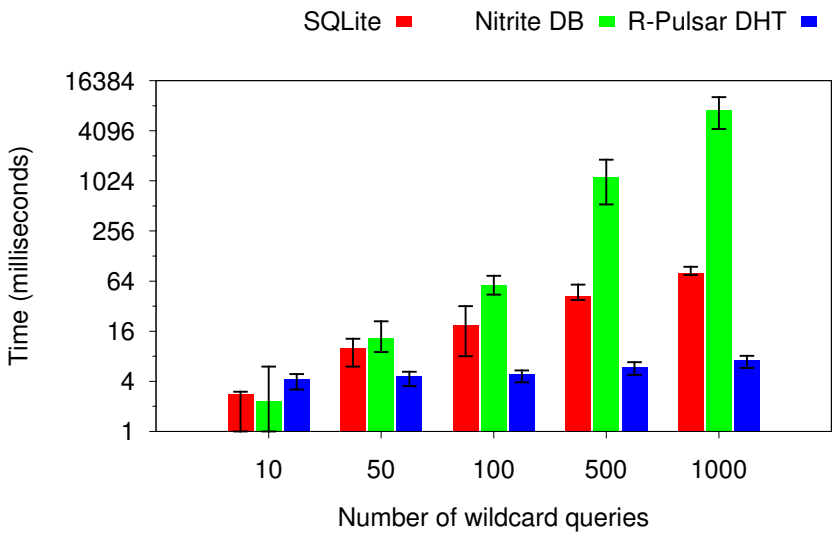

Figure 6.19: Wildcard query performance of R-Pulsar, SQLite and Nitrite DB as the number of wildcard queries increases over 10 Raspberry Pi.

reason why the graph flattens after adding 50 or more rules into the system is due to the fact that the system stops as soon as it finds a rule that is satisfied.

6.8 End To End Evaluation

In this section we implement one of the five uses cases using R-Pulsar to showcase the ability to express and decide what, where and when data gets collected and processed, using edge and cloud resources, and we compared it against a traditional approach of moving all the data to the cloud for analysis.

6.8.1 Disaster Response Use Case

For the disaster response use case we performed a set of experiments to compare the proposed split architecture (edge and core processing) with the current state of the art approach in which the stream processing is located in a fixed location at the core of the infrastructure. To perform these experiments we deployed them all in the same Chameleon cluster and we introduced artificial latency between the edge of the network and the core of the network. For the edge setup, 3 instances of type m1.small simulate computation capabilities of a drone (1 VCPUs and 2 GB of memory) and 3 other

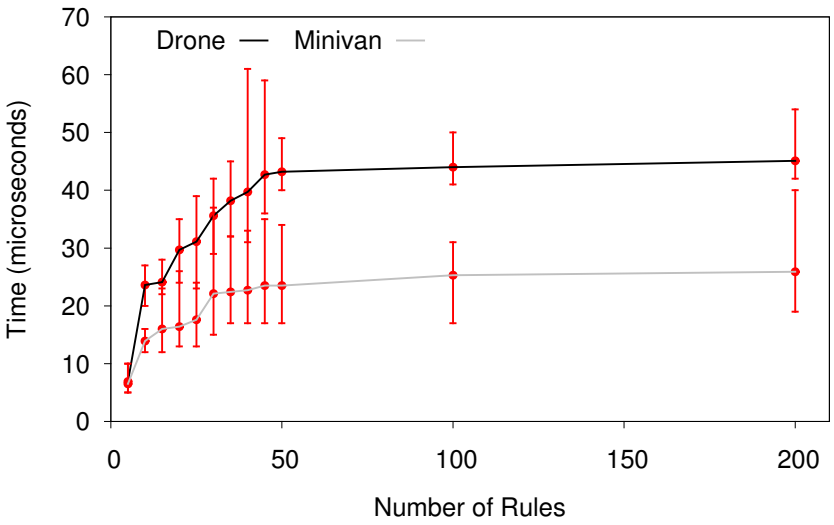

Figure 6.20: Rule engine overhead for different number of rules.

instances of type m1.medium simulate the minivan (2 VCPUs and 8 GB of memory). The core of the network is represented by 3 instances of type m1.large (4 CPUs and 8 GB of memory). To simulate that the edge infrastructure was located in a minivan or a drone we stored all the images that need to be processed locally to simulate a small latency since the minivan or the drone will be producing the data. For the traditional approach the storm topology gets the data from an external server to simulate the latencies need it to transfer between the edge and the core of the network.

For the experiment in Figure 6.22 we wanted to demonstrate that if we simply deployed our entire workflow at the edge of the network without any "quality" trade-off and we compared it to the traditional approach where each of the LiDAR images will be sent to the core for processing. We can see that with small workflows the edge is significantly faster than the core of the network, but as the number of images that need to be processed grows (the affected area is large) the core performs better than the edge due to limited resources at the edge of the network.

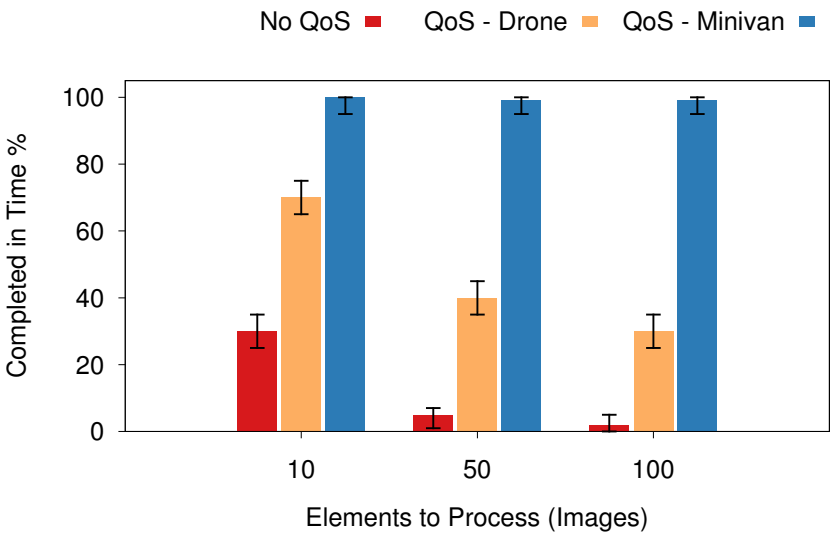

Figure 6.21: Data quality rule-based system 20 second deadline.

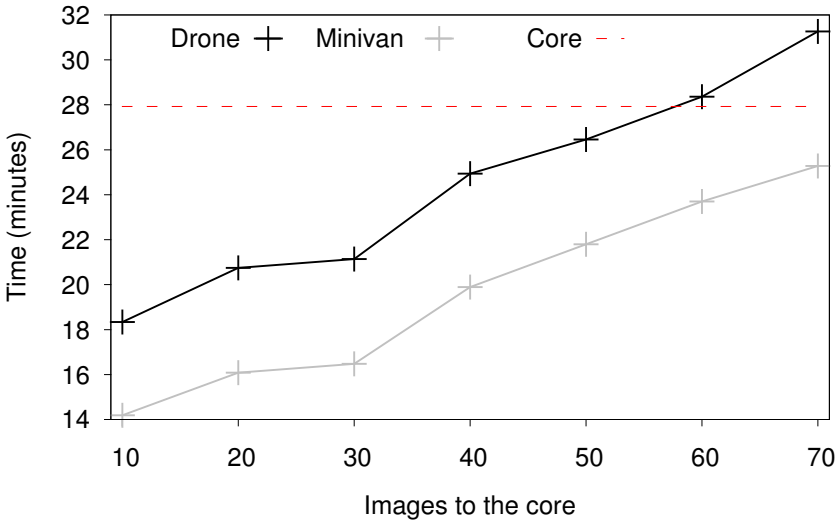

Figure 6.24: Disaster response workflow 50 images edge speed-up.

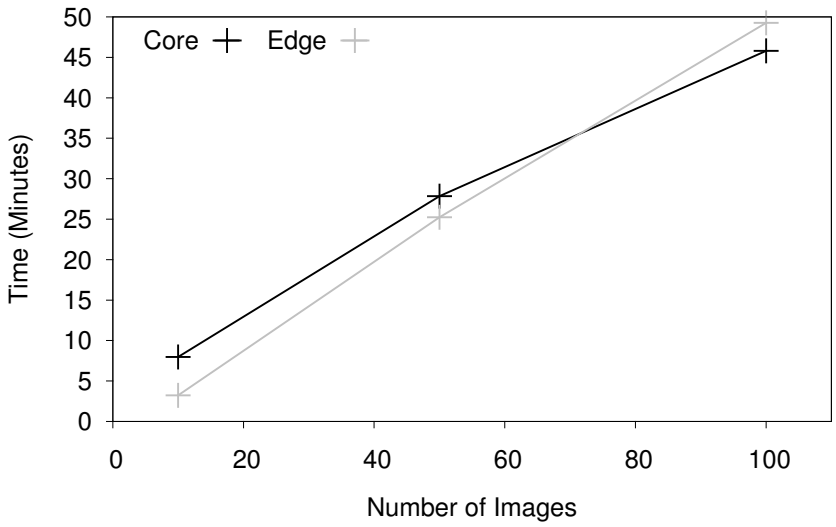

Figure 6.22: Disaster response workflow edge vs core no QoS.

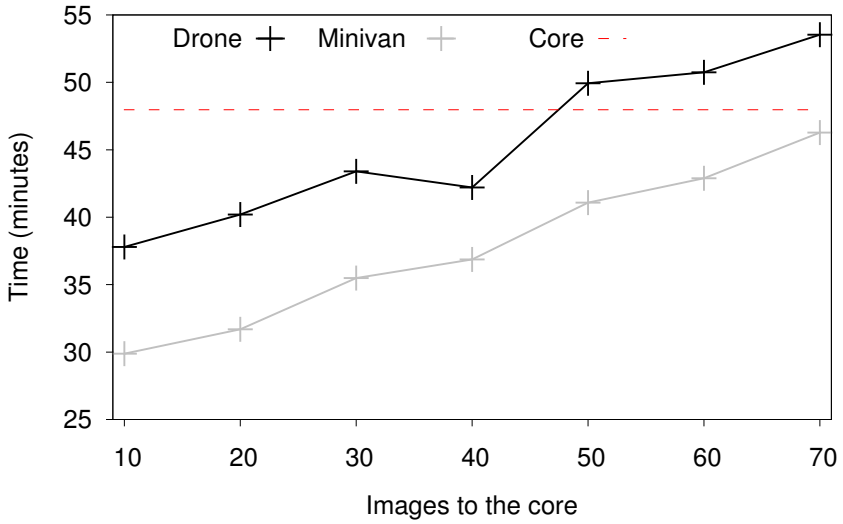

Figure 6.25: Disaster response workflow 100 images edge speed-up.

Figures 6.23, 6.24, and 6.25 showcase the speed-up that can be obtained by trading off some image "quality" for some computational complexity. Figure 6.23 is the graph for a workflow with 10 LiDAR images to be processed; one can observe that if we perform all the computations in the "drone" and only 10% of those images need further processing we get a speed up of 44% compared to sending all 10 images to the core of the network. We can observe from figures 6.24 6.25 that due to the small compute limitations, the drone gets a faster speed-up when the worflow size is small and a

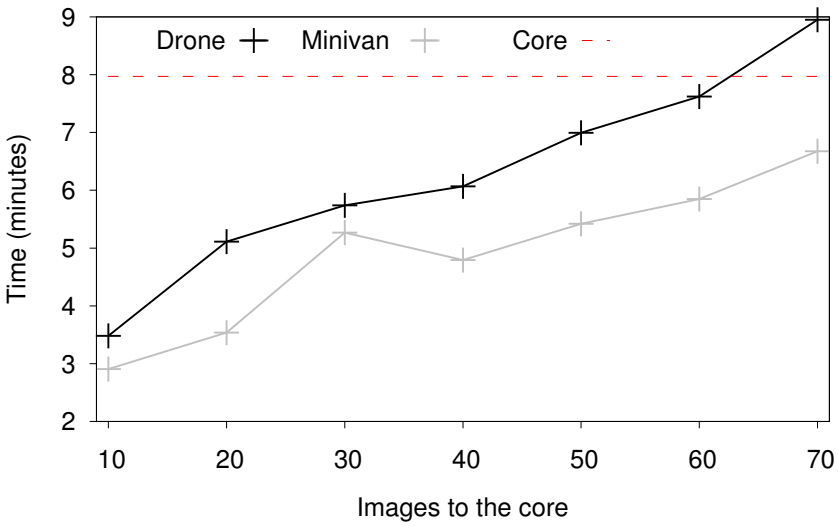

Figure 6.23: Disaster response workflow 10 images edge speed-up.

small percentage of tuples needs further processing, which makes it perfect for assessing affected areas where minivans can't get to due to road blocks. In the case of the minivan, it gets a higher speed-up in all three cases since it has higher computational resources and we can also observe that we can still send a large percentage of rules to the core and we still get a higher speed-up. In the best case the minivan is 71% faster than the traditional approach where all the images are sent to the cloud.

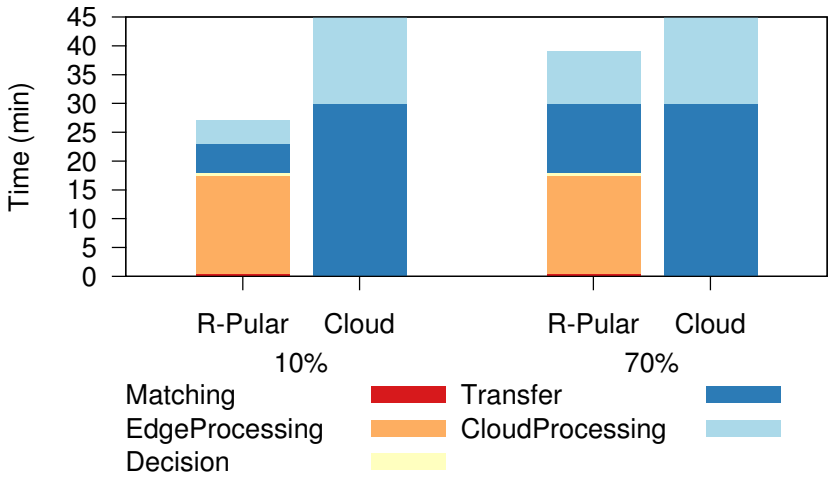

Figure 6.26: Disaster response workflow 100 images with breakdown.

For this last experiment we performed the same end-to-end evaluation of the disaster

recovery use case as Figure 6.25, and recorder the time it took to perform all the operations inside R-Pulsar to see where the time is being spend. The graph measures 5 different components of R-Pulsar, the first one is the matching of the AR profiles, the second one is the time it takes to process at the edge, the third is the time it takes to make a decision with the rule engine and the last two are the time to transfer the data from the edge to the cloud and processes the data in the cloud. Figure 6.26 consists of two experiments the first experiment assumes that only 10% of the images need to go to the cloud for post-processing, and the second graph assumes that 70% of the images need to go to the cloud for post-processing. We can observe that in the cloud only approach we are spending about 80% of the time just moving data from the edge to the cloud, and very little time computing. In the case of R-Pulsar we spend more time computing at the edge due to the limited computational resources but we cut the transfer time between the edge and the cloud by more than in half, since only a fraction of the images need post-processing.

Chapter 7

Application to the Distributed Operator Placement Problem

7.1 Introduction

The number of Internet of Things applications is forecast to exponentially grow within the coming decade. Owners of such applications strive to make predictions from large streams of complex input in near real time. The heterogeneity among the edge devices and cloud servers introduces an important challenge for deciding how to split and orchestrate the IoT applications across the edge and the cloud. In this chapter, we propose a solution on how to split IoT applications dynamically across the edge and the cloud, allowing us to improve performance metrics such as end-to-end latency (response time), bandwidth consumption, and edge-to-cloud and cloud-to-edge messaging cost.

7.2 Problem Description

We focus on three performance metrics for placing Internet of Things (IoT) applications across edge and cloud resources, *i.e.*, the end-to-end application latency [128], the WAN traffic, and the messaging cost (messages exchanged between the edge and the cloud). The IoT operator placement problem consists of defining how to accommodate the application components (*i.e.*, operators) on the available resources of the network topology to optimize one or more performance metrics.

Table 7.1 summarizes the notation used throughout the paper.

We define a computational resource (*i.e.*, cloud server or edge device) as a triple $r_k = \langle cpu_k^r, mem_k^r, f_k^r \rangle \in \mathcal{R}$, where cpu_k^r is the CPU capability in Millions of Instructions per Second (MIPS), mem_k^r is the memory capability in bytes, and $f_k^r \in \{0, 1\}$

Table 7.1: Main notation adopted for the problem description.

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

signals whether r_k is a *cloud* resource. Similarly, the network link is drawn as a triple $l_{k \leftrightarrow l} = \langle bdw_{k \leftrightarrow l}, lat_{k \leftrightarrow l}, f_{k \leftrightarrow l} \rangle \in \mathcal{L}$, where $k \leftrightarrow l$ represents the interconnection between resource k and l, $bdw_{k \leftrightarrow l}$ the bandwidth capability in bits per second (bps), $lat_{k \leftrightarrow l}$ the latency in seconds, and $f_{k \leftrightarrow l}$ signals whether the link is part of a WAN. We consider the latency of a resource k to itself (*i.e* $lat_{k \leftrightarrow k}$) to be 0.

Each operator of the IoT application is a quintuple $o_i = \langle cpu_i^o, mem_i^o, \psi_i^o, \omega_i^o, f_i \rangle \in \mathcal{O}$, where cpu_i^o is the CPU requirement in Instructions per Second (IPS) to handle an individual event, mem_i^o is the memory requirement in bytes to load the operator, ψ_i^o is the ratio of number of input events to output events (*i.e.*, selectivity), ω_i^o is the ratio of the size of input events to the size of output events (*i.e.*, data compression/expansion factor), and $f_i \in \{source, sink, transformation\}$ signals whether o_i is a *source*,

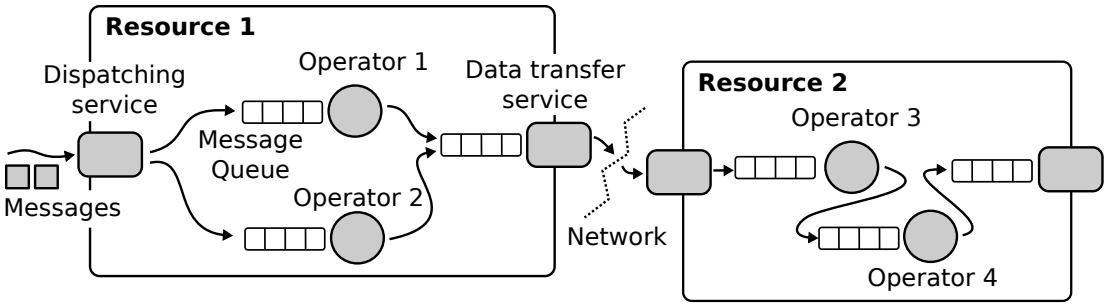

Figure 7.1: Example of four operators and their respective queues placed on two resources.

sink/output, or *transformation*. The rate at which operator i can process events at resource k is denoted by $\mu_{\langle i,k \rangle}$ and is essentially $\mu_{\langle i,k \rangle} = cpu_k^r \div cpu_i^o$. An event stream $s_{k \to l}^{\rho} \in \mathcal{S}$ connects operator k to l with a probability ρ that an output event emitted by k will flow through to l.

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

A computational resource can host one or more operators; operators within a same host communicate directly whereas inter-node communication is done via a communication service as depicted in Figure 7.1. Events are handled in a First-Come, First-Served (FCFS) fashion both by operators and the communication service that serialises messages to be sent to another host. Both operators and the communication service follow an M/M/1 model for their queues which allows for estimating the waiting and service times for computation and communication. The computation or service time $stime_{\langle o_i, r_k \rangle}$ of an operator i placed on a resource k is hence given by:

$$stime_{\langle i,k \rangle} = \frac{1}{\mu_{\langle i,k \rangle} - \lambda_i^{in}} \tag{7.1}$$

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

send a message to operator j on a resource l is:

$$ctime_{\langle i,k\rangle\langle j,l\rangle} = \frac{1}{\left(\frac{bdw_{k\leftrightarrow l}}{\varsigma_i^{out}}\right) - \lambda_j^{in}} + l_{k\leftrightarrow l} \tag{7.2}$$

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

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

$$p_i = o_0, o_1, \ldots, o_k, o_{k+1}, \ldots, o_{n-1}, o_n \tag{7.3}$$

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

$$\begin{aligned} L_{p_i} &= \sum_{\substack{o\in\mathcal{O}\\ r\in\mathcal{R}}} mo_{\langle o,r\rangle} \times stime_{\langle o,r\rangle} \\ &+ \sum_{r'\in\mathcal{R}} ms_{\langle o\to o+1,r\leftrightarrow r'\rangle} \times ctime_{\langle o,r\rangle\langle o+1,r'\rangle} \end{aligned} \tag{7.4}$$

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

$$W_{p_i} = \sum_{\substack{s_{i\to j}\in\mathcal{S}\\ k\leftrightarrow l\in\mathcal{L}}} \nVdash_{\{f_{k\leftrightarrow l}=1\}} \times ms_{\langle i\to j,k\leftrightarrow l\rangle} \times \varsigma_i^{out} \tag{7.5}$$

Likewise, the messaging cost is calculated by the number of messages that reaches the cloud from the edge and vice versa. The indicator $\nVdash_{\{f_k^r=0 \text{ and } f_{k'}^r=1\}}$ indicates that

the previous operator i is placed on edge ($f_k^r = 0$) and it sends messages to operator i' in cloud ($f_{k'}^r = 1$), the second part of the cost refers the other way cost (cloud to edge). The number of messages in p_i is given as:

$$C_{p_i} = \sum_{\substack{i\in\mathcal{O}\\ i'\in\mathcal{O}\\ k\in\mathcal{R}\\ k'\in\mathcal{R}}} \Big(\mathbb{1}_{\{f_k^r=0 \text{ and } f_{k'}^r=1\}} \times \big(mo_{\langle i',k'\rangle} \times \lambda_{i'}^{in} + mo_{\langle i,k\rangle} \times \lambda_{i'}^{out} \big) \Big) \tag{7.6}$$

The parameters latency (Par_{lat}), WAN traffic (Par_{wan}), and monetary cost (Par_{cost}) receive the current values of the running application. A single aggregate cost metric uses the parameters and *Simple Additive Weighting method* [129] (normalized in the interval [0,1]) offers a unified metric where w_l, w_w and w_c, with $w_l + w_w + w_c = 1$, are non-negative weights for the different costs. Each metric of path p_i is divided by its corresponding parameters and is then multiplied by its weight. The sum of the three metrics in the path p_i results in the aggregate cost. Formally, the *AggregateCost* in p_i is determined as:

$$\begin{aligned} AggregateCost_{p_i} = w_l \times \frac{L_{p_i}}{Par_{lat}} + \\ w_w \times \frac{W_{p_i}}{Par_{wan}} + w_c \times \frac{C_{p_i}}{Par_{cost}} \end{aligned} \tag{7.7}$$

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

$$\min \sum_{p_i\in\mathcal{P}} AggregateCost_{p_i} \tag{7.8}$$

Subject to:

$$\lambda_o^{in} < \mu_{\langle o,r\rangle} \qquad \forall o \in \mathcal{O}, \forall r \in \mathcal{R} | mo_{\langle o,r\rangle} = 1 \tag{7.9}$$

$$\lambda_o^{in} < \left(\frac{bdw_{k\leftrightarrow n}}{\varsigma_{o-1}^{out}} \right) \qquad \forall o \in \mathcal{O}, \forall k \leftrightarrow n \in \mathcal{L} | mo_{\langle o,k\rangle} = 1 \tag{7.10}$$

$$\sum_{o\in\mathcal{O}} mo_{\langle o,r\rangle} \times \lambda_o^{in} \leq cpu_r \qquad \forall r \in \mathcal{R} \tag{7.11}$$

$$\sum_{o\in\mathcal{O}} mo_{\langle o,r\rangle} \times mem_{\langle o,r\rangle} \leq mem_r \qquad \forall r \in \mathcal{R} \tag{7.12}$$

$$\sum_{\substack{s_{i\to j}\in\mathcal{S}\\ k\leftrightarrow l\in\mathcal{L}}} ms_{\langle i\to j,k\leftrightarrow l\rangle} \times \varsigma_i^{out} \leq bwd_{k\leftrightarrow l} \qquad \forall k\leftrightarrow l\in\mathcal{L} \tag{7.13}$$

$$\sum_{r\in\mathcal{R}} mo_{\langle o,r\rangle} = 1 \qquad \forall o\in\mathcal{O} \tag{7.14}$$

$$\sum_{k\leftrightarrow l\in\mathcal{L}} ms_{\langle i\to j,k\leftrightarrow l\rangle} = 1 \qquad \forall s_{i\to j}\in\mathcal{S} \tag{7.15}$$

Constraint 7.9 guarantees that a resource can provide the service rate required by its hosted operators whereas Constraint 7.10 ensures that the links are not saturated. The CPU and memory requirements of operators on each host are ensured by Constraints 7.11 and 7.12 respectively. Constraint 7.13 guarantees the data requirements of streams placed on links. Constraints 7.14 and 7.15 ensure that an operator is not placed on more than a resource and that a stream is not placed on more than a network link respectively.

7.3 R-Pulsar Framework Extension

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

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

R-Pulsar Plan Finder: This component computes the most optimized operator placement plan. It uses a three-step approach for calculating the optimal operator placement plan for deploying dataflows between the edge and the cloud. Section 7.3.2 presents the three-step operator placement strategy developed for R-Pulsar.

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

7.3.1 R-Pulsar Nodes

Each rendezvous point (RP) in the R-Pulsar P2P network can be elected as a master or as a worker. R-Pulsar differs from other master/slave clusters such as Apache Storm [47] in the sense that R-Pulsar master and worker node roles are assigned dynamically every time a dataflow is deployed.

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

Deploying a topology to the R-Pulsar P2P network involves submitting the prepackaged dataflow file along with topology configuration. Then the information will be routed to the responsible RP using the content-based interactions [130]. The content-based interactions allow users to route dataflows to unknown RPs; the RP who receives the message will be automatically elected as the master RP for that dataflow. Once the master RP has been elected, it then uses the infrastructure controller component to collect the network information of all the worker RPs. That information is then passed to the operator placement algorithm to generate a placement strategy. Once the operator placement algorithm has an efficient operator placement plan, then the master RP distributes the tasks to the worker RPs.

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

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

worker RPs assigned with tasks remain healthy.

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

7.3.2 Placement Strategy

The strategy for operator placement on R-Pulsar applies statistics collected by profiling the application and the location of sinks and sources. The operator placement aims to minimize the *AggregateCost* (Equation 7.8) by splitting the IoT application across edge and cloud by considering priorities of operators according to the infrastructure to which the sinks are assigned. The operator placement strategy comprises three phases: (i) application profiling; (ii) candidate placement and (iii) final placement.

Phase 1 – Application Profiling:. In the first phase the worker RPs and the master RPs using the infrastructure controller component to continuously collect statistics [131] from the running dataflow. The collected data includes the following information about the operators:

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

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

Phase 2 – Candidate Placement: In phase two, the user-predefined locations of

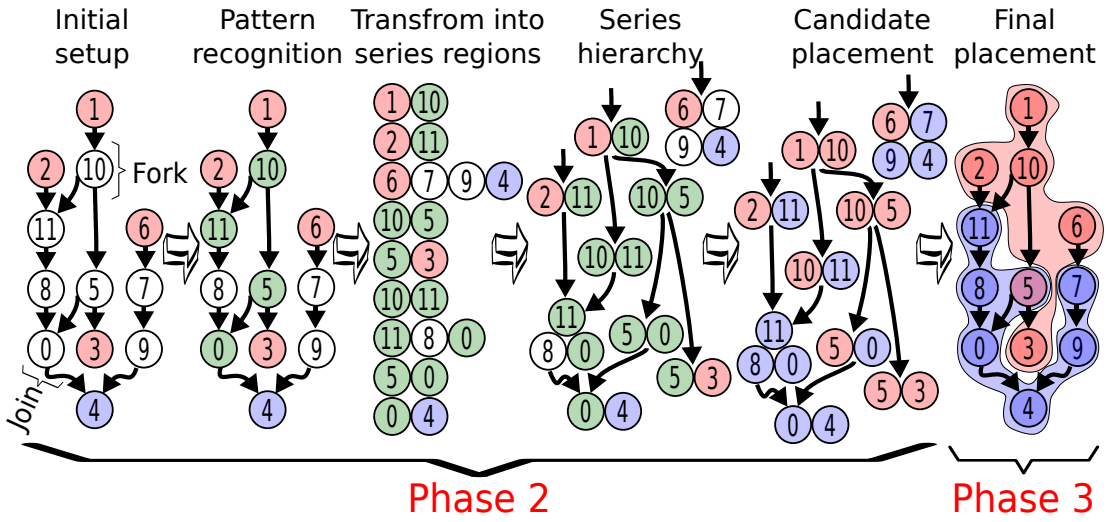

Figure 7.2: Phases to determine the final placement using split points, where red means placed on edge, blue represents placed on cloud, and green delimits forks and joins.

sinks and sources are used to identify patterns in the dataflow (Section 7.3.3). As depicted in Figure 7.2, a dataflow can comprise multiple patterns such as (i) forks, where messages can be replicated to multiple downstream operators or scheduled to downstream operators in a round-robin fashion, using message key hashes, or considering other criteria [132]; (ii) parallel regions that perform the same operations over different sets of messages or where each individual region executes a given set of operations over replicas of the incoming messages; and (iii) joins, which merge the outcome of parallel regions.

We consider that an IoT dataflow is a Series-Parallel-Decomposable Graph which either consists of a series of linearly dependent operators, or operators that can be executed independently in parallel, or a combination thereof. Phase 2 uses related techniques to identify graph regions that present these patterns [67]. This information is used to build a hierarchy of region dependencies (*i.e.* downstream and upstream relations between regions) and assist in placing operators across cloud and edge resources. The streams in the graph paths that separate the operators are hereafter called the *split points*. The rationale behind building such region hierarchy is to evaluate first the operators that can have greater impact on the overall end-to-end latency. Figure 7.2 illustrates the phases of the method to determine the split points (green circles), where red circles represent operators placed on edge resources whereas blue ones are on the

cloud: (i) The method starts with sources and sinks whose placements are predefined by the user; (ii) split points are discovered (green circles) as well as sinks that correspond to actuators that can be placed on the edge; (iii) the branches between the existing patterns (green, red, and blue circles) are transformed into series regions; (iv) a hierarchy following the dependencies between regions is created; and (v) the regions provide information to split the operators on edge candidate placement evaluating if the operator flows events to actuators.

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

Phase 3 – Final Placement: Once phase two has completed and the profiling phase has established the requirements from the different operators, an operator placement strategy is created and deployed. The strategy reduces the combinatorial space by estimating only once the computation (Equation 7.1) and communication (Equation 7.2) overheads to operators targeted to cloud (Phase 2). Otherwise, operators to edge (edge candiate placements) have their overheads estimated for all edge devices evaluating their constraints (Equation 7.9 – Equation 7.15). The strategy gives high priority to edge since cloud sinks often store messages for batch processing, whereas the edge side

Algorithm 1: Algorithm to get the candidate placement.

```
1  Function GetCandidates(G = (O, S))
2      O ← O ∪ virt_src ∪ virt_sink
3      S ← S ∪ s_{virt_src→o}, ∀o ∈ O and f_o = source
4      S ← S ∪ s_{o→virt_sink}, ∀o ∈ O and f_o = sink
5      for p ∈ GetAllPaths(G, virt_src, virt_sink) do
6          for o ∈ p do
7              temp ← temp ∪ {o}, ∀o ∉ {virt_src, virt_sink}
8              ups ← |⟨∗, o⟩ ⊂ S|, downs ← |⟨o, ∗⟩ ⊂ S|
9              if ups > 1 or downs > 1 and o ∉ {virt_src, virt_sink} then
10                 regions ← regions ∪ temp, temp ← {o}
11     Delete duplicate regions
12     for src ∈ regions do
13         for dst ∈ regions do
14             if src ≠ dst then
15                 if src[|src| − 1] = dst[0] then
16                     hierarchy ← hierarchy ∪ {src, dst}
17     for operators ∈ regions do
18         for o ∈ operators do
19             if f_o ∉ {source, sink} then
20                 for sink ∈ GetSinks(o) do
21                     if GetLocation(sink) = edge then
22                         candidate = candidate ∪ o
23                         Break
24     return candidate
```

hosts actuators. If edge devices cannot meet all operator requirements then the operator is moved to the cloud, hence, the cloud hosts its operator candidates and those that do not meet the constraints on edge. For instance, Operator 5 in Figure 7.2 was reallocated since the edge does not respect the resource constraints. Along with the overhead estimations, the strategy greedly uses the edge candiate placements for sequentially estimating the *AggregateCost* (Equation 7.7) and at each iteration, it picks the device with the minimal value (Equation 7.8) to assign the operator.

7.3.3 R-Pulsar API

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

Listing 7.1 is for specifying the operator constraints. In our case some of the operators need to be placed at the cloud and some others need to be placed at the edge. Note that if the wildcard or no placement is specified R-Pulsar will automatically decide the best placement for the operator. CloudTableInsert, MQTTPublish, and BloomFilterTask are tasks used in the ETL dataflow.

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

Listing 7.1: User specified operator physical placements constraints.

Listing 7.2 is for specifying the optimizations to apply to the dataflow. The R-Pulsar operator placement algorithm offers three optimizations: minimize end-to-end latency, bandwidth, or messaging cost. Each of the functions requires a weight normalized in the interval [0,1]; the sum of all three weights must be one. By doing so, users have the ability to optimize the latency, data transfer rate and messaging cost at the same time.

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

Listing 7.2: User specified dataflow optimizations (latency, data transfer rate and cost).

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

7.4 Evaluation

This section presents an experimental evaluation of our system. First, we present the setup and the other approaches in which the experiments will be evaluated and

compared against. Second, we present an evaluation of our system based on latency, data transfer rate, and messaging cost.

7.4.1 Setup

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

- We used an experimental edge testbed developed by the authors, inspired by Hu *et. al.* [133] that consists of 13 Raspberry Pis; 5 Raspberry Pis model 3 (4x ARM Cortex-A53 1.2GHz, 1GB of RAM and 10/100 Ethernet), and 8 Raspberry Pis model 2 (4x ARM Cortex-A7 900MHz, 1GB of RAM and 100 Ethernet).
- For the cloud we used the Chameleon cloud [120] with 5 instances of type m1.medium (2 CPU and 4 GB RAM).

The 13 Raspberry Pis are connected to the same LAN. The Raspberry Pis use the external WAN [134] (the Internet) for connecting to cloud. The LAN has a latency 0.523 ms and a bandwidth of 15 Mbits/sec. The WAN has latency 66.75 ms, and bandwidth of 87.0 Mbits/sec.

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

- **Cloud:** deploys all operators in the cloud, apart from operators provided in the initial placement.
- **LB (Taneja et. al. [68]):** iterate a vector containing the application operators, gets the middle host of the computational vector, and evaluates CPU, memory, and bandwidth constraints to obtain the operator placement.
- **Random:** simulates the user trying to guess the best placement for the dataflow between the edge and the cloud. Random is the average of 15 different dataflow deployments between the edge and the cloud resources.

All the tests are evaluated using the ETL dataflow. The ETL dataflow is an implementation of the ETL RIoTBench topology, it consists of: a single data source outputting

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

7.4.2 End-to-end Tuple Latency Evaluation

The end-to-end tuple latency corresponds to the sum of the mean times from the two paths in ETL dataflow (cloud and edge). The conducted experiment evaluates the end-to-end tuple latency using Equation 7.7 where w_l is equal to 1, and w_w and w_c are equal to 0. The experiment aims to evaluate how efficient the cloud, Random, and LB approaches are at minimizing the end-to-end tuple latency and compare the R-Pulsar operator placement approach. In addition, three failures were manually injected to showcase the dynamicity and flexibility to recover from node failures. The first failure makes 38% of the edge cluster unavailable (100 ms). The second failure affects the remaining 62% of the nodes (300 ms). Before the 62% of the nodes fail, the 38% of the nodes are back online. The third and last failure affects 50% of the cloud instances (505 ms).

Figure 7.3 shows that on average tuples are computed 31% faster when compared to the traditional cloud setup, and 38% faster than Random and the LB placement approaches. The reason why the Random failures recover much faster than LB when compared to R-Pulsar is because Random is the average of multiple different deployments and in some cases the first failure is not affected. Figure 7.3 demonstrates that R-Pulsar operator placement strategy is capable of splitting the dataflow efficiently between the edge and the cloud and reduce the end-to-end tuple latency.

The second experiment aims to evaluate how efficient the Cloud, Random, and LB

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

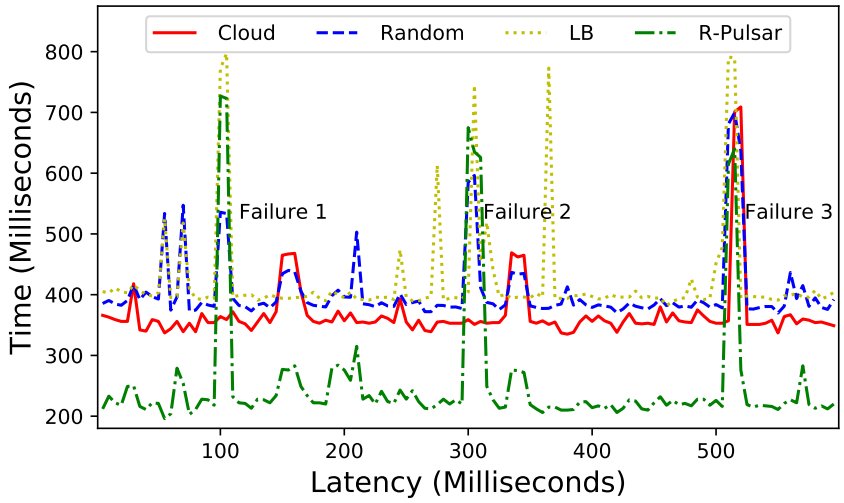

Figure 7.3: End-to-end tuple latency optimization with 3 self injected failures affecting edge and cloud nodes, while comparing it with Cloud, Random and LB approaches.

approaches are at minimizing end-to-end tuple latency and compare it with R-Pulsar approach. In this experiment no failures were injected.

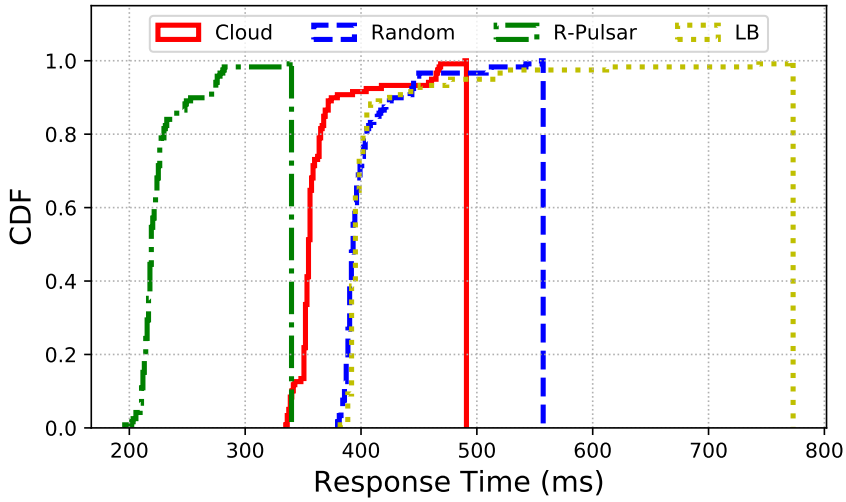

Figure 7.4: End-to-end tuple latency optimization cumulative distribution function (CDF) comparison with Cloud, Random and LB approaches.

Figure 7.4 shows that when R-Pulsar operator placement approach is used 80% of the tuples see a reduction in the end-to-end tuple latency by 44% compared to the LB and Random approaches and 38% compared to the cloud approach.

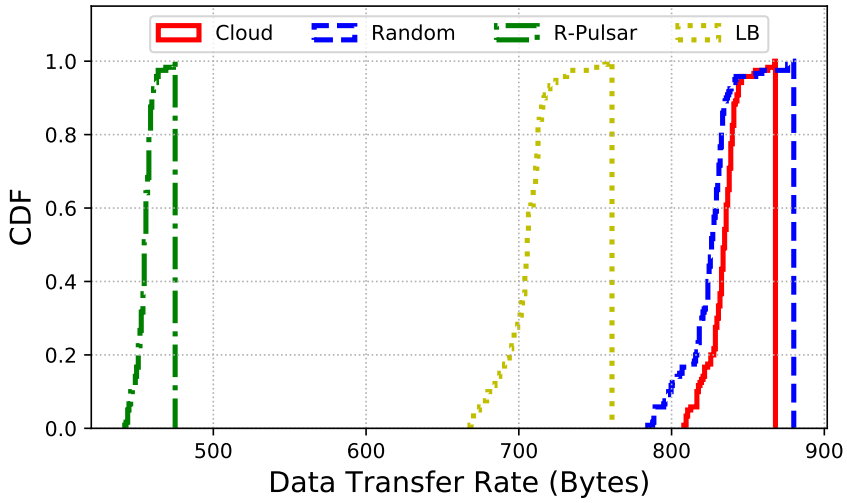

Figure 7.5: End-to-end data transfer rate optimization cumulative distribution function (CDF) comparison with cloud, Random and LB approaches.

7.4.3 Data Transfer Rate Evaluation

The Data transfer rate consists of the sum of all message sizes that traverse a WAN link per second. The values for Equation 7.7 are w_w equal to 1, and w_l and w_c equal to 0. This third experiment aims to evaluate how efficient are the cloud, Random, and LB approaches at minimizing the transfer rate between the edge and the cloud and compare the results with R-Pulsar operator placement approach. Minimizing the transfer rate between the edge and the cloud is a critical point in order to achieve timely analytics.

Figure 7.5 shows that 80% of the time R-Pulsar reduces the transfer rate between the edge and the cloud on average 35% when compared to the LB approach. And it reduces the data transfer rate by 45% when compared to the cloud and Random approaches.

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

Figure 7.6 shows that the R-Pulsar operator placement approach can also optimize the data transfer rate and the end-to-end latency by 46% and 38% respectively when compared to the cloud, 36% and 45% respectively when compared to the LB, 38% and

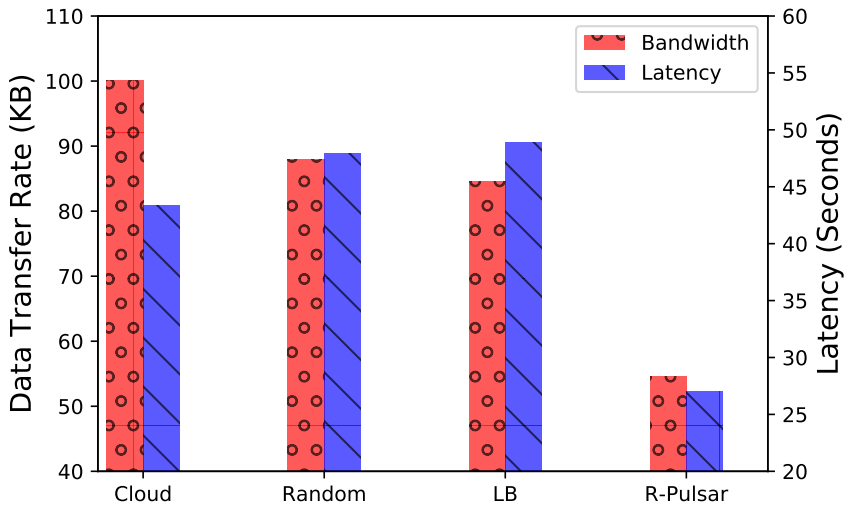

Figure 7.6: Multi optimization evaluation, end-to-end tuple latency and data transfer rate comparison with cloud, Random, and LB approaches.

44% respectively when compared to the Random approach.

7.4.4 Messaging Cost Evaluation

This two experiments aim to calculate the messaging cost of running the dataflow for a month using the cost models of two major actors, AWS and Microsoft, in a real life edge and cloud scenario. For this reason, we setup Equation 7.7 with w_c equal to 1, and w_l and w_l equal to 0. The goal of this optimization is to reduce the number of messages that reach the cloud servers.

Table 7.2: Azure IoT Hub and Amazon IoT Core messaging pricing.

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

Table 7.2 depicts two IoT cost models. The first cost model is the Microsoft Azure IoT Hub [135]. Each tier enables a maximum number of messages exchanged between

the Azure IoT Edge and the Azure IoT Hub and vice versa per day. T1 allows up to 400,000 messages a day, T2 allows up to 6,000,000 messages a day, and T3 allows up to 300,000,000 messages a day.

The second cost model is the Amazon IoT Core [136] where messaging is metered by the number of messages transmitted between your devices and AWS IoT Core and vice versa per day. Amazon offers multiple costs for different regions, for this experiment we choose the cheapest region (N.Virginia) which charges $1 per million messages sent, and the cost per message decreases after the first 1 billion messages per day.

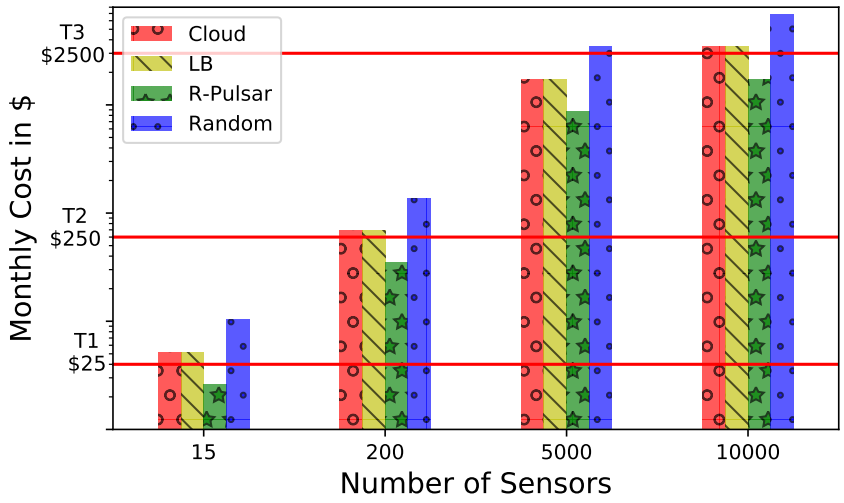

Figure 7.7: Messaging cost savings evaluation based on the Microsoft Azure IoT Hub pricing model, for four different setups.

Figure 7.7 depicts the cost of deploying the ETL dataflow using the Microsoft cost model using the four different approaches presented earlier. When using a small setup (15 sensors), the monthly cost for our system will be $25 a month while the cloud, LB, or Random approaches will cost $250 a month, savings of 90%. A similar behavior happens with a medium (200 sensors) and extra large (10000 sensors) setups.

Figure 7.8 depicts the cost of deploying the ETL dataflow using the Amazon cost model. Our system obtains a 50% cost reduction when compared to the cloud and LB approaches in all four different setups (15, 200, 5000 and 10000 sensors). In addition our system obtains a 97% savings when compared to the Random approach in all four different setups.

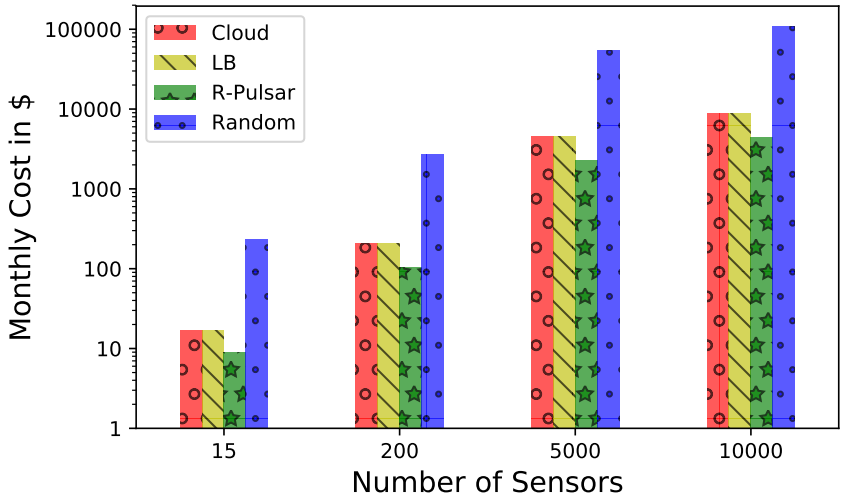

Figure 7.8: Messaging cost savings evaluation based on the Amazon IoT pricing model, for four different setups.

7.4.5 Model Evaluation

This last three experiments aim to evaluate the scalability, overhead and validate the operator placement algorithm.

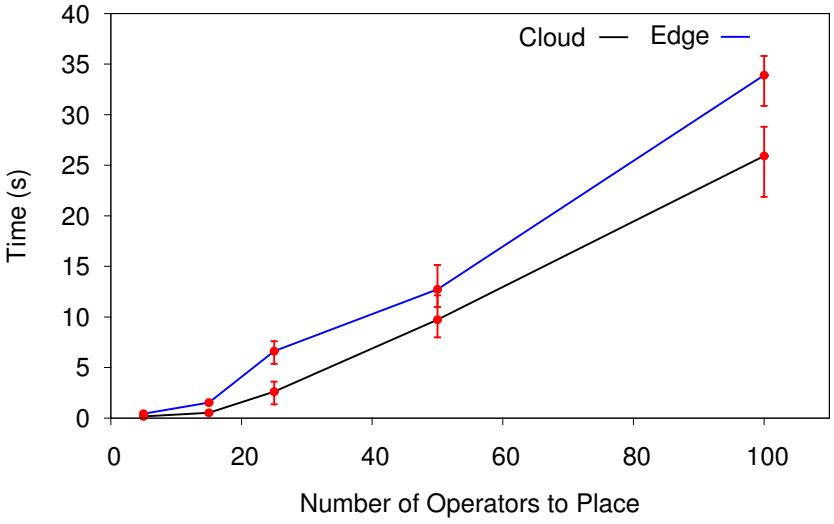

Figure 7.9: Evaluation of the scalability of the operator placement problem algorithm as the number of operators to place increase over edge and cloud resources.

Figure 7.9 depicts the scalability and the overhead of the initial operator placement problem algorithm in both edge and cloud resources. We can observe that the algorithm scales with the number of operators to place and a valid solution can be found in less

than half a minute. Not that this step is only performed when the entire workflow needs to be placed, once the operators are placed, only a portion of the operators are moved.

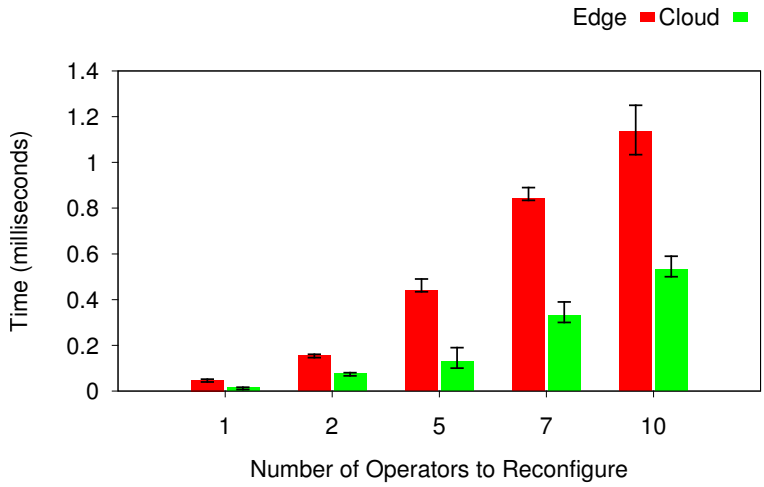

Figure 7.10: Evaluation of the cost of redeploying a subset of operators over edge and cloud resources.

Figure 7.10 depicts the redeployment scalability and overhead of the algorithm. We can observer that since we do not need to redeploy all the operators when the workflow is not meeting the constraints, we can see that the cost of redeploying a subset of operators is very low and more important decisions can be done in real time since it the redeployment calculations are small.

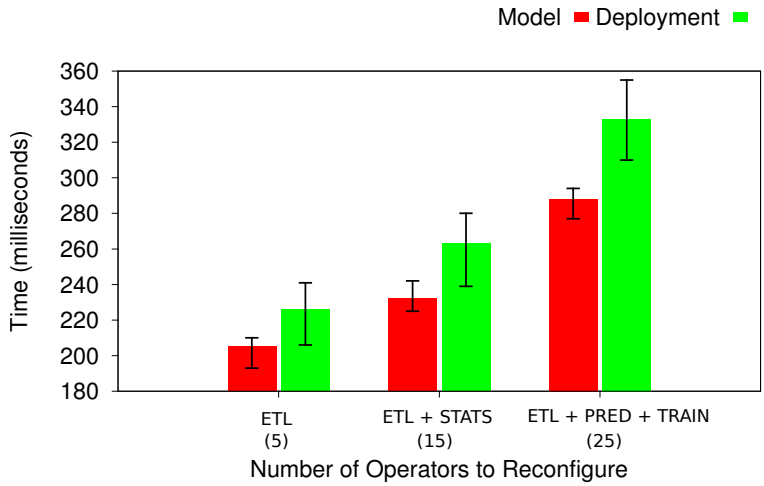

Figure 7.11: Evaluation of the actual deployment vs the modeled time as the number of operators increases.

For the final evaluation we wanted to validate the model by comparing the expected results and the actual results after deploying the workflow in a real setup, Figure 7.11 depicts the results. We can observe that when the number of operators is small the difference between the expected and the actual results only differs by at most 10%, when the number of operators grows to 25 the difference slightly grows up to 15%.

Chapter 8

Conclusion

8.1 Summary

This thesis identified and addressed key problems and requirements of IoT applications. Specifically, this thesis presented a programming abstraction enables to address the what, where, and when data needs to be processed by specifying content and action descriptors. In addition this thesis also tackles the how to split a given dataflow and place operators across edge and cloud resources. First, presented a modification of the Associative Rendezvous programming abstraction to allow to decide what and where data needs to be processed. Second presented a R-Pulsar an IoT Edge Framework that extends cloud capabilities to edge devices, enabling users to collect and analyze data closer to the source of the information. Third presented a rule-based programming abstraction for specifying when to trigger data-processing tasks based on data observations. Finally, presented a solution to the distributed operator placement problem for allowing to decide how to split the application operators between the edge and the cloud, by specifying a set of constraints. We evaluated the effectiveness, scalability, performance and overheads of R-Pulsar by using three sets of IoT applications and validated every layer by performing scalability, overhead and performance tests.

8.2 Contributions

To recapitulate, the primary contributions presented in this dissertation are as follows.

- In chapter 4 presented a content- and location-based programming abstraction for specifying **what** data gets collected and **where** the data gets analyzed.
- In chapter 5 presented a rule-based programming abstraction for specifying **when**

to trigger data-processing tasks based on data observations.

- In chapter 7 presented a programming abstraction for specifying **how** to split a given dataflow and place operators across edge and cloud resources.
- Also in chapter 7An operator placement strategy that aims to minimize an aggregate cost which covers the end-to-end latency (time for an event to traverse the entire dataflow), the data transfer rate (amount of data transferred between the edge and the cloud) and the messaging cost (number of messages transferred between edge and the cloud).
- In chapter 6 presented a performance optimizations on the data-processing pipeline in order to achieve high performance on constrained devices.
- Also in chapter 6 presented an implementation of the above capabilities as part of the R-Pulsar architecture and its evaluation using embedded devices (Raspberry Pi and Android phone).

8.3 Perspectives

The research presented in this dissertation opens several research problems that need to be addressed in order to further advance on the edge computing area.

Energy Management: A study published in 2017 determined that due to the large number of IoT devices connected to the internet in 2025 they will consume 20% of all the worldwide electricity consumption [137]. For those reasons there is a need to implement energy management policies. Energy management needs to be incorporated in the service layer in order to be able to schedule computations based on the energy consumption. A large amount of research exists focused on modeling and optimizing the energy consumption in the Cloud, but there is limited research targeting edge computing. R-Pulsar does not have the ability to quantify the amount of energy spend or to schedule computations while being energy efficient. There is a need for tools that will give feedback on how energy efficient the code is. For those reasons energy management

is a potential research direction.

Security and Privacy: IoT data differentiates itself from any other type of data due that is mostly built upon personal and highly sensitive data. For those reasons security is an important research topic. There is a need for algorithms that provide strong security guarantees, while still being suitable for constrained environments. R-Pulsar does not address any security or privacy cancers, so a possible research direction is to create algorithms that provide strong security protection, while fitting within an acceptable footprint. It needs to be lightweight enough that will still leave room for the embedded OS and applications code.

Edge based stream processing engines: There is a need to develop more lightweight stream processing engine that can be deployed on constrained devices. Current stream processing engines (SPEs) such as Storm [47], Flink [48], Heron [46] and Spark [50] where designed to be deployed in the Cloud, with large number of clusters with powerful computing resources and plenty of memory. However, these assumptions do not hold at the Edge of the network. R-Pulsar does not offer a new SPE, it just simply uses Apache Edgent. For this reason there is a need to research a develop new SPE that are designed to run in constrained devices.

References

[1] "Cisco Internet of Things At-a-Glance - https://bit.ly/38Xdd5n," 2016.

[2] D. McAuley, R. Mortier, and J. Goulding, "The dataware manifesto," in *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*, pp. 1–6, 2011.

[3] L. F. Bittencourt, R. Immich, R. Sakellariou, N. L. S. da Fonseca, E. R. M. Madeira, M. Curado, L. Villas, L. da Silva, C. Lee, and O. Rana, "The internet of things, fog and cloud continuum: Integration and challenges," *CoRR*, vol. abs/1809.09972, 2018.

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

[5] A. V. Dastjerdi and R. Buyya, "Fog computing: Helping the internet of things realize its potential," *Computer*, vol. 49, no. 8, pp. 112–116, 2016.

[6] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in *Big data and internet of things: A roadmap for smart environments*, pp. 169–186, Springer, 2014.

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

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

[9] E. Ahmed, I. Yaqoob, I. A. T. Hashem, I. Khan, A. I. A. Ahmed, M. Imran, and A. V. Vasilakos, "The role of big data analytics in internet of things," *Computer Networks*, vol. 129, pp. 459–471, 2017.

[10] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 24–31, 2015.

[11] J. Gong, C. Feeley, H. Tang, G. Olmschenk, V. Nair, Z. Zhou, Y. Yu, K. Yamamoto, and Z. Zhu, "Building smart transportation hubs with internet of things to improve services to people with special needs," tech. rep., Department of Civil Engineering, Rutgers University, 2016.

[12] A. R. Zamani, M. AbdelBaky, D. Balouek-Thomert, I. Rodero, and M. Parashar, "Supporting data-driven workflows enabled by large scale observatories," in *2017 IEEE 13th International Conference on e-Science (e-Science)*, pp. 592–595, Oct 2017.

[13] M. Ali, A. Anjum, M. U. Yaseen, A. R. Zamani, D. Balouek-Thomert, O. Rana, and M. Parashar, "Edge enhanced deep learning system for large-scale video stream analytics," in *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pp. 1–10, May 2018.

[14] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, pp. 637–646, Oct 2016.

[15] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

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

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

[18] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," tech. rep., 2009.

[19] M. Bahrami, M. Singhal, and Z. Zhuang, "A cloud-based web crawler architecture," in *2015 18th International Conference on Intelligence in Next Generation Networks*, pp. 216–223, Feb 2015.

[20] W. Tan, M. B. Blake, I. Saleh, and S. Dustdar, "Social-network-sourced big data analytics," *IEEE Internet Computing*, vol. 17, pp. 62–69, Sep. 2013.

[21] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, Z. Zhang, M. Varvello, and M. Steiner, "Measurement study of netflix, hulu, and a tale of three cdns," *IEEE/ACM Transactions on Networking*, vol. 23, pp. 1984–1997, Dec 2015.

[22] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge computing for the internet of things: A case study," *IEEE Internet of Things Journal*, vol. 5, pp. 1275–1284, April 2018.

[23] O. Salman, I. Elhajj, A. Kayssi, and A. Chehab, "Edge computing enabling the internet of things," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 603–608, Dec 2015.

[24] S. Basudan, X. Lin, and K. Sankaranarayanan, "A privacy-preserving vehicular crowdsensing-based road surface condition monitoring system using fog computing," *IEEE Internet of Things Journal*, vol. 4, pp. 772–782, June 2017.

[25] Toyota, "DENSO Corporation - https://global.toyota/en/detail/18135029," 2017.

[26] "Cisco fog computing - https://bit.ly/35h5o8c," 2015.

[27] Postscapes, "Edge middelwares - https://www.postscapes.com/internet-of-things-platforms/," 2019.

[28] "Particle cloud - https://www.particle.io," 2017.

[29] "Salesforce IoT Cloud - https://www.salesforce.com/products/salesforce-iot/overview/," 2003.

[30] "Ifttt - https://ifttt.com/discover," 2010.

[31] F. Paganelli and D. Parlanti, "A dht-based discovery service for the internet of things," *Journal of Computer Networks and Communications*, vol. Vol. 2012, Article ID 107041, 10 2012.

[32] M. Liu, T. Leppänen, E. Harjula, Z. Ou, M. Ylianttila, and T. Ojala, "Distributed resource discovery in the machine-to-machine applications," in *2013 IEEE 10th International Conference on Mobile Ad-Hoc and Sensor Systems*, pp. 411–412, Oct 2013.

[33] S. Cirani, L. Davoli, G. Ferrari, R. Léone, P. Medagliani, M. Picone, and L. Veltri, "A scalable and self-configuring architecture for service discovery in the internet of things," *IEEE Internet of Things Journal*, vol. 1, pp. 508–521, Oct 2014.

[34] A. J. Jara, P. Lopez, D. Fernandez, J. F. Castillo, M. A. Zamora, and A. F. Skarmeta, "Mobile digcovery: A global service discovery for the internet of things," in *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pp. 1325–1330, March 2013.

[35] M. Zhou and Y. Ma, "A web service discovery computational method for iot system," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, vol. 03, pp. 1009–1012, Oct 2012.

[36] G. Tanganelli, C. Vallati, and E. Mingozzi, "Edge-centric distributed discovery and access in the internet of things," *IEEE Internet of Things Journal*, vol. 5, pp. 425–438, Feb 2018.

[37] J. Mäenpää, J. J. Bolonio, and S. Loreto, "Using reload and coap for wide area sensor and actuator networking," *EURASIP Journal on Wireless Communications and Networking*, vol. 2012, p. 121, Mar 2012.

[38] W. Zhang, Y. Hu, Y. Zhang, and D. Raychaudhuri, "Segue: Quality of service aware edge cloud service migration," in *Proceedings - 8th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2016*, (United States), pp. 344–351, IEEE Computer Society, 1 2017.

[39] L. Chaufournier@ARTICLE6596496, author=W. Tan and M. B. Blake and I. Saleh and S. Dustdar, journal=IEEE Internet Computing, title=Social-Network-Sourced Big Data Analytics, year=2013, volume=17, number=5,

pages=62-69, keywords=cloud computing;data analysis;personal information systems;social networking (online);Social Network-Sourced Big Data Analytics;very large datasets;big data processing;personal ad hoc clouds;Social network services;Information management;Data handling;Data storage systems;Data models;Computers;Internet;big data analytics;cloud computing;crowdsourcing, doi=10.1109/MIC.2013.100, ISSN=1089-7801, month=Sep.,, P. Sharma, F. Le, E. Nahum, P. Shenoy, and D. Towsley, "Fast transparent virtual machine migration in distributed edge clouds," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, (New York, NY, USA), pp. 10:1–10:13, ACM, 2017.

[40] I. Farris, T. Taleb, M. Bagaa, and H. Flick, "Optimizing service replication for mobile delay-sensitive applications in 5g edge network," in *2017 IEEE International Conference on Communications (ICC)*, pp. 1–6, May 2017.

[41] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7, Oct 2017.

[42] J. Kreps, L. Corp, N. Narkhede, J. Rao, and L. Corp, "Kafka: a distributed messaging system for log processing. netdb'11," 2011.

[43] Eclipse, "Mosquitto message broker - https://mosquitto.org/," 2013.

[44] "Rabbitmq - https://www.rabbitmq.com," 2008.

[45] "Activemq - https://activemq.apache.org/," 2009.

[46] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *ACM SIGMOD Intl. Conf. on Management of Data*, pp. 239–250, 2015.

[47] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pp. 147–156, 2014.

[48] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Data Engineering*, p. 28, 2015.

[49] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endow.*, vol. 6, pp. 1033–1044, Aug. 2013.

[50] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *USENIX*, 2012.

[51] "Apache edgent - http://edgent.apache.org/," 2016.

[52] F. Pisani, J. R. Brunetta, V. M. d. Rosario, and E. Borin, "Beyond the fog: Bringing cross-platform code execution to constrained iot devices," in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 17–24, Oct 2017.

[53] H. Gupta, Z. Xu, and U. Ramachandran, "Datafog: Towards a holistic data management platform for the iot age at the network edge," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, (Boston, MA), USENIX Association, 2018.

[54] H. Gupta and U. Ramachandran, "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, DEBS '18, (New York, NY, USA), pp. 148–159, ACM, 2018.

[55] R. Mayer, H. Gupta, E. Saurez, and U. Ramachandran, "Fogstore: Toward a distributed data store for fog computing," *2017 IEEE Fog World Congress (FWC)*, pp. 1–6, 2017.

[56] M. Y. Jung and J. W. Jang, "Data management and searching system and method to provide increased security for iot platform," in *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 873–878, Oct 2017.

[57] T. Li, Y. Liu, Y. Tian, S. Shen, and W. Mao, "A storage solution for massive iot data based on nosql," in *2012 IEEE International Conference on Green Computing and Communications*, pp. 50–57, Nov 2012.

[58] E. G. Renart, D. Balouek-Thomert, and M. Parashar, "An edge-based framework for enabling data-driven pipelines for iot systems," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 885–894, May 2019.

[59] Scalagent, "Benchmark of mqtt servers - https://bit.ly/36wgjok," 2015.

[60] "smartsantander," 2007.

[61] C. Y. Leong, A. R. Ramli, and T. Perumal, "A rule-based framework for heterogeneous subsystems management in smart home environment," *IEEE Transactions on Consumer Electronics*, vol. 55, pp. 1208–1213, August 2009.

[62] L. Mainetti, V. Mighali, L. Patrono, and P. Rametta, "A novel rule-based semantic architecture for iot building automation systems," in *2015 23rd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 124–131, Sep. 2015.

[63] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, (New York, NY, USA), pp. 15–20, ACM, 2013.

[64] L. Riliskis, J. Hong, and P. Levis, "Ravel: Programming IoT Applications as Distributed Models, Views, and Controllers," in *Proceedings of the 2015 International Workshop on Internet of Things towards Applications (IoT-App'15)*, November 2015.

[65] M. Etemadi, W. McGrath, S. Roy, and B. Hartmann, "Fabryq: Using phones as smart proxies to control wearable devices from the web," Tech. Rep. UCB/EECS-2014-134, EECS Department, University of California, Berkeley, Jun 2014.

[66] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "Fogflow: Easy programming of iot services over cloud and edges for smart cities," *IEEE Internet of Things Journal*, vol. 5, pp. 696–707, April 2018.

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

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

[69] T. Elgamal, A. Sandur, P. Nguyen, K. Nahrstedt, and G. Agha, "Droplet: Distributed operator placement for iot applications spanning edge and cloud resources," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 1–8, July 2018.

[70] R. Ghosh and Y. Simmhan, "Distributed scheduling of event analytics across edge and cloud," *ACM Trans. Cyber-Phys. Syst.*, vol. 2, pp. 24:1–24:28, July 2018.

[71] E. Gibert Renart, A. Da Silva Veith, D. Balouek-Thomert, M. D. De Assunçã, L. Lefèvre, and M. Parashar, "Distributed operator placement for iot data analytics across edge and cloud resources," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 459–468, May 2019.

[72] R. Lu, K. Heung, A. H. Lashkari, and A. A. Ghorbani, "A lightweight privacy-preserving data aggregation scheme for fog computing-enhanced iot," *IEEE Access*, vol. 5, pp. 3302–3312, 2017.

[73] E. Shi, T.-H. Hubert Chan, E. G. Rieffel, R. Chow, and D. Song, "Privacy-preserving aggregation of time-series data," vol. 2, 01 2011.

[74] R. Behrens and A. Ahmed, "Internet of things: An end-to-end security layer," in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, pp. 146–149, March 2017.

[75] B. Mukherjee, R. L. Neupane, and P. Calyam, "End-to-end iot security middleware for cloud-fog communication," in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pp. 151–156, June 2017.

[76] T. Kothmayr, C. Schmitt, W. Hu, M. Brünig, and G. Carle, "A dtls based end-to-end security architecture for the internet of things with two-way authentication," in *37th Annual IEEE Conference on Local Computer Networks - Workshops*, pp. 956–963, Oct 2012.

[77] Amazon, "Aws greengrass - https://aws.amazon.com/greengrass/," 2007.

[78] Microsoft, "Azure iot edge - https://azure.microsoft.com/en-us/services/iot-edge/," 2013.

[79] X. Xu, S. Huang, L. Feagan, Y. Chen, Y. Qiu, and Y. Wang, "Eaaas: Edge analytics as a service," in *2017 IEEE International Conference on Web Services (ICWS)*, pp. 349–356, June 2017.

[80] "Google Cloud IoT Edge - https://cloud.google.com/iot-edge/," 2015.

[81] "Everyware IoT - https://www.eurotech.com/en/products/iot," 2014.

[82] G. Electric, "Predix - https://www.predix.io/," 2017.

[83] "Bosch IoT - https://www.bosch-si.com/iot-platform/bosch-iot-suite/homepage-bosch-iot-suite.html," 2016.

[84] "Yanzi - http://www.yanzinetworks.com/," 2007.

[85] E. G. Renart, J. Diaz-Montes, and M. Parashar, "Data-driven stream processing at the edge," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pp. 31–40, May 2017.

[86] E. Renart, D. Balouek-Thomert, X. Hu, J. Gong, and M. Parashar, "Online decision-making using edge resources for content-driven stream processing," in *2017 IEEE 13th International Conference on e-Science (e-Science)*, pp. 384–392, Oct 2017.

[87] "Foghorn - https://www.foghorn.io/," 2014.

[88] B. Cheng, A. Papageorgiou, F. Cirillo, and E. Kovacs, "Geelytics: Geo-distributed edge analytics for large scale iot systems based on dynamic topology," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 565–570, Dec 2015.

[89] "Openmtc - https://www.openmtc.org/," 2013.

[90] "Sitewhere - https://sitewhere.io/en/," 2009.

[91] "Smartthings - https://www.smartthings.com/," 2012.

[92] "Kaa - https://www.kaaproject.org/," 2014.

[93] "Samsung artik - https://artik.cloud/," 2016.

[94] "Ayla networks - https://www.aylanetworks.com/," 2010.

[95] "Altair smartworks - https://www.altairsmartworks.com/," 2015.

[96] "Edgex - https://www.edgexfoundry.org/," 2017.

[97] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft, "Picasso: A lightweight edge computing platform," in *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, pp. 1–7, Sep. 2017.

[98] H. Hong, P. Tsai, A. Cheng, M. Y. S. Uddin, N. Venkatasubramanian, and C. Hsu, "Supporting internet-of-things analytics in a fog computing platform," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 138–145, Dec 2017.

[99] D. Pizzolli, G. Cossu, D. Santoro, L. Capra, C. Dupont, D. Charalampos, F. De Pellegrini, F. Antonelli, and S. Cretti, "Cloud4iot: A heterogeneous, distributed and autonomic cloud platform for the iot," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 476–479, Dec 2016.

[100] "Nebulae - http://www.nebulae.io/," 2016.

[101] R. K. Barik, H. Dubey, A. B. Samaddar, R. D. Gupta, and P. K. Ray, "Foggis: Fog computing for geospatial big data analytics," in *2016 IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics Engineering (UPCON)*, pp. 613–618, Dec 2016.

[102] F. Mehdipour, B. Javadi, and A. Mahanti, "Fog-engine: Towards big data analytics in the fog," in *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pp. 640–646, Aug 2016.

[103] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pp. 813–818, Feb 2018.

[104] H. Khazaei, H. Bannazadeh, and A. Leon-Garcia, "Savi-iot: A self-managing containerized iot platform," in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 227–234, Aug 2017.

[105] E. Yigitoglu, M. Mohamed, L. Liu, and H. Ludwig, "Foggy: A framework for continuous automated iot application deployment in fog computing," in *2017 IEEE International Conference on AI Mobile Services (AIMS)*, pp. 38–45, June 2017.

[106] E. Yigitoglu, L. Liu, M. Looper, and C. Pu, "Distributed orchestration in large-scale iot systems," in *2017 IEEE International Congress on Internet of Things (ICIOT)*, pp. 58–65, June 2017.

[107] "Macchina.io - https://macchina.io/," 2014.

[108] "Clearblade - https://www.clearblade.com," 2007.

[109] "Ibm watson iot - https://www.ibm.com/internet-of-things," 2016.

[110] "Cisco iot cloud connect - https://www.cisco.com/c/en/us/solutions/service-provider/iot-cloud-connect/index.html," 2013.

[111] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, pp. 114–131, June 2003.

[112] N. Jiang, A. Quiroz, C. Schmidt, and M. Parashar, "Meteor: a middleware infrastructure for content-based decoupled interactions in pervasive grid environments," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 12, pp. 1455–1484, 2008.

[113] D. S. Hirschberg and J. B. Sinclair, "Decentralized extrema-finding in circular configurations of processors," *Commun. ACM*, vol. 23, pp. 627–628, Nov. 1980.

[114] C. Schmidt and M. Parashar, "Squid: Enabling search in dht-based systems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 7, pp. 962 – 975, 2008.

[115] S. H., "Space-filling curve.," in *Springer: Berlin*, 1995.

[116] "Amazon Kinesis Firehose - https://aws.amazon.com/kinesis/firehose/."

[117] "Sysbench - https://github.com/akopytov/sysbench," 2019.

[118] "RocksDB - http://rocksdb.org/."

[119] "R-pulsar - https://github.com/egibert/rutgers-pulsar," 2017.

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

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

[122] "Azure - https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-quotas-throttling."

[123] R. Young and al., "An architecture for intelligent data processing on iot edge devices," *2017 UKSim-AMSS 19th International Conference on Computer Modelling & Simulation (UKSim)*, pp. 227–232, 2017.

[124] Q. Zhang, X. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: Big data sharing and processing in collaborative edge environment," in *2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pp. 20–25, Oct 2016.

[125] L. Prosperi and al., "Planner: Cost-efficient Execution Plans Placement for Uniform Stream Analytics on Edge and Cloud," in *WORKS 2018: 13th Workflows in Support of Large-Scale Science Workshop, held in conjunction with the IEEE/ACM SC18 conference*, (Dallas, United States), pp. 1–10, Nov. 2018.

[126] "SQLite - https://www.sqlite.org/mostdeployed.html."

[127] "Nitrite DB - https://github.com/dizitart/nitrite-database."

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

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

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

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

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

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

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

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

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

[137] "Climate Change News - https://www.climatechangenews.com/2017/12/11/tsunami-data-consume-one-fifth-global-electricity-2025/."