EDIRO: Edge-driven IoT Resource-aware Orchestration Framework for Collaborative Processing in Large Scale Internet of Things

Niket Agrawal

Delft University of Technology

EDIRO: Edge-driven IoT Resource-aware Orchestration Framework for Collaborative Processing in Large Scale Internet of Things

Master's Thesis in Embedded Systems

Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Niket Agrawal

16th April 2020

Author
Niket Agrawal

Title
EDIRO: Edge-driven IoT Resource-aware Orchestration Framework for Collaborative Processing in Large Scale Internet of Things

MSc presentation
April 24, 2020

Graduation Committee

ir. dr. D. H. J. Epema (chair)	Delft University of Technology
dr. Jan Rellermeyer	Delft University of Technology
dr. Aaron Yi Ding	Delft University of Technology

Abstract

Edge computing extends the cloud computing capabilities to the edge of the network to facilitate processing of the data in the close proximity of its generation. It augments the deployment of several applications in the Internet of Things (IoT) domain which demand low latency and near real-time response for their reliable operation. However, the existing approaches that help accomplish this are inherently static and suffice only for scenarios offering a fixed service at the edge. Thus, rendering them infeasible for a large scale IoT scenario with several heterogeneous services and end users such as a *Smart City*. Among the several challenges in realizing edge computing solutions for such a scenario, the impact of incorporating key aspects of the interactions of the IoT devices and the clients with the edge, remains to be explored. These interactions translate to the ease of use which is vital to determining the degree of adoption of an edge computing infrastructure by the end users and subsequently its profitability for the service provider.

This thesis presents EDIRO, which is an edge-driven distributed orchestration framework for edge computing that enables the edge to drive the workload orchestration through the collaboration of multiple constituent edge nodes. It takes into account the existence of on-demand and ephemeral nature of workloads that require an input, i.e., an *IoT resource* such as an image or sensor data for their execution. These IoT resources are sourced from the end users and the IoT devices in the vicinity. Recent studies that highlight the idea of collaborative processing and the simultaneous presence of data producers and consumers in an IoT ecosystem, support this vision. The underlying concept in EDIRO is the utilization of such IoT resources that are contributed by the end users in the vicinity, to carry out service orchestration for the client requests. To the best of author's knowledge, this is the first work in the edge computing domain to conceptualize the idea of *edge-driven distributed orchestration* and implement a proof of concept followed by its evaluation and practical feasibility analysis.

The main contribution of this thesis is the edge computing orchestration framework EDIRO, which is developed in Golang and released as open source to encourage collaboration with the community. Experiments are conducted on three different types of computing devices emulating edge nodes in a field scenario to determine the practical feasibility of EDIRO. The measurements include the time to serve a client request, the overhead due to the distributed orchestration approach and the computing resource utilization under bursty and normal traffic scenarios. The evaluation suggests that EDIRO is feasible for practical IoT use cases and provides a reasonable trade-off in terms of the benefits offered by this edge-driven approach and the overhead incurred. This thesis shares valuable insights into the ways in which this work opens up the scope of further research in this domain along with the key findings from the system development and experimentation phase.

Preface

As I approach the completion of my masters degree program at TU Delft with the conclusion of this thesis, I'd like to take this opportunity to thank and acknowledge everyone for their valuable contributions during this period.

First of all, I express my sincere thanks to my daily supervisor Aaron Yi Ding for his supervision throughout this thesis. I have thoroughly enjoyed learning about and working on an exciting and emerging topic of IoT Edge Computing under his guidance. I'm grateful for his inputs on the importance of channelizing my ideas well in the context of this thesis and his encouragement during difficult times. His constructive feedback has helped me immensely to learn the scientific approach of addressing research problems. In the process of working under his supervision, I have developed a level of competency necessary for conducting quality and impactfull research.

I'm thankful to Jan Rellermeyer for taking out the time to have discussions with me to help brainstorm potential topics for the thesis, and later, for his valuable inputs at different stages of this thesis. I'd like to thank Dick Epema for chairing the thesis committee. Since this thesis was done across two faculties, several administrative regulations needed to be adhered to. I'm thankful to the Embedded Systems masters program coordinator Arjan van Genderen for his quick responses and clarifications to my queries regarding the same, ensuring that the entire process was hassle-free.

I got the opportunity to present the preliminary results of this thesis at ACM CoNEXT 2019 in Orlando, Florida. This was my first international conference experience and a very valuable and rewarding one indeed. Not only did I get a chance to present my work before a distinguished research community and receive their crucial feedback, but also to learn about the other interesting research works presented at the venue and network with the fellow attendees. I'm overwhelmed by the positive impact that this conference experience has had on me with regards to my future career aspirations. I'm extremely grateful to ACM and the faculty of Technology, Policy and Management (TPM) at TU Delft for providing me with generous funds to facilitate my attendance to the conference. I extend my sincere thanks to my daily supervisor Aaron Yi Ding for arranging the availability of funds on behalf of the TPM faculty and Laura Bruns for prioritizing managing the

logistics on a short notice.

I'm thankful to Fernando Kuipers for his supervision during my internship at TNO and for keeping me updated regularly on the opportunities to participate in various conferences and hackathons. It was only because of this that I got a chance to participate in such events, broaden my knowledge and build a network with the people from the industry and the academia.

Besides my studies at TU Delft, I was also able to pursue some of my newly discovered interests in coaching, mentoring and teaching. I'm thankful to Susanne van Aardenne for giving me the opportunity to mentor the incoming batch of master students in my study program. It helped me develop some valuable soft skills alongside socializing with the other students. I'm also thankful to Matthias Möller and Aaron Yi Ding for giving me teaching assistant opportunities. I enjoyed the role of providing guidance to the students and also the sense of responsibility that came along with it. I appreciate the company of my fellow TAs Anda Filipovic, Sybold Hijkema and Tijs Ziere who made this experience even better. I'd like to also thank some of the organizations that have played an important role in me developing new ideas and providing the platform to interact with several knowledgeable people. I'm thankful to SURF and The Things Network for facilitating my participation in conferences and seminars.

I've been very fortunate to have a wonderful company of friends in the Netherlands during my time here. Thank you Jonathan Lévy, Saumil Sachdeva, Xianhao Ni, Shivanand Kohalli, Dhaval Shah and Magne Hov. Working on projects would've not been as much fun as it was with them. I also appreciate the company of Nilay Sheth, Himanshu Shah, Chinmay Pathak, Eline Stenwig, Wanning Yang, Xingchen Liu, Yu Zhang, Umeer Mohammed and Harris Suwignyo for their help and the time that I spent together with them.

I'm grateful to Floor, Hina, Marianne, David and Eva for welcoming me to stay with them in their house. I was deeply impressed by their gesture during a tough time. The lovely ambiance of their house with its cheerful vibe and their amazing company was a breath of fresh air for me which reflected in my work too. No wonder why several highs that were part of this thesis came during the period when I stayed together with them. Thank you Bram and Tim for introducing me to the WTOS cycling club and arranging all the necessities for me. Spending time cycling together in the beautiful countryside with a wonderful group of people served as a refreshing break from the academics.

Last but foremost, I express my heartfelt gratitude to my parents and elder brother for their unconditional love and support during the time away from home. Their constant encouragement provided me with the much needed source of motivation during my studies. They have always let me chase my dreams and given me the freedom to make the best decisions for myself.

Niket Agrawal

Delft, The Netherlands
16th April 2020

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis proposes EDIRO, a distributed orchestration framework for edge computing which enables the edge to drive the workload orchestration and serve the client requests by utilizing the IoT resources available at the edge of the network. This edge-driven orchestration based approach to edge computing is crucial to addressing the gaps left behind by the literature in their efforts to realize edge computing solutions for the Internet of Things (IoT).

This chapter begins by stressing on the scientific and the practical relevance of the topic of this thesis as the motivation behind this work in Section 1.1. The research questions defined in this thesis are described next in Section 1.2. Illustration of the research methodology, contributions and the thesis outline are captured in Section 1.3, Section 1.4 and Section 1.5 respectively.

1.1 Motivation

Lately, there has been a keen interest towards realizing IoT in an urban context to achieve the goal of a *Smart City*. Under this vision, IoT will be utilized to provide an easy and ubiquitous access to a variety of public services such as healthcare, transportation, waste management, tourism, safety and emergency, etc., while optimizing cost and resource consumption simultaneously [106, 90, 55, 91, 47]. This vision is the key to solving the problem of provisioning, administration and maintenance of the public services in the wake of the increasing population and sustainable livelihood related challenges faced by the urban areas. The prospective benefits that this concept promises to both the city administration and the citizens has led to a push by many government organizations to take a step towards its adoption [102, 75].

Meanwhile, the edge computing paradigm has contributed remarkably to the advancement of the IoT by significantly reducing the network congestion, privacy concerns and the latency involved in the response times for the IoT applications [86, 88, 105]. It has been able to accomplish the aforementioned by bringing the computing and storage resources close to the source of data generation, i.e., the IoT

devices at the edge of the network [1]. Thus, avoiding the need for transmitting the massive amounts of data generated by these devices all the way to the distant cloud data centre for processing and the subsequent wait for the arrival of the results. It is this high latency and the poor response times resulting from the above round trip in the Cloud Computing paradigm that renders the safety critical applications in the IoT domain such as connected and autonomous vehicles infeasible [108, 109, 79, 107]. Edge computing, through its distributed architecture and geographical proximity to the IoT devices, facilitates offloading of computations and data to achieve near real-time response times. This further prevents the network from getting overloaded with the traffic bursts while considerably reducing the attack surface of an IoT application simultaneously.

However, the state of the art in edge computing is inherently static and suffices only for scenarios that offer a particular service at the edge. This limitation stems from the master-slave architecture that the existing approaches employ [101, 2, 50, 96]. It marks the presence of a control plane (master) in the cloud and in a few cases, also at the edge [2], that remotely dispatches, deploys, orchestrates and monitors the workloads that executes on the edge (slave). The major drawback that emanates from such an architecture is the limitation of it not being applicable in the context of a large scale distributed and complex application domain such as Smart City which is characterized by the existence of a plethora of heterogeneous services, devices, systems and end users that utilize these services.

One of the key challenges in realizing edge computing solutions for the IoT is large scale service management and orchestration [88, 100, 54]. Accommodating multi-tenancy, service level agreements (SLA), privacy concerns and the indigenous heterogeneity among the end users calls for a more granular and uniquely distinguishable service management per user or a group of users, for their ***on-demand service requests*** [78, 106, 35]. However, due to the master-slave topology, ***the state of the art renders the edge oblivious to the events and the interactions among the IoT devices and end users in its immediate vicinity***, of which, it lies at the receiving end of. It is these events and interactions which forms the channel for the end users to offload data and computations on the edge or issue on-demand service requests, especially in the context of IoT and pervasive computing. Hence, the state of the art would require the cloud to regularly fetch information regarding the interactions at the edge concerning these on-demand requests. The round trip between the edge and the cloud involved in performing such updates would result in a significantly high latency. It also becomes infeasible for the cloud from the latency

[1] There appears to be no standardized or formal definition for the term *edge of the network*. This term is used throughout this thesis to refer to the imaginary boundary, the immediate vicinity of which, marks the existence of the IoT devices. The cloud resources lie at the other end of this boundary at a far greater geographical distance from it as compared to the IoT devices. Usage of this term is a logical means to convey the physical location of the *close proximity to the source of data generation*, that is often referred to in this thesis.

[2] For simplicity, the term *edge* is used consistently throughout this thesis to refer to the infrastructure consisting of interconnected distributed computing and storage resources placed in close proximity to the sources of data generation to implement the edge computing paradigm.

perspective to monitor the individual workloads executing at the edge and act on them in response to any updates or events at the edge of the network. Besides the high latency, this approach also suffers from failures in the event of an intermittent or total absence of network connectivity between the edge and the cloud, thereby disrupting the transmission of updates and subsequently the orchestration process.

Although the subject of realizing edge computing solutions for large scale IoT has been approached in the literature from different angles and has shown promising results [103, 84, 50, 42, 69], some of the challenges remains to be explored. ***Incorporating key aspects of the interactions of the IoT devices with the edge in the edge computing infrastructure is one such challenge***. It holds a crucial importance from both the user and the service provider's perspective as it translates to the ease of use of the edge infrastructure by the former and subsequently its profitability to the latter and the other stakeholders. One such key aspect is collaborative processing in the IoT context which enables a group of IoT devices to leverage each other's resources on a sharing basis to combine their computational capacity for carrying out a certain computing task.

Collaborative processing has transformed the way IoT devices and clients at the edge of the network interact with and utilize the edge computing infrastructure [60, 61, 103]. In a typical connected vehicles scenario, a vehicle's query about the condition of the road ahead of it is serviced by executing a workload on the nearby edge compute node installed on a traffic light that utilizes the input, i.e., high definition (HD) maps and sensor data offloaded on it by other vehicles in the vicinity [108, 109]. The aforementioned input is termed as an ***IoT Resource***. ***Existing approaches lack to consider the dynamic availability of the mobile IoT resource on the edge in making workload offloading decisions***. This is central to the idea of data locality aware workload offloading on the edge which involves bringing the computations closer to the data. In an ideal scenario, from the user perspective, it should be possible to offload data, computations and requests ubiquitously. On the contrary, ***the master-slave topology in the the state of the art does not offer this transparency at the edge*** and limits the offloading to be done on a specific node(s), thereby requiring the users to be aware of the topology at the edge.

To this, we propose that an edge-driven orchestration architecture possesses the potential to solve all the above highlighted problems. The edge is in the vicinity of the events at the edge of the network and also to the IoT devices that interact with it. Hence, it is both logical and efficient for the edge to drive the workload orchestration considering its capability of utilizing the local knowledge available in the form of IoT resources and context related information to aid in the orchestration process. The scope of collaborating with the other edge nodes to accomplish the above opens up exciting research challenges and opportunities from the distributed systems perspective in edge computing [82]. Although reducing the dependence of the edge on the cloud certainly offers benefits in this context, it also implicitly burdens the resource constrained edge with the task of performing control and management plane operations in addition to executing the workloads corresponding to

the client requests. Investigation of this trade-off could lead to gaining valuable insights into the practical feasibility of this edge-driven approach and the broader idea of an autonomous edge computing architecture. This could not only open up interesting research directions in the edge computing domain, but also benefit the users from an improved user experience and the edge infrastructure provider from the subsequent profits.

1.2 Problem Statement

The primary goal of this thesis concerns with exploring the feasibility of an edge-driven IoT resource-aware orchestration architecture for practical IoT use cases. In this regard, three major research questions (RQ) are defined with further sub-questions identified for each of them as listed below.

RQ1: What is the significance of and the need for an edge-driven orchestration based edge computing architecture for IoT?

a) What are the impacts of such an edge computing infrastructure on the users and edge service providers in the context of IoT?

b) Why are existing proposals inadequate for the evolving IoT environment?

RQ2: How to orchestrate the workloads corresponding to the on-demand service requests by utilizing the local knowledge available at the edge in the form of IoT resources?

a) How to perform IoT resource-aware edge offloading via the collaboration of constituent edge nodes in the infrastructure?

b) How to provide dedicated service management features to an individual or a group of end users in such a scenario?

c) How to accomplish providing edge transparency to the end users to maximize their ease of use of the edge computing infrastructure in place?

d) What are the architectural design and operational challenges associated with this edge-driven distributed orchestration approach?

RQ3: What is the feasibility and the associated trade-off of this edge-driven approach for practical IoT use cases?

a) What is the latency overhead incurred due to the inter edge collaboration and what is the trend of its variation with the increasing load on the system?

b) How does the overhead from the inter edge collaboration affect the handling of the on-demand client requests and what is the trend of its variation with the increasing load on the system?

c) How does the system respond to the traffic bursts that emanate from the data and computation offloads from several clients simultaneously?

d) What is the trade-off between the benefits from this edge-driven approach and the overhead incurred due to the same?

e) What are the limitations of the framework with respect to its feasibility for practical IoT use cases?

1.3 Methodology

The research style of the author can be summarized as one that is driven by identifying and approaching to solve problems of scientific and practical significance in the literature through proof of concept development and measurements. The research methodology followed in this thesis comprises of several phases in which this project was carried out as described below. The answers to the questions concerning the *What, Why and How* for the activities in each phase are also provided below to render further clarity and justification to the research methodology followed in this thesis.

1. Literature survey

The literature survey for this thesis was conducted in two phases. The first phase served as a means to discover a challenging research topic of significant scientific and practical relevance within an emerging but widespread technical domain such as edge computing. This was followed by thoroughly reviewing the current state of research in the selected research topic to identify the existing limitations, open questions, unconvincing implicit assumptions, trends and opportunities which would contribute in constructing the problem statement for this thesis. The outcome of the literature survey was an organized category wise classified collection of the papers published on edge computing. The comments added in the papers were later utilized in writing the literature review for the research proposal.

Inputs from key stakeholders and developer community

A crucial part of the literature survey was the involvement and interactions with several teams and stakeholders directly associated with the state of the art in edge computing for IoT such as Microsoft Azure IoT Edge [2], Amazon IoT Greengrass [1], Kubeedge [14, 101], etc. Valuable insights were gained into several design and deployment aspects of the respective state of the art through discussions with the aforementioned people via active participation in online community forums and

conferences. One of the main highlights of these interactions was recording of some findings in particular that do not directly feature on the online documentations of the aforementioned projects.

2. Formulation of Research Proposal

Post the conclusion of the literature review, a research proposal was formulated containing the literature review, research questions, target use cases, intended contributions and a tentative project timeline. However, it underwent significant modifications later following the redefinition of the scope of the thesis, the problem statement, deprioritization and the subsequent elimination of contributions and ideas that were purely engineering focused.

3. System Design

The goal of the system design phase was to come up with the blue print of the system architecture which can be later implemented as a software prototype using appropriate tools. Since the proposed orchestration framework should be able to answer all the research questions formulated earlier, the very same research questions are laid as the foundations for the development of the system design.

Approach to answering research questions:

- The approach to answering RQ1 is driven by studying the evolving trends in the target application domain and the use case in the context of which an edge computing solution in this thesis is proposed. Because the research question concerns with the significance and the need for the proposed idea, it is logical and important to study its impact on the associated application domain, the end users and the concerned stakeholders. Moreover, it is clearly highlighted why the existing approaches are inadequate in this regard which helps further strengthen the relevance of the proposed idea.

- The approach to answering RQ2 is driven by the architectural and design perspective of the orchestration framework as each of the sub questions maps to a feature that is provided in the framework. The thought process behind the design of the orchestration strategies and the service management techniques concerning the research question was based on studying the requirements of the application domain, i.e., the evolving trends in the large scale IoT environment and the limitations of the state of the art. Hence, this approach is in synchronization with the ultimate goal of the orchestration framework and is the ideal choice for answering the associated sub-questions.

- In order to convey the overhead incurred due to the proposed approach in RQ3, certain measurement metrics are identified and relevant experiments are devised to measure it in different scenarios. The main criteria for the

selection of these metrics was based on the overhead that emanates from the frequent communication and synchronization in the operation of a distributed system, as is the case in the proposed orchestration framework.

4. Proof of concept implementation and experiments

This phase was marked by the conscious design choices that are made for the tools, techniques, hardware and software platforms to be utilized for the software development of the orchestration framework and its subsequent evaluation. These design choices include the selection of the programming language, virtualization technology and a baseline orchestration platform which could be further extended with the proposed framework. The most common development tools and techniques in the context of edge computing were compared and weighed against each other based on their pros and cons before proceeding to select one. Besides being a systematic way of going about implementing a proof of concept, this exercise helped in gaining key insights into some of the tools and discover their capabilities from system design perspective especially in the context of edge computing.

1.4 Contributions

The contributions in this thesis are summarized as follows:

1. Identification, explanation and documentation of the current research and knowledge gap that exists in literature of edge computing for IoT in form of a gap analysis. It renders insights into the shortcomings of the current research in this domain and lays emphasis on the need for an alternative architecture for edge computing through relevant justifications and examples.

   This part of the work addresses the research question, *"What is the significance of and the need for an edge-driven orchestration based edge computing infrastructure for the IoT?"*

2. Design and implementation of a distributed orchestration framework in Golang for edge computing that is capable of utilizing the local knowledge available at the edge in the form of IoT resources to orchestrate the workloads.

   This part of the work addresses the research question, *"How to orchestrate the workloads corresponding to the on-demand service requests by utilizing the local knowledge available at the edge in the form of IoT resources?"*

3. Experimental evaluation and practical feasibility analysis of EDIRO providing insights into the into the ways in which the work carried out in this thesis opens up the scope of further research in this domain along with the key findings from the system development phase.

This part of the work addresses the research question, *"What is the feasibility and the associated trade-off of this edge-driven approach for practical IoT use cases?"*

4. Open source contribution: The source code of EDIRO is released as an open source at [5] to encourage collaboration with and experimentation by the research and developer community related with edge computing.

5. Research contribution: The proposed idea and the preliminary results of this work were published at the poster session of ACM CoNEXT 2019. The associated paper published for the same is available at [26]. This activity served as an effort to make a contribution to the research community by sharing the valuable insights into the ways in which the work carried out in this thesis opens up the scope of further research in this domain along with the key findings from the system development phase. It was also a means to receive feedback on the work which could be incorporated for further improvements.

1.5 Thesis Outline

The remainder of the thesis is structured in the following way.

- **Chapter 2: Background and Related Work** provides the technical background and the literature survey for the work carried out in this thesis.
- **Chapter 3: EDIRO: Edge-driven IoT resource-aware Orchestration Framework** describes the design and implementation of EDIRO, i.e., the proposed edge-driven distributed orchestration framework for edge computing.
- **Chapter 4: Experiments and Results** presents the experimental evaluation of EDIRO.
- **Chapter 5: Conclusions and Future Work** summarizes the contributions, provides answers to the research questions defined in the thesis, makes recommendations for future work and presents the concluding remarks.

Chapter 2

Background and Related Work

This chapter provides the technical background and presents the literature survey conducted for this thesis. Section 2.1 provides the necessary background, definitions of core concepts and terminologies related to edge computing to equip the reader with the required technical competency in order to understand and appreciate the work done in this thesis. Section 2.2 introduces the underlying technologies and concepts that enable the edge computing paradigm. Section 2.3 describes the impact of edge computing on the IoT by highlighting relevant application scenarios and gives on overview of the core challenges concerning the realization of an edge computing solution for the IoT. This is followed by the literature review of the existing work in the domain of edge computing for the IoT. The chapter concludes with presenting a gap analysis in Section 2.4 highlighting the unexplored challenges and open questions as identified by a detailed analysis of the key findings from the literature survey.

2.1 Edge Computing Overview

Edge computing is an emerging technology which has had a profound impact on the Information, Communication and Technology (ICT) domain in a quick span of time. This is evident from the growing research interests towards it from the academia and investments from the industry. It has rendered a significant contribution in enabling the modern trends in the ICT domain such as IoT, mobile and ubiquitous computing by addressing the key concerns and shortcomings that arise from the Cloud Computing model.

In order to set up the context for the work done in this thesis, understanding why and how edge computing came to the fore in the first place in a space that was dominated by the cloud including the very concept of it, is of paramount importance. This chapter begins by explaining the same in Section 2.1.1. This is followed by introducing the related concept and the explaining the edge computing architecture in Section 2.1.2.

2.1.1 The Shift from Cloud Computing to Edge Computing

Over the years, Cloud Computing has been the key enabler of the deployment and distribution of infrastructure, services and applications in the information technology domain [81, 31]. Enterprises leverage the infinite compute, storage and networking resources in the cloud through a convenient *pay as you go* model for their routine business operations and to deploy services and applications for their customers [52]. The Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) and serverless computing models [62, 68, 57, 98] offered by the cloud liberates the enterprises and the individuals from having to possess, operate and manage the hardware and software assets themselves in order to use the aforementioned services. To put things into perspective, all one needs is an internet connection on their personal computing device in order to access an otherwise computationally and storage intensive application which is hosted on a cloud server at a large data centre in some part of the world. The ubiquitous adoption of Cloud Computing has made it possible to access desired media content over the internet delivered by platforms such as Netflix, download and use billions of mobile applications on a smartphone, leverage high end servers to train a machine learning model or set up an IT infrastructure for an entire organization in a matter of few seconds to minutes among many others.

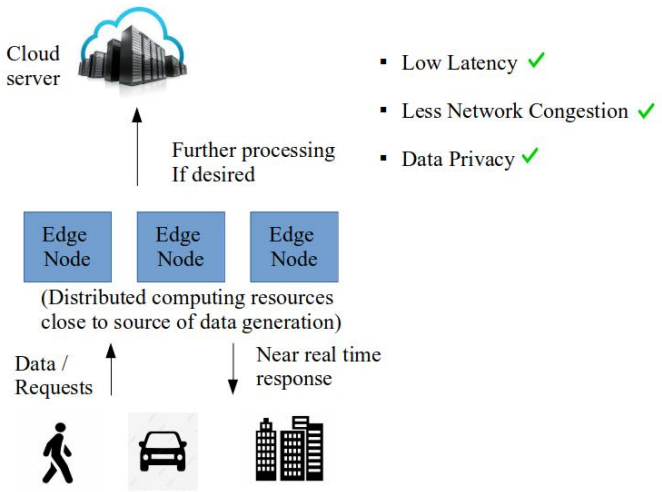


Figure 2.1: Shift from Cloud to Edge Computing

With its highly available centralized pool of abundant resources, cloud becomes the ideal choice for offloading data intensive and resource greedy computations or workloads. However, this approach fails to deliver when it comes to latency sensitive applications especially in the Internet of Things (IoT) domain as explained in

the discussion to follow.

The arrival of the IoT has witnessed a proliferation of devices or the *Things* embedded with processing and sensing capabilities [28, 32]. These devices account for a significant presence on the internet owing to their connections to the mobile applications and other similar devices. Such devices have a ubiquitous presence around us by virtue of their integration into wearables, smartphones, homes, transportation, urban infrastructure, etc., and are characterized by their limited computation capabilities and battery life. In a cloud computing infrastructure, the data generated by these devices is offloaded to the cloud servers for processing from the perspective of the end to end IoT applications. Lately, there has been an emergence of IoT applications in the domain of autonomous and connected vehicles, smart city, security and surveillance which enable innovative use cases. These applications demand near real time response and low latency for their reliable operation [106, 45, 48, 99].

However, Cloud Computing fails to deliver on this front due to the following reasons.

1. The Round Trip Time (RTT) from the IoT device to the cloud corresponding to the transportation of the entire data from the device to the cloud for processing, to the reception of the results back to the device, is beyond the latency and real time response requirements for the applications in discussion [30, 108]. Taking into account the criticality and the context of these applications, a violation of the latency requirements could lead to fatal consequences.

2. Considering the substantial increase in the number of IoT devices and subsequently the data produced by them, the network bandwidth consumption increases manifold due to the data transaction to and from the cloud which ultimately puts the network under immense load further worsening the scenario.

3. Transmitting raw data from the IoT devices over the network to the cloud raises privacy and security concerns.

Edge computing is a computing paradigm that facilitates computation and processing close to the source of data generation by extending the cloud resources to the edge of the network [86, 88]. This is achieved via an infrastructure consisting of interconnected distributed computing and storage resources, commonly referred to as *edge nodes*, placed in close proximity to the sources of data generation itself as shown in Figure 2.1. By doing so, it compliments the cloud by extending its processing capabilities to the edge of the network, although in a limited capacity. The practical realization of edge computing paradigm has been made possible by the technological advancements in the hardware aspect of the IoT which have marked the arrival of low cost stand alone embedded computing devices and single board

computers. The small form factor and sufficient computing capabilities that these devices possess makes them ideal for utilization in the context of edge computing [70].

2.1.2 Related Concepts and Architecture

The idea of provision computing and storage resources at the edge of the network is the key to ensuring service delivery guarantees to the end user and devices besides enabling several applications. This subsection introduces the key concepts and technologies enabling computation at the edge of the network followed by the architecture of edge computing. Although referred in the literature by different names, these concepts share the same end goal and differ only slightly in their approaches and the underlying implementation. The following discussion is structured in a manner which offers insights into the evolution of this idea over the years since its inception and how the technological trends around it have influenced it to a state that it is at present.

A. Cyber Foraging

Cyber foraging, proposed by Satyanarayanan [85], is among the first works in this regard. In this concept, a mobile device with limited computational resources leverages the rich computing resources of servers available in its close proximity to augment its computational capabilities. The servers referred to in this work are termed as *surrogates* and specified as the desktop computers with wired access to high bandwidth internet. In a typical scenario, a mobile device could offload a computation heavy task to such a surrogate server in the vicinity which can then make use of its computation resources and the high speed internet connection to perform the computation and fetch the required data in quick time. After the processing, the results are conveyed back to the mobile device. The underlying technique in cyber foraging is *computation offloading* which is discussed in detail in Section 2.2.2.

B. Cloudlet

The concept of Cloudlet was introduced in 2009 in order to adapt cyber foraging for mobile computing [87]. A cloudlet is a resource rich micro data centre comprising of a single computer or clusters of computers with access to high speed internet connection. Presence of such multiple cloudlets facilitate seamless computation offloading by mobile clients. The cloudlets are implemented using the virtual machine (VM) technology in order to meet the demands of the mobile clients to be able to instantiate multiple diverse and customized applications on the cloudlets with specific run time and operating system requirements.

C. Mobile Edge Computing

Mobile edge computing (MEC) is an initiative by ETSI (European Telecommunications Standards Institute) to provide cloud computing resources at the edge of the mobile networks within the radio access network (RAN) [25, 41]. The RAN is part of the cellular network infrastructure which connects the end users to the core network of their respective mobile network provider through which cellular services are offered. The large geographical area covered by RAN is split up in specific sectors which are covered by individual base stations. MEC servers with computing and storage resources are installed on these base stations for the end users to leverage them. MEC is a key enabling technology for 5G as the end users can benefit from leveraging the computing and storage resources at the edge of the mobile network accessible over high speed 5G internet connection. MEC servers are operated by the mobile network infrastructure provider. Being co-located with base stations, MEC servers benefits from the additional features of accessing position and mobility related information of the end users by being co-located with the base station [84]

D. Fog Computing

Fog computing was introduced by Cisco in 2012 [73]. It shares the same end goal as edge computing which is to facilitate computation at the edge of the network by extending the cloud resources there. There are numerous overlapping definitions and interpretations of fog computing in the literature. A common notion among these is that fog computing lays specific emphasis on the interaction between the resources at the edge of the network and the cloud [73]. Thus, focusing on the networking aspects in particular. On the other hand, edge computing is typically associated with provisioning the computation capabilities at the edge of the network and is often associated with the compute and storage infrastructure rather than its communication and connection with the cloud. However, both edge and fog computing aims to solve the same problem and share similar design philosophy and technical challenges. Since this thesis is concerned about the design, development, and feasibility analysis of an edge computing infrastructure that can operate autonomously with minimal support from cloud, the term 'edge' is used throughout this work for consistency except when referring specific works in the literature which use the title 'fog' explicitly.

Edge Computing Architecture

The edge computing paradigm extends the cloud resources to the edge of the network, i.e., in close proximity of the sources of data generation. The most widely used and referred to architecture of edge computing is the three tier architecture [74] as shown in Figure 2.2. The description of the entities that constitutes these tiers is as follows:

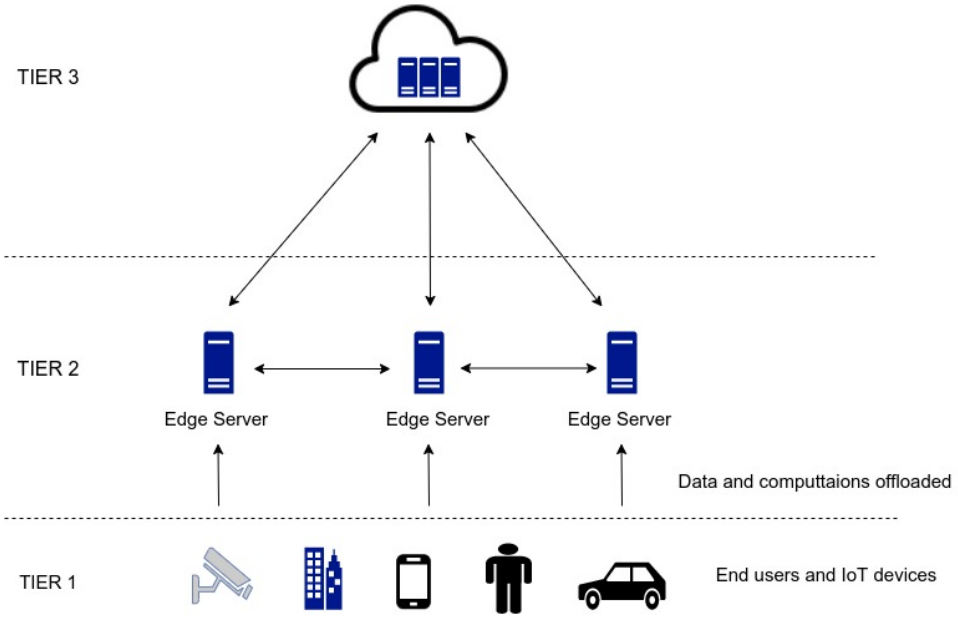


Figure 2.2: The three tier edge computing architecture

1. Tier 1 comprises of the end devices which are the source of data generation. These could be the sensors installed in an IoT environment, mobile phones, handheld mobile computing devices, vehicles embedded with IoT sensors, etc.

2. Tier 2 constitutes the edge layer which comprises of one or more interconnected compute servers and storage resources. These are the resources which are extended from the cloud towards the edge of the network in order to support the applications and end users by facilitating computation offloading.

3. Tier 3 represents the cloud data centre which houses a large pool of abundant resources and is typically characterized by centralized management and processing of computations. Often the cloud is responsible for managing the workload deployment and management operations at the edge while also acting as an offloading destination itself for the computations that are too resource heavy for the compute resources at the edge.

While the three tier architecture is the most basic and logical architecture for edge computing, it has been revisited by several studies in the past according to the requirements of the target application and strategies of resource management. Variations and modifications to this exist as per target application domain and the

enabling technologies used. For example, [72] proposed a multi-tier cloud architecture deployed over the geographic span of the network. It provides compute and storage resources of varying capacities in succession along the path between the edge of the network and the cloud. The authors termed the proposed computing paradigm as path computing.

2.2 Key Characteristics and Enabling Technologies

This section introduces and discusses three main contributing technologies and key characteristics of an edge computing infrastructure, namely, lightweight virtualization, computation offloading and service orchestration.

2.2.1 Lightweight Virtualization

Edge computing deals with the deployment of a multitude of applications which have dependencies on specific operating systems, run time environments and library packages. Furthermore, the privacy and security regulations demands these applications to be executed in isolation from other applications on the same edge server. In order to meet these requirements, the edge computing paradigm utilizes virtualization technology to facilitate provisioning of multiple heterogeneous applications by providing them a virtualized build and runtime environment. The sandbox execution environment provided by these technologies is ideal in this scenario.

This subsection discusses of the impact of the state of the art in lightweight virtualization technology on the edge computing paradigm. It describes how the usage of virtualization technology in edge computing has evolved over time with respect to the impact of the evolving characteristics and trends in the IoT domain on edge computing.

A. From Virtual Machine to Containers

The first use of an underlying virtualization infrastructure in edge computing was done in the form of virtual machines in Cloudlet [87]. This was the first step towards improving the scalability of edge computing by enhancing the capability of the edge servers to simultaneously accommodate multiple applications with different operating system and run time dependencies. The sandbox like functionality offered by VMs and the fact that multiple of such VMs could run in isolation with the help of a hypervisor on a single computer, opened the doors for its quick adoption in edge computing. However, the large memory footprint, high I/O overhead and slow boot up time of VM is a major bottleneck in the context of edge computing.

An operating system based virtualization technology such as container is able to solve the aforementioned issues that exists in VM. A container is basically a running instance of a container image which is created by packaging the application

code together with the required set of libraries and other dependencies. Since multiple containers on a machine share the same host operating system's kernel, the size of the application images are significantly smaller as compared to a VM. This is conveyed by an architectural comparison between VM and containers as shown in Figure 2.3. It is for these reasons that container is currently the go to choice for the underlying virtualization technology in edge computing [39, 63, 77, 33, 27].

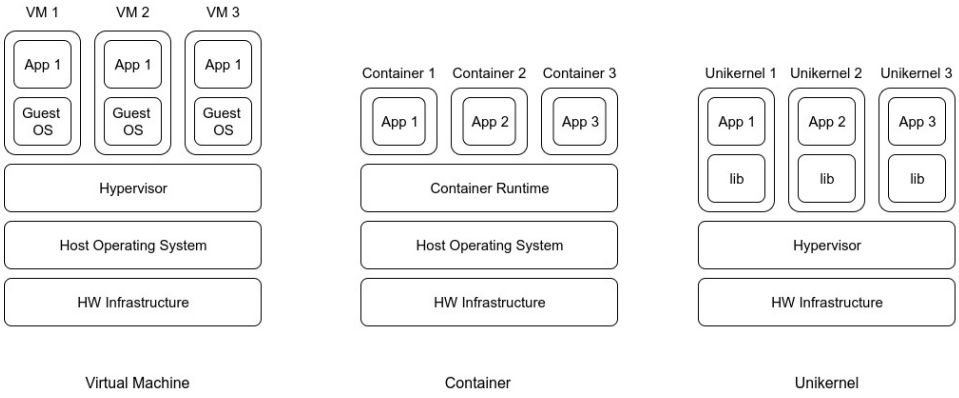


Figure 2.3: Comparison of VM, Containers and Unikernels [24]

B. Unikernel

Unikernels are specialised, application specific, single address space machine images created using library operating systems [64, 23, 65]. They are a very resource efficient way of deploying services on the cloud. A developer can create a standalone kernel by compiling the application code written in a high level language bundled together with the libraries required by the application. These libraries are offered by a library operating system such as Mirage OS [17] under a programming framework which provides the underlying operating system constructs that are required for the application's execution. The resulting machine image, i.e. the unikernel, can be booted on hypervisor such as Xen or KVM. As conveyed through Figure 2.3, unikernels are considerably lightweight as compared to VMs and containers and can boot in milliseconds since they only contains the libraries as required by the application, thereby reducing the attack surface on the application simultaneously [23].

C. MicroVM

The choice of selecting either VM or containers comes with its own set of advantages and disadvantages. VM provides better security and workload isolation features but suffer primarily from a large memory footprint, while containers drastic-

ally reduce the memory footprint as compared to virtual machine but compromise on the level of security and workload isolation aspects. To this end, Firecracker is a new open source virtualization technology from Amazon that brings the best of both worlds [6]. It enables the deployment of lightweight virtual machines or microVms. Firecracker in essence is a virtual machine monitor (VMM) that uses KVM as the underlying virtualization technology to run and manage microVMs. It was initially planned to be utilized for development and experimental purposes in this thesis with a motive to make the first effort to benchmark it for IoT edge computing use cases and generate insights into its capabilities. But, this decision had to be done away with in the interest of time and the lack of community support because this technology was still in early development phase then.

Takeaways

Although by providing better security and isolation among the various applications executing on the host machine VM forms the ideal choice for supporting multi tenancy at the edge, its large memory footprint, high I/O overhead and slow boot up time makes its utilization infeasible for large scale IoT scenarios. In such cases, the use of VM becomes a bottleneck on resource constrained edge servers due to the issues highlighted above. Containers are the preferred virtualization technology in edge computing these days due to their small memory footprint and fast boot up times [39, 63, 94]. Presence of several open source projects offering container runtime, full fledged platforms and even tools to build a customized container platform [18] further make their usage convenient and popular [77, 33, 27]. The latest lightweight virtualization technology such as unikernels possesses even smaller footprint than containers and quicker boot up times. Unikernels are relatively new in the virtualization technology world, however, their benefits have attracted them to their utilization in edge computing. Recent researches have exploited unikernels to run computations at the edge and have shown promising results [71, 36, 37]. While unikernels offer several benefits over containers and VMs, their adoption in the edge computing domain is still in a very early stage. The downside being that unikernels need to be built from scratch which can prove to be a significant engineering effort based on the complexity of the base application. Recent works in this area in the form of programming frameworks and open source projects [11, 34] is an encouraging sign and the adoption of this technology in edge computing domain is expected to pick up pace in near future.

To summarize, the development of lightweight virtualization technology has resulted in a significant positive impact on edge computing. Besides facilitating executing multiple different computations on the edge servers packaged as applications with a tiny memory footprint, they also play a vital role in enabling an important phenomena in edge computing as discussed in the following subsection.

2.2.2 Computation Offloading

Computation Offloading enables the end users with resource limited computing devices to offload any compute-intensive or latency-sensitive tasks to a more powerful computing device in the vicinity possessing rich processing capabilities. In the context of edge computing, this phenomena is central to facilitating the end users to leverage the compute and storage resources on the edge servers and also on the mobile computing devices possessed by other clients in the vicinity. Hence, computation offloading is the backbone of edge computing in many ways. It is the way of delegating the computation to a capable edge server or device to ensure service guarantees to the end users and reduce the delay in response to the client requests.

Computation offloading is a phenomena in edge computing that is applicable to both the end users and the edge infrastructure, and can be studied or analyzed from either perspective. The end user perspective of computation offloading is concerned with relaying a service request, transmitting the raw data or offloading a computation on the edge servers. This procedure involves the discovery of the appropriate edge server(s), the subsequent communication and data exchange with it which goes out of the scope of this thesis. Hence, the discussion related to it is skipped. The edge infrastructure perspective of computation offloading is of interest in the context of this thesis as it is an integral part of the edge orchestration process. The computation offloading procedure in edge computing involves two main steps concerning with the questions concerning ***What to offload?*** and ***Where to offload?***. A brief description of each step is provided below and highlight the most common approaches undertaken in the literature to implement them.

Step 1. Application Partitioning

Application or task partitioning is the first step in the computation offloading procedure and it corresponds to answering the question concerning ***What to offload?*** in edge computing. This practice is carried out to accelerate the process of executing a workload at the edge by splitting the task into multiple smaller tasks which can then be offloaded to either the other edge nodes in the vicinity, to the cloud or to the IoT devices as well. This practice is utilized in scenarios when either the size of the offloaded task is big enough to be handled by a single resource constrained edge node or when certain components of the task are best suited to be handled by specific edge nodes considering the parameters such as data locality, computing resource availability, etc. This mechanism is of significant importance especially from a distributed edge computing architecture point of view. An effective task partitioning scheme and its subsequent distribution to the constituent nodes in the system is the key to ensure quick servicing of client requests by breaking computations into smaller chunks, distributing it among the edge servers and operating on them in parallel. The computations offloaded to the edge servers by the end users is partitioned on the basis of certain set of constraints. These constraints are specific to the application or the desired optimization parameter such as cost, latency,

energy, etc. [53, 59]. For instance, Tsai et al. proposed a fog computing platform for distributed analytics in which the multimedia applications such as crowd detection is split into three separate sub components using TensorFlow and deployed on respective edge nodes [96].

The downside of application partitioning approach is the post processing accumulation of the results from all the participant nodes and conveying the results back to the end user. The overhead will only grow with increasing complexity of the applications. Hence, application partitioning is dependent to a large extent on the feasibility of partitioning the application itself into smaller chunks of computations which can be executed independently and then aggregated towards the end to generate the final results.

Step 2. Determining Offloading Destination

This step corresponds to answering the question concerning ***Where to offload?*** in the computation offloading process of edge computing. From a broader perspective, an edge computing infrastructure comprises of the cloud, the edge servers at the edge of the network and the IoT devices in possession of the end users. Although the edge servers are the most appropriate choice to execute the offloaded and subsequently partitioned computations, recent studies have also explored the prospects of offloading the computations to the IoT devices held by end users and also to the cloud. The possession of sufficient processing capabilities in the IoT devices as a result of the advancements in embedded computing technology has made it feasible for them to contribute to the processing of the workloads offloaded at the edge. For instance, Yigitoglu et al. proposed an edge computing infrastructure that intelligently partitions the tasks in real time based on the capacity possessed by by the edge devices and servers to execute them [103]. In this manner, besides the edge servers, a chunk of the computations offloaded by mobile clients can also be handled by other mobile clients in the vicinity. Offloading a part of computations to the IoT devices is an effective and clever way of exploiting the computing resources on mobile devices. However, in order to accomplish this, the end users must be a part of the edge infrastructure itself and must identify themselves with it.

On the other hand, offloading part of the computations to the cloud is a way for the edge to relieve itself from computation heavy, latency insensitive tasks or utilize cloud as a backup for handling client requests in case of peak loads. For instance, Tong et al. proposed a hierarchical edge cloud architecture to efficiently serve the peak loads of the offloaded computations from mobile users [95]. They developed algorithms for workload placement to distribute the loads between the edge and the cloud. Their results showed that their hierarchical edge cloud approach outperformed the flat edge model by more than 25% when considering the average delay in program execution. Bruneo et al. proposed a fog computing platform for Smart City applications in which a major part of the workload was executed on the end devices [35]. Kar et al. leverage the vehicles as edge compute nodes for estimating the traffic on the road by processing the video streams from front facing cameras

[56]. The decision regarding determining the offloading destination can be based on several factors and optimization metrics such as cost, latency, energy, mobility, etc. as explored in the literature [53, 59, 104]. Among these, two main techniques of computation offloading are discussed particularly given their relevance in this thesis.

A. Computing resource-aware offloading

Computing resource-aware offloading is driven by the availability of the computing resource such as the RAM, CPU, disk space, etc on an edge node to execute a workload. In this technique, the incoming computation at the edge is offloaded to the edge node which possesses sufficient computing resources to execute it. Such an offloading scheme requires continuous monitoring of the resource availability on all the edge nodes that constitute the edge infrastructure. Typically, this is accomplished via a master or a central controller entity either on the edge or in the cloud. Recent research have also proposed techniques to further compliment this offloading technique by monitoring the environmental parameters that may cause potential impact like network bandwidth. For instance, Albalawi et al. proposed an architecture for in network computing called INCA that considers the availability of computing resources and the current status of the network bandwidth as the basis of workload placement across the edge and the cloud [29]. The main objective of their work is to achieve join optimization of the computing and networking resources in order to reduce the delay in handling client requests.

B. Edge Offloading

Edge offloading is a technique proposed by Cozzolino et al. in their work on developing an edge offloading architecture called FADES [36] which enables applications to be shipped to the edge from the cloud. The objective of this architecture is to carry out small and lightweight tasks at the edge by directly taking advantage of the data locality there. The execution of small task fragments on the edge further enables to partially hide the core of the application logic that executes on the cloud. The authors utilize unikernels as the lightweight virtualization technology to package applications with tiny memory footprint which can be offloaded to the edge. The aforementioned also answers the question concerning ***How to offload?*** in the computation offloading process.

2.2.3 Service Orchestration

A key characteristic of an edge computing infrastructure is the presence of geographically distributed micro clouds servicing IoT devices and end users across various sites. Providing edge computing services to a diverse group of end users and devices involves provisioning of as many heterogeneous applications on the edge servers. In order to optimize the resource utilization on the resource constrained edge servers and assure the guaranteed QoS to such a large number of end

users, an automated deployment and management of workloads becomes crucial. Such an automation can intelligently determine the workload placement locations for the end user requests based on either a predefined set of optimization constraints or actively monitoring the current status of parameters like network bandwidth, availability of sufficient computing resources (RAM, CPU, disk space, etc.) on the edge servers, context related information or correlation among the incoming requests and the reusability of the intermediate results produced.

This automated management and deployment of workloads and resources at the edge is termed as orchestration. A more formal definition for the same is provided by [38] which defines orchestration as a process which coordinates the interactions between different applications and services at the edge of the network. Service orchestration is the key to automated handling of user requests and computation offloads, deployment of respective applications, management and allocation of required software and hardware resources in order to deploy the applications on the edge. A major differentiation factor among the many edge computing architectures and solutions proposed for the IoT is the architecture of the orchestrator and the underlying orchestration technique they employ. At its core, the orchestrator implements the computation offloading strategy and handles the execution and management of the applications that run on the edge.

Containers are among the most commonly utilized underlying virtualization technology in edge computing currently for executing applications at the edge servers. Thus, the orchestration of containerized applications is an important feature in modern edge computing solutions for the IoT. The following subsection provides an overview on the state of the art in container orchestration, related concepts and the review some of the relevant works in the literature.

Container Cluster Management and Orchestration

Following are the main features that forms the core of the container cluster management and orchestration solutions in the edge computing domain [77].

- Placement of containerized applications on specific edge nodes in the cluster. This ensures portability and interoperability among the applications executing on the edge.

- Provision of adding or removing edge nodes from a cluster. This facilitates scaling up the edge infrastructure by adding new edge compute nodes in the events of peak load or decommissioning an edge node in an event where the security of an edge node is compromised.

- Resource allocation to the containerized applications. Resources mentioned here refer to RAM, CPU, access to disk storage on the edge compute node and networking capabilities to communicate with other applications on the edge cluster or exchange information with the outside world.

- Fault management: Facilitates the restart of the applications in case of failure during execution and their migration to another active edge node on the cluster in case of an event where the host edge node suffers a downtime.

There exists several platforms for container cluster management and orchestration such as Docker Swarm, Kubernetes, Mesos, etc. which offer the above mentioned features. A brief overview of Docker is provided below in particular as it is extensively utilized for implementing edge computing solutions in various application domains [27, 76, 38, 49]. It not only provides the virtualization technology infrastructure that enables executing applications as containers, but also provides inbuilt cluster management and orchestration features.

Docker

Docker is an open source platform that enables building and running applications as containers. In order to run an application as a container, a corresponding container image referred to as the *Docker image*, needs to be created. The process of creating a Docker image involves creating a text document called as a *Dockerfile* specifying the dependencies for the base application such as operating system, programming run time, support libraries, etc. and any commands or metadata needed to run the container. Docker provides command line utilities to create a container image for the base application from a Dockerfile and to subsequently launch and monitor the containerized applications. The description of Docker's internal architecture, APIs and the underlying technology is skipped to keep the discussion here relevant to just introducing the main features offered by Docker in the context of its usage in edge computing.

The container cluster management and orchestration features in Docker are provided through a dedicated mode called as the *Swarm mode* [22, 21]. Swarm mode turns a cluster of multiple separate physical Docker host machines into a logical single large pool of compute nodes where containers can be deployed and migrated across. The control and core control and management responsibilities are fulfilled by dedicated *manager* node(s) while the *worker* nodes execute the workloads. The workloads deployed on a swarm of docker hosts are referred to as a *service*. A desired state of the service is required to be mentioned while creating the service. This is done by specifying certain parameters in the configuration file or on the command line at the time of service creation. These parameters may include the desired number of replicas of the service running on the cluster, opening of certain ports to enable networking, access to specific storage volumes on the host machine to fetch data or store intermediate results, etc. The built in orchestrator works to maintain this desired state by continuously monitoring the service. For example, if three replicas of the service was requested to run and one of the instances suddenly crashes, the orchestrator will instantiate another instance of the same service to maintain the desired state.

2.3 Edge Computing for the Internet of Things

IoT, which is typically characterized by the devices and applications that often operate in geographically distributed locations and with a significant degree of mobility [58, 93, 97], benefits immensely from the edge computing paradigm by being able to exploit the compute and storage resources available in their close proximity at the edge of the network [89, 51, 67]. At the same time, the IoT devices also prolong their battery lives by being exempted from transmitting the data to the distant cloud data centres. Besides supporting the deployment of the latency sensitive safety critical applications, the edge computing paradigm also provides better QoS (Quality of Service) to the end users as compared to the cloud due to the significant reduction in latency achieved by the former.

2.3.1 Application Scenarios

This subsection discusses three main application scenarios in IoT in which edge computing plays an important role. These also form the ***target use case*** for the work done in this thesis as the problems that are being tackled in this thesis apply directly to the use cases in these application domains.

A. Smart City

Smart City is the concept of applying IoT in an urban context with the objective of providing a range of services to the citizens by setting up an information and communication infrastructure that integrates the IoT technology with the individual sectors and industries associated with these services. In this manner, services such as healthcare, transportation, waste management, etc. will be rendered under one umbrella by the city administration for the citizens to avail. This concept leads to an efficient use of public resources, offers quality of service offered to the citizens while reducing operational cost of managing these services. Recent studies such as [43, 44] have explored setting up edge infrastructure for smart cities by placing computing and storage resources in base stations, traffic lights and other public infrastructure. For instance, Taleb et al. proposed an approach to enhance the real time video streaming experience of the users by utilizing mobile edge computing (MEC) [92]. Sapienza et al. proposed an MEC based approach to detect critical events in a city and trigger notifications to end users in response [84]. They processed the crowd sourced data from various IoT devices such as surveillance cameras at the edge servers installed on the mobile base stations to detect an abnormal activity or a critical event. The base stations then cooperate with each other and broadcast alerts to a large group of people under their coverage area.

B. Connected and Autonomous Vehicles

The use of edge computing in automotive and transportation domain has rendered an enhanced and safe driving experience to the drivers. In such a scenario, the

roadside infrastructures such as traffic lights, street lights, lamp posts, etc. are embedded with computing and storage resources that facilitate offloading of the data captured by the sensors on the vehicles. By enabling features such as vehicle to infrastructure (V2I) communication and vehicle to everything communication (V2X), edge computing provides the vehicles with a better and quick knowledge about their surroundings in form of alerts such as unsafe road condition, traffic congestion etc. For instance, Zhou et al. proposed a technique to augment a driver's view by rendering a 3D map generated from the data shared by other vehicles by an edge server placed on the roadside infrastructure [108].

C. Collaborative Processing in IoT

The end users with their IoT devices at the edge of the network not only act as data producers that continuously offload data to the edge, but also as data consumers that request services from edge. Thus, there is a dual and simultaneous presence of such data producers and consumers in an IoT environment [88]. Meanwhile, the computing capabilities of the IoT devices is only improving by the day, thanks to the advancements in embedded computing technology that packages sufficient computing power in credit card sized single board computers. Slightly more sophisticated end user devices such as smartphones now feature RAM and CPU specifications comparable to a desktop computer. Enabled by wireless connectivity and 4G/5G connections, further makes them a self-sufficient mobile computing device. Another example in this category include the modern vehicles which are embedded with an array of sensors and inbuilt computing capabilities. Such presence of end devices with sufficiently rich computing capabilities offer rich opportunities for collaboration with the edge servers in handling the offloaded computations.

This is an effective and innovative way of approaching to solve the challenges of utilizing edge computing for large scale IoT , i.e., by exploiting the rich presence of computing capabilities within the end user devices. Although there are several factors that needs to be taken into consideration such as determining which IoT device(s) can participate in processing the computations together with the edge servers and security concerns regarding this participation to avoid penetration by a malicious IoT device that could access a chunk of computation. There are mainly two types of collaborative processing scenarios in edge computing. In the first category, the computations offloaded to the edge servers are partitioned and distributed intelligently among the edge, the cloud and the IoT devices or commonly referred to as the edge devices. In this case, the end user devices collaborate with the edge by sharing and contributing their computing resources. The second category refers to the scenario where the data produced by end device(s) is aggregated at the edge servers to serve other end user(s). This category is the one where the end device share and contribute the data generated by them to achieve the end goal of serving other end users.

2.3.2 Key Challenges

While there are numerous benefits that edge computing offers to the IoT [40], there are several challenges that exists in its utilization for IoT [88, 54, 100]. An overview of the main challenges that exist in applying edge computing to the IoT is provided below.

1. Service Orchestration: A large scale IoT scenario is characterized by a large number of applications that need to be provisioned at the edge in order to cater to demands of several end users at the same time. This requires an efficient orchestration strategy for workload orchestration and resource management at the edge servers in order to ensure that the edge servers are able to handle many concurrent client requests in manner that provides a sufficient QoS to them. It is also a challenge to find effective ways to provision several different applications on the resource limited edge servers.

2. Interoperability : IoT is characterized by presence of several devices in form of mobile computing devices, IoT sensors, personal hand held devices, cell phones, wearables, etc. which use different communication technologies to interact with and exchange information with the edge resources and among themselves. Also, the edge resources in a large scale IoT could be owned by different service providers and hence may differ vastly in the hardware they possess, the applications they can run, etc. It is crucial to have these different technologies to operate seamlessly and interact with each other if the goal of a edge computing solution for IoT needs to be realized.

3. Scalability: When operating in a large scale scenario, scalability of an edge computing solutions becomes important as it determines how effectively it can handle large number of end users, adapt to changing traffic conditions, scale horizontally as well as vertically. Horizontal scaling refers to scaling up the number of instances deployed for an application in order to serve several service requests and vertical scaling refers to the ability to add new applications, services, hardware and other computing resources on top of existing edge computing infrastructure to accommodate new requirements and vendors.

4. Security and Privacy : IoT devices at the edge of the network produce data and offload it to the edge servers for further processing. Varying levels of security needed at different times, example, in smart home scenario, only when a critical event is identified, the information will be shared by other areas in the vicinity to prioritize safety over privacy. Ownership of data at the edge in the presence of several service providers presents another challenge, for example, several individuals or a group could be managed by one service provider at the edge and this activity must not be visible to other service provider at the edge that manages other group. This is required to keep

business logic private for services in what is a very competitive market these days.

2.3.3 Related Work

The aforementioned challenge of service orchestration for large scale IoT has been addressed in the literature from different angles. Orchestration is still in a very early stage of research in the edge computing domain with several works limited to just proposing an architecture and projecting its benefits rather than a prototype-driven experimental evaluation in a simulated or a real test bed to give an idea of its practical feasibility. This subsection reviews the research proposals and industry solutions that particularly focus on solving problems related to applying edge computing to large scale IoT scenarios in the context of service orchestration.

Research proposals overview

Jiang et al. explored the applicability of the cloud computing orchestration model to edge computing for supporting IoT applications [54]. The authors envision an orchestration framework that supports affinity-aware offloading and enables an application to be partitioned into segments and offloaded onto different fog nodes. The discussion refers to the three layer cloud computing orchestration in which the top service layer provides abstractions for cloud applications to access the computing services. The central controller in the control layer allocates the resources in the lower physical resource layer to the cloud applications in the upper service layer. The authors identify challenges regarding scalability of the control layer and support for affinity aware offloading, and propose design recommendations for the same. For achieving scalability, the authors focus on the interoperability aspect and propose a distributed orchestration architecture where edge nodes communicate with each other in per to peer fashion. The flow of operation of the edge infrastructure is driven from top to bottom with the northbound interface provides the abstraction for the application to access resources similar to cloud computing.

Yigitoglu et al. proposed a system called ISYMPHONY [103] to better scale the IoT service provisioning for on-demand and real-time requests in a large scale IoT environment. The solution is particularly focused on mobile clients such as vehicles which submit on-demand requests to the edge servers, for example, querying the condition of the road ahead of them. The main idea behind their proposed distributed orchestration architecture is the intelligent partitioning of the computations corresponding to the clients requests into the tasks which could be executed by the edge servers and the tasks which could be executed by other mobile clients in the vicinity. Their results showed that this coordinated processing approach can lead up to two orders of magnitude of reduction in the edge server load and also provide better accuracy.

Santoro et al. proposed an edge computing architecture named Foggy [83] for workload orchestration in a Smart City scenario. The architecture utilizes Kuber-

netes as the underlying container orchestration platform. Their experiments show that the system is able to perform workload orchestration for on demand client requests. It is able to do this when clients submit their requests to the manager node in the edge cluster.

Flores et al. propose a large scale offloading architecture in [42] to address the challenge of handling multiple concurrent client requests. However, it is studied and experimented purely from a cloud perspective. The authors propose an offloading architecture to analyze the capacity of several cloud servers to handle multiple concurrent client requests in a large scale IoT scenario. Each client request corresponded to offloading a computation on the cloud server. A component called Autoscaler was developed to receive the simulated client requests and then distribute them to several cloud servers in a round robin fashion. The architecture scales horizontally with increasing traffic and is suitable for multi tenancy. From the experiments and results they observed that the Autoscaler component added an overhead of about 150 milliseconds to the response time of a client request.

Some solutions utilize the crowd sourcing approach to detect and then generate fast response to critical events such as disasters, emergency and safety cases. For instance, Sapienza et al. proposed a MEC based crowd sourcing approach to detect critical events in a Smart city domain [84]. The edge servers co-located on the baste stations run ML based applications continuously on the data received from the surveillance systems and air quality monitoring sensors installed in the city to detect an abnormal activity. The base stations then collaborate to send alerts and safety evacuation information details to the citizens in the respective coverage areas. Through the concept of MEC, a large geographical area is served in a quick span of time. Moreover, MEC particularly benefits in this case as the edge servers are co-located with base stations and have knowledge about users position and mobility that helps in planning a safe evacuation routes.

In a large scale IoT, distributed IoT analytics is a key. Hsieh et. al realized this gap of distributedly deploying and managing IoT analytics on heterogeneous devices from data center servers, edge cloud workstations, and embedded IoT devices (such as smart sensors). The authors proposed a series of works on a managed edge computing platform using Docker, Kubernets and tensorflow [50, 96] to run distributed data analytics on edge nodes by launching containers. It allows IoT devices to be remotely monitored, diagnosed, and upgraded while IoT analytics can be split and distributedly deployed on multiple IoT devices. The authors added support for event-triggered container deployment using the MQTT (Message Queuing Telemetry Transport) pub/sub protocol. The authors present only a preliminary evaluation plan to measure the deployment time with varied sizes of applications and instance of container deployment. The authors also plan to measure fast application deployment via distributed analytics by cutting the applications into several chunks and then deploying the individual chunks on different nodes.

Research efforts have also proposed design of a middleware that facilitates integration, communication of several applications in a large scale IoT domain like Smart City. For instance, Mohamed et. al proposed a service oriented middleware

for smart city applications in fog computing [69]. Rausch proposed a message oriented middleware for edge computing applications [80]. However, since a middleware deals with infrastructure development for facilitating interoperability, it is not in the scope of this thesis as the focus lies mainly on the orchestration framework.

Industry Solutions for IoT edge computing

Kubeedge [14, 101] is an open source platform which provides an edge computing infrastructure based on Kubernetes [15]. It provides a unified runtime environment for the applications to be deployed and access resources across the edge and the cloud. The edge infrastructure proposed by Kubeedge consists of a cloud part and an edge part. The cloud part features a Kubernetes controller plugin that allows remote deployment and management of applications on the edge nodes. A communication protocol and metadata sync service is implemented to facilitate bi-directional data synchronization between the cloud and the edge. It also enables the edge nodes to operate in offline mode or adapt the size and rate of data transmission to cloud in the case of a poor network connectivity between the edge and the cloud.

Azure IoT Edge by Microsoft [2, 3] is another open source solution which enables packaging the business logic of the applications into containers and deploying them on the edge devices. The application deployment and management is done through a device twin of the target edge device that resides in the cloud. The edge devices runs the Azure IoT Edge runtime which manages the containerized applications that runs on these edge devices after they are deployed from the cloud.

Amazon IoT Greengrass extends the Amazon Web Services (AWS) to the edge [1]. It consists of the AWS IoT Greengrass Core which provides cloud based management of applications that runs on the edge devices. The Greengrass Core itself runs on the edge device and enables the local execution of the Lambda functions in response to certain events as programmed by the developer. Moreover, it facilitates multiple edge devices to interact with each other securely and exchange messages without relying on the cloud.

Solutions by the standardization bodies

The OpenFog Reference Architecture (OpenFog RA) [46] is the IEEE standard for fog computing [10]. It is a joint effort of several technical working groups from the industry and the academia such as Microsoft, Intel, Princeton University, ARM, etc. under the umbrella of the OpenFog Consortium (now merged with Industrial Internet Consortium) [12] to contribute towards an open architectural framework concerning the design, development and operational aspects of fog computing. It is intended to help multiple stakeholders such as software developers, end users, network operators, investors in the fog/edge computing infrastructure, hardware

developers, etc., in analyzing and understanding the best practices for an edge computing infrastructure that tackles both technical and business challenges.

The OepnFog RA lays the foundation for an edge/fog computing solution for the IoT by defining eight core design principles called as the *pillars*. These pillars are Security, Scalability, Autonomy, Openness, Programmability, RAS (Reliability, Availability and Serviceability), Agility and Hierarchy [46]. Aligning the design choices with some of the architectural guidelines mentioned in the OpenFog RA provides a level of credibility to the work carried out in this thesis. Incorporating both the end users and service providers perspective in the design of an edge computing architecture is one of the objectives of this thesis. Synchronizing the ideas proposed in this thesis with the standardization body that aims to not only benefit the end users but also the service providers helps to meet this objective.

2.4 Summary and Gap Analysis

Table 2.1 provides an overview of the key findings from the literature survey, their associated significance in the context of a large scale IoT scenario such as a Smart City and the potential solutions proposed in this thesis.

The first main finding from the literature survey on edge computing solutions for IoT is that the existing proposals employ a master-slave model for edge architecture and service orchestration. The edge infrastructure possesses a fixed hierarchy and consists of one or more masters that perform management and control operations and a set of slaves or worker nodes that are dedicated to executing the workloads. The control plane consisting of the aforementioned managers is often located in the cloud and shares a communication link with the worker nodes at the edge. In some cases, the master resides on the edge cluster itself. The control plane remotely dispatches, deploys, orchestrates and monitors the workloads that executes on the slave nodes on the edge. The workload orchestration process in such an architecture involves the following steps.

1. Supplying a configuration file at the control plane containing the information regarding the applications to run at the edge, desired number of replicas, opening of network ports, etc.

2. The manager node in the control plane parses this configuration, determines the ideal worker node primarily based on the computing resource requirements to run the requested workloads.

3. Post the determination of the target edge node(s), the workloads are deployed on them. Continuous updates are sent from worker nodes to the manager node about status of the application, availability of the computing resources such as CPU, RAM, etc.

Gaps in the literature	Impact in large scale IoT	Potential solution
Cloud driven edge architecture and master-slave orchestration model	Renders the edge servers passive and oblivious to the interactions of the end users with the edge that govern issuing of service requests or contribution of computing resources	Edge-driven orchestration that enables the edge nodes to react to the events and interactions at the edge of the network
Absence of transparency at the edge	Restricts ubiquitous sensing and access of computing resources by the end users, thereby degrading their ease of use and subsequently the profits of the concerned stakeholders	Flat hierarchy at the edge in the form of edge nodes with equal privileges to orchestrate client requests
Runs applications indefinitely on the edge servers, only suitable for scenarios that offer a fixed service at the edge, for example, IoT analytics	Renders existing solutions infeasible for handling multiple different and simultaneous client requests that require ephemeral workloads to be executed on the edge servers and demand customized service	Dedicated service management features for on-demand client requests with the ability to serve multiple clients simultaneously
Lack of incorporating the tight coupling between the workloads that execute on the edge servers and their input data source	Renders existing solutions infeasible for evolving IoT trends such as collaborative processing in connected vehicles domain	Enable ubiquitous sensing and pooling of the IoT and computing resources contributed by end users at the edge and utilize them in orchestrating client requests

Table 2.1: Overview of the Gap Analysis

The term ***top-down*** is used to refer this flow of operation and design ideology in the existing edge architectures, analogous to the flow of information from the higher end of the hierarchy, i.e., manager node to the lower end, i.e., the worker nodes.

The second main finding is that the existing solutions lack to address some key aspects of the interactions of the edge devices and end users with the edge computing infrastructure particularly in the context of IoT. The first of these include the on-demand service requests that require specific ephemeral applications to be deployed on the edge. This is in contrast to the workloads that require continuous up time on the edge servers such as data analytics applications which continuously processes the incoming sensor data streams or the web applications which are accessed by several clients simultaneously.

Significance of on-demand service requests in a large scale IoT scenario

A large scale and complex IoT application domain such as the Smart City involves the provisioning of several applications through the ICT infrastructure of the city with a target to encompass the varied service requirements of the large urban population. Consequently, in order to provide these applications at the edge of the network for a better QoS, the edge computing solutions must deal with deployment and management of a plethora of heterogeneous applications and end user devices. A typical practice followed in edge computing that reflects in the literature is to keep the applications running on the edge servers indefinitely. However, considering the limited compute and storage resources at the edge, this approach is infeasible. A more effective approach would be to exploit the lightweight virtualization techniques that offer a sand box like execution environment enabling the applications to be deployed and terminated on-demand [78]. Connected vehicles is identified as a use case that can benefit from such an approach. A typical example of an on-demand service request in this scenario is a vehicle contacting the nearby roadside infrastructure embedded with edge computing resources to query about the condition of the road or traffic situation in the neighborhood for an improved driving experience [103].

Collaborative processing: Tight coupling between the service requests and the corresponding IoT resource

The second key aspect of the end user's interactions with the edge computing infrastructure concerns with the nature of service requests and the evolving trend in the IoT. The workloads corresponding to the on-demand client requests under consideration are tightly coupled with the input data source which they consume in order to generate the desired end result. This input data is referred to as the *IoT resource* corresponding to the service request. To render further relevance and clarity to the aforementioned dependence between the IoT resource and the service request, a practical IoT use case applicable in this context is discussed. Recent studies have

highlighted the idea of simultaneous and dual presence of data producers and consumers in an IoT ecosystem [88]. When this phenomena is combined with the sufficiently powerful computing capabilities that the portable IoT devices possess nowadays, the outcome is the existence of rich opportunities for collaborative data sharing and processing.

As discussed in Section 2.3.3, some of the works in the literature have proposed techniques to incorporate and address this aspect in designing edge computing solutions for the IoT in order to further augment the benefits offered by edge computing via leveraging end devices as additional compute, storage and data resources. One such research effort proposes a technique to augment a driver's view by collecting and combining the HD maps, snapshots and sensor data offloaded by other vehicles in the vicinity to render a full view of the surroundings and updates on incidents to the driver [108, 109]. Hence, the IoT resource in this case, i.e., the data contributed by other vehicles is a key ingredient for the applications to generate useful information.

Limitations of the master-slave edge computing architecture

Due to the master slave topology, the state of the art and the existing approaches in edge computing renders the edge oblivious to the events and the interactions among the IoT devices and end users in its immediate vicinity, of which, it lies at the receiving end of. As explained above, it is these events and interactions which forms the channel for the end users to offload data and computations on the edge or issue on-demand service requests, especially in the context of IoT and pervasive computing. Hence, the state of the art would require the cloud to regularly fetch information regarding the interactions at the edge concerning these on-demand requests. The round trip between the edge and the cloud involved in performing such updates would result in a significantly high latency. It also becomes infeasible for the cloud from the latency perspective to monitor the individual workloads executing at the edge and act on them in response to any updates or events at the edge of the network. Besides the high latency, this approach also suffers in the event of an intermittent or total absence of network connectivity between the edge and the cloud, thereby disrupting the transmission of updates and subsequently the orchestration process.

Thus, this makes the existing approaches static and only suitable for scenarios that offer fixed services at the edge, for example, data analytics applications that continuously monitor sensor data streams. To tackle this issue, IFTTT (If This Than That) or rule based approaches have been proposed in the literature to enable the edge to react and deploy workload in response to events at the edge of the network such as joining of a new IoT device, critical threshold overflow detected by a CEP (Complex Event Processing) engine that fuses multiple sensor streams, etc. However, such approaches only solves a tiny and rather trivial part of the problem in this context.

Absence of transparency at edge degrades ease of use

One of the main objectives of the edge computing paradigm is allowing the involvement of the edge servers in helping the end users and IoT devices with computing, storage, and networking capabilities [78]. The effectiveness of the edge computing approach will ultimately depend on the manner in which the edge and the IoT devices interact [78]. However, the presence of the worker nodes in the existing approaches restricts the ubiquitous sensing and interaction of mobile clients with the edge servers, thus, limiting the effectiveness of the edge computing solution in place. This is due to the inability of the worker edge nodes to participate in the sensing and orchestration process as they are dedicated towards executing the workloads passively. As a result, the end users are restricted to issue the service requests or offload IoT resources only to a specific set of edge nodes, thereby requiring them to be aware of the topology of the edge infrastructure. This limitation deteriorates the ease of use and the QoE (Quality of Experience) of the edge computing infrastructure as perceived by the end users which ultimately affects the profitability of the edge service provider.

Need for an edge-driven distributed orchestration

Existing approaches lack to address the orchestration of on-demand client requests and incorporate the above discussed key aspects of the end user's interactions with the edge infrastructure in a large scale IoT scenario. As discussed above, the interactions of the clients with the edge infrastructure are not only limited to issuing service requests but also offloading data or contributing other computing resources to the edge servers. In such a scenario, the challenges faced by the orchestration framework of the edge computing infrastructure are manifold.

1. The edge nodes, i.e., the edge servers offering the computing and storage services, needs to be intelligent enough to differentiate the interactions at the edge of the network into client requests and offload of data or computing resources.

2. Dynamic orchestration and computation offloading based on real time availability of the IoT resources as extracted from the ubiquitous sensing of events and interactions of end user devices at the edge of the network. The availability of the IoT resources on the edge is subjective to the mobility of the end users.

3. Ensure a sufficient level of QoS to the end users by handling the multiple concurrent client requests and traffic bursts during peak loads with minimal overhead.

4. Maximize the profitability of the edge computing service provider by enhancing the ease of use of the edge computing infrastructure for the end users.

An edge-driven distributed orchestration architecture possesses the potential to address all the above highlighted challenges. The edge nodes are in in the vicinity of the events at the edge of the network and also to the IoT devices and end users that interact with it. Hence, it is both logical and efficient for the edge nodes to drive the workload orchestration considering their capability of utilizing the local knowledge available in the form of IoT resources and context related information to aid in the orchestration process. The distributed operation of the orchestrator across the constituent edge nodes in the infrastructure will enable the geographically distributed edge nodes to take part in the orchestration and decision making process. Moreover, by proposing such an architecture for edge computing, this thesis also aims to significantly reduce the dependence of the edge on and the involvement with the cloud. This enables the edge to operate autonomously and unaffected by network outages and disruptions. However, the dual responsibility on the edge nodes to orchestrate and execute the workloads is bound to induce an overhead which may affect the QoS as perceived by the end users. This thesis aims to study this trade off and collect insights. The first impression is that the benefits offered by this approach outweigh the overhead induced by it.

This design ideology is supported by referring to the edge computing architectural guidelines as defined in the standards. It can be particularly related with the ***Autonomy pillar***, i.e., one of the core design principles for an edge computing architecture listed by the OpenFog Reference Architecture (OpenFog RA). The autonomy pillar particularly refers to the features enabling autonomy at the edge of the network through collective intelligence from end user devices, information or data exchanges via peer to peer connections and discourages centralized decision making model. This is in alignment with the proposed idea of distributed orchestration via collaboration of edge nodes in a peer to peer fashion and the concept of having a flat hierarchy at the edge enabling all the edge nodes to take part in the decision making process.

Chapter 3

EDIRO: Edge-driven IoT resource-aware Orchestration Framework

This chapter describes the design and implementation of EDIRO : the edge-driven IoT resource-aware orchestration framework proposed in this thesis. Section 3.1 describes the design of EDIRO including the justification of the approaches and associated challenges foreseen at the time of system design. Section 3.2 describes the implementation of the software prototype of EDIRO. Section 3.3 provides the reasoning for the choice of tools used for software prototyping of EDIRO.

3.1 System Design

Following from the gap analysis presented in Section 2.4, this thesis proposes an edge-driven IoT resource-aware orchestration framework called EDIRO which facilitates collaborative processing in large scale IoT. EDIRO has the following key design goals.

1. An autonomous edge infrastructure that processes the events at the edge of the network and drives the service orchestration through the collaboration of the constituent edge nodes. This design goal is partially aligned with the ***Autonomy*** pillar of the OpenFog Reference Architecture (OpenFog RA) which is the IEEE standard for fog computing [10]. The aspect of the Autonomy pillar that this specific design goal relates to is facilitating autonomy at the network edge that enables the intelligence from peer data and local devices to carry out the desired tasks at the most logical location [46].

2. Offer a flat hierarchy or transparency at the edge to the end users, enabling them to interact with the edge infrastructure ubiquitously.

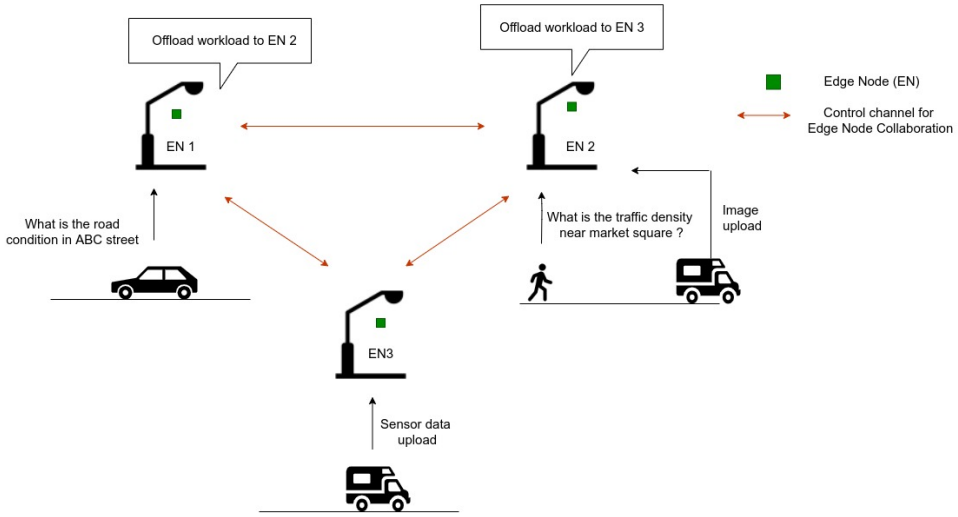


Figure 3.1: Overview of EDIRO in a connected vehicles scenario

3. Support collaborative processing use cases by performing IoT resource-aware offloading in which effective resource utilization takes priority over low latency.

4. Demonstrate the ability to service a large number of multiple simultaneous on-demand client requests and process traffic bursts resulting from offload of IoT resources such as sensor data, images, etc. in a large scale IoT scenario.

3.1.1 Overview

Figure 3.1 presents an overview of the application EDIRO to a connected vehicles use case in a Smart City scenario where an edge infrastructure consisting of multiple interconnected edge nodes run the EDIRO framework in a distributed fashion. These edge nodes are installed on the roadside infrastructure such as street lights. The orchestration of on-demand service requests by the end users which are vehicles in this case is considered particularly. An example of such an on-demand request is ”What is the condition of the road in sector 1 of the city?”. The edge nodes collaborate among themselves to orchestrate the on-demand service requests of the end users by utilizing the IoT resources collected at the edge nodes contributed by other clients. Besides servicing end user requests, EDIRO also facilitates their ubiquitous interactions with the edge infrastructure which includes sharing and offloading the IoT resources such as sensor data, images, etc. to the edge nodes [108, 109].

3.1.2 Transparency at the Edge

One of the main design goals of EDIRO is to offer transparency at the edge for the end users so that they are not required to be aware of the edge topology and be restricted to offload their computations, data or place service requests only at a specific edge node(s). This is essential to *maximizing the ease of use of the edge infrastructure* for the end users which brings subsequent benefits to the the edge service providers as well. The existing approaches deems the edge nodes passive under the role of *worker nodes* and renders them oblivious to the events and end user interactions. Consequently, the end users are often limited to offload their requests or workloads to the centralized entity or the master node(s) in the edge infrastructure. In the context of IoT, this is clearly ineffective considering the presence of mobile clients such as vehicles.

In an IoT environment, when the end users perceive the edge infrastructure as transparent or flat, they need not be obliged by the service provider to be aware of the edge topology. ***The benefits are twofold.***

1. Hiding the complexity of the edge infrastructure from the end users and enabling ubiquitous access of edge compute and storage resources can enhance the user experience significantly.

2. Abstracting the details about the topology of the edge infrastructure from the outside world results in an increased privacy and security of the edge infrastructure. For example, a publicly known set of master node(s) in an edge infrastructure can be potential targets of disruption and their security could be compromised as well. This is highly undesirable for the edge service providers from a business perspective and also for the end users.

The design of EDIRO concerns with incorporating both the ***end user perspective*** and the ***edge infrastructure perspective*** of rendering transparency at the edge. In alignment with the goals of EDIRO, both the perspectives must be considered and incorporated in the system design. In order to render transparency at the edge from end user perspective, each edge node should be able to receive client requests and IoT resource offloads. This suffices the requirement of the end users enabling them to offload their service requests or resources to any edge node(s).

However, rendering transparency at the edge from the edge infrastructure perspective holds much greater importance and forms the centre of the discussion. Since the events at the edge of the network play an important part in the orchestration of the workloads, enabling each edge node to actively receive and process such events would require them to be also involved in the orchestration process. Hence, the need for a distributed orchestration which is driven by the edge.

3.1.3 Distributed Orchestration

The rationale behind the need for distributed orchestration is that the various events at the edge of the network impact the orchestration of service requests. Since the

end user interactions are spread across multiple edge nodes in the edge infrastructure, the edge nodes must collaborate to orchestrate the workloads. The essence of distributed orchestration lies in achieving a decentralized decision making for workload orchestration. Since, practically any of the edge nodes can be at the receiving end of the client requests, it is essential that each edge node must possess the ability to participate in the decision making process of workload orchestration. This is achieved by designing EDIRO such that it enables an individual edge node to work independently and benefit from the presence of other edge nodes if any.

In EDIRO, the orchestrator performs three main functions: ***event parsing***, ***service management*** and ***resource management***. These are highlighted in the system architecture shown in the Figure 3.5 and explained in detail below.

A. Event Parsing

Event parsing corresponds to the processing of events and end user interactions at the edge followed by their segregation into client requests and IoT resource offloads. Each edge node listens for these events continuously and processes them.

B. Service Management

Service management is the core of the orchestration process in EDIRO. It corresponds to the serving of on-demand client requests by determining and offloading the corresponding workloads to the target edge node followed by their execution. This procedure involves involves several sub-processes as explained below.

Computation Offloading

As explained earlier in Section 2.2.2, computation offloading is one of the main procedures within workload or service orchestration. In EDIRO, the edge servers perform computation offloading upon reception of an on-demand service request from the end users. This thesis focuses on the computation offloading that happens in the edge infrastructure. The discussion regarding the computation offloading from end users onto the edge nodes does not fall in the context of this thesis as it concerns with aspects such as detection of the ideal edge node to offload requests, the type of networking technology used to offload computations, etc. The aforementioned aspects often concerns with an embedded intelligence in the form of a software logic running on the end devices which enables them to accomplish such tasks. However, this thesis mainly concerns with developing an autonomous edge that can facilitate edge computing demands of the end devices ubiquitously and any form of embedded intelligence in the end device will only compliment the overall QoS perceived by it.

The computation offloading from the edge perspective refers to the offloading of an application to a specific edge node for its execution that serves a particular end user service request. The two important considerations concerned with the computation offloading procedure are ***What to offload?*** and ***Where to offload?***. In the

context of the design of EDIRO, the answer to the first question is determined by using an application orchestration reference model and the second by the proposed computation offloading strategy as explained below.

Application Orchestration Reference (AOR) Model

In a large scale IoT scenario such as a Smart City, a plethora of different applications need to be provisioned on the edge servers to cater to a large number of end users that differ in their service requirements, inbuilt technology, QoS agreements, etc. The approach of hosting applications indefinitely on the edge servers tends to become infeasible given the limited computing resources on the edge nodes. The on-demand execution and termination of the applications on the edge servers in response to the end users service requests is a more effective approach. Existing preliminary but promising research on lightweight virtualization technologies and recent studies on practical scenarios such as connected vehicles in which such on-demand requests already exists, supports this vision [36, 71, 103].

In considering the application of EDIRO to the target use case of connected vehicles in a Smart City, this thesis particularly deals with applications that have a dependence on an IoT resource such as sensor data, image, etc for their execution. The rationale behind the same is EDIRO's goal of supporting collaborative processing in the IoT domain. These IoT resources are the input for the applications that run on the edge servers. In such a scenario, the edge must not only possess the knowledge of the workloads that needs to be executed in response to the end user service requests but also the associated IoT resource(s). This knowledge can be acquired in two ways.

1. ***Imperative approach*** - By using a predefined model that defines the relational dependency of the end user service requests on the applications and the IoT resources. Such a model can provided by the edge service provider in an attempt to advertise the services on offer.

2. ***Declarative approach*** - By using machine learning and artificial intelligence techniques that starts with a training set initially and eventually predicts the associated applications and IoT resources required to serve the future end user service requests.

The declarative approach tends to go out of the scope of this thesis as the end goal is to not develop such an algorithm or predictive model. The aim of this thesis is to develop an edge orchestration framework that can work with several of such dependency models. Therefore, an imperative approach is chosen that utilizes a static relational dependency model. Although not a standardized term but this is referred to as the Application Orchestration Reference (AOR) model throughout this work. The AOR model defines the relation of the end user service requests with the workloads that need to be run on the edge servers and the IoT resources they require. For the evaluation of EDIRO, a one to one mapping based on an

IFTTT (If This Than That) logic is considered , i.e., each end user service request maps to exactly one unique application and one unique IoT resource.

IoT Resource-aware Edge Offloading

This is the underlying computation offloading strategy that is employed in EDIRO to determine the answer to the question concerning ***where to offload?*** in the workload orchestration process. Typically, computing resource-aware offloading techniques are utilized in the existing edge computing solutions in which the edge node with sufficient computing resources (CPU, RAM, etc) becomes the default choice to run the workload. This thesis proposes IoT resource-aware edge offloading for collaborative processing use cases in IoT where the availability of an IoT resource with an edge node takes higher priority than the computing resources on it. With this idea, this thesis aims to extend the existing computing resource aware orchestration platforms and further make them IoT resource-aware, thereby rendering them feasible for related IoT scenarios. The offloading strategy is based on data locality aware edge offloading that aims to place applications on the edge in a manner which exploits the data locality [36]. With this technique, applications packaged using lightweight virtualization technologies are offloaded from the cloud to run on the edge.

In relation to EDIRO's application to the connected vehicles use case, the availability of an IoT resource is dynamic and spread throughout the edge cluster[1] considering the ubiquitous access of the edge infrastructure by the end users. Hence, the edge nodes must coordinate among themselves and share the information regarding the availability of the IoT resources on the edge cluster in order to enable the proposed IoT resource-aware edge offloading strategy. This process is termed as resource management which is yet another important feature of the orchestrator in EDIRO and is explained in detail below.

C. Resource Management

One of the main design goals of EDIRO is to enable each edge node to participate in equal capacity and drive the service orchestration process by acting on the end user service requests received by it. In order to serve these requests, the edge node must offload the corresponding application to the edge node on the edge cluster that possesses the associated IoT resource. However, as a ***side-effect of rendering transparency at the edge***, the availability of the IoT resources in the edge cluster turns dynamic and is spread across multiple constituent edge nodes. Thus, the information regarding availability of a particular IoT resource may not be available with an edge node resulting in it being unable to serve a client's service request. This behavior may further deteriorate with the increasing number of subsequent client requests.

[1]The term *edge cluster* is used throughout this thesis to collectively refer to the group of interconnected edge nodes or edge servers that form the edge computing infrastructure.

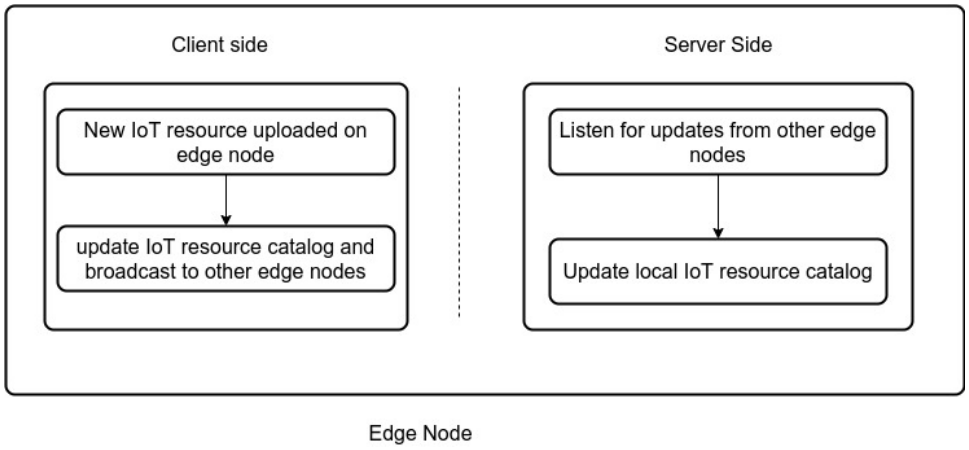


Figure 3.2: IoT Resource Management

To solve the above problem, a resource management feature is provided in EDIRO which runs as a distributed application on the edge nodes in the edge cluster. It enables each edge node to know the availability of all the IoT resources offloaded on the edge cluster and their locations at all times via synchronized event-driven updates. The event here refers to the event of an IoT resource being offloaded on the edge infrastructure. A centralized resource management scheme isn't feasible considering the non uniform and dynamic distribution of the IoT resources among the edge nodes resulting from the ubiquitous utilization of the edge infrastructure by the end users. Thus, a technique is proposed with which the edge nodes in the cluster collaborate to maintain a consistent state of the IoT resource availability information throughout the edge cluster on each edge node in a distributed fashion.

The design of the distributed IoT resource management technique in EDIRO is as follows. The event parser module is the first point of processing of the end user interactions with an edge node. Among these interactions, the ones that are identified as an IoT resource offload are forwarded to the resource management module. Each edge node maintains a local record to store the global IoT resource availability information. The term global is used to highlight that this record captures details about the IoT resources available on all edge nodes in the edge infrastructure. Thus, each edge node stores the metadata about the IoT resources available in the entire edge cluster. After an IoT resource is offloaded by an end user on an edge node, the local record of that edge node is updated with a new entry corresponding to this new IoT resource. An update request in form of a control signal is then broadcasted to all the other edge nodes in the edge cluster requesting them to update their respective local records with this new entry. In this update request, only the metadata about the IoT resource is transmitted and not the data itself to maintain a low communication and bandwidth usage overhead.

A consistent state after every new IoT resource offload on the edge cluster is said to have reached when all the edge nodes have identical contents in their respective local IoT resource catalog following the update request cycle. This is achieved when the other edge nodes acknowledge the change request by the host edge node. In between this there might be other new IoT resource arriving but the term consistent state is used to refer the successful update of an IoT resource by all the edge nodes in order to ensure that a client request depending on this IoT resource for its service will be surely served regardless of which edge node in the edge cluster receives it. Since this is the bottleneck for a client request to get served, it also serves as a metric to evaluate the system performance. Because, otherwise the client request may have to be offloaded to the cloud or may have to wait for a certain predefined time period hard coded in the system as indicated in the edge processing pipeline in the Figure 3.3.

Since all the edge nodes running the EDIRO framework are identical, they behave both as a client and as a server simultaneously. The client behavior of an edge node corresponds to it announcing the availability of a new IoT resource to the entire edge cluster while the server behavior of the same corresponds to it being actively listening for update requests from other edge nodes. Figure 3.2 captures this dual client-server behavior.

An alternate approach is to avoid regular broadcasts after every new IoT resource offload and generate a query broadcast only when a certain IoT resource is needed at a certain instant of time. However, considering the distribution of the end user service requests and IoT resource offloads in large scale IoT scenario, this approach would tend to overlap with the proposed resource management technique. Moreover, maintaining a consistent state on the edge cluster renders each edge node a global view of the edge infrastructure at all times. This is beneficial in scenarios where if one of the edge node shuts down abruptly, the other edge nodes would get notified after multiple control signal exchanges withing the edge cluster and a consistent state can be reached eventually. However, this particular failure recovery mechanism is not implemented in EDIRO at the moment.

Service Management Design Principle

An important aspect of service orchestration in a large scale IoT scenario is accommodating parameters such as multi-tenancy, service level agreements (SLA), privacy concerns and the indigenous heterogeneity among the end users [88, 100, 54]. This requires a granular and uniquely distinguishable service management per user or a group of users, for their *on-demand* service requests so that specific controls, policies and updates can be applied while processing their respective offloaded computations or service requests [78, 106, 35].

Existing container orchestration platforms such as Docker Swarm and Kubernetes enables the users to perform updates to the in progress workloads by supplying a set of commands to the control plane entity on the edge. Examples of the parameters and the aspects of a workload that can be updated include increas-

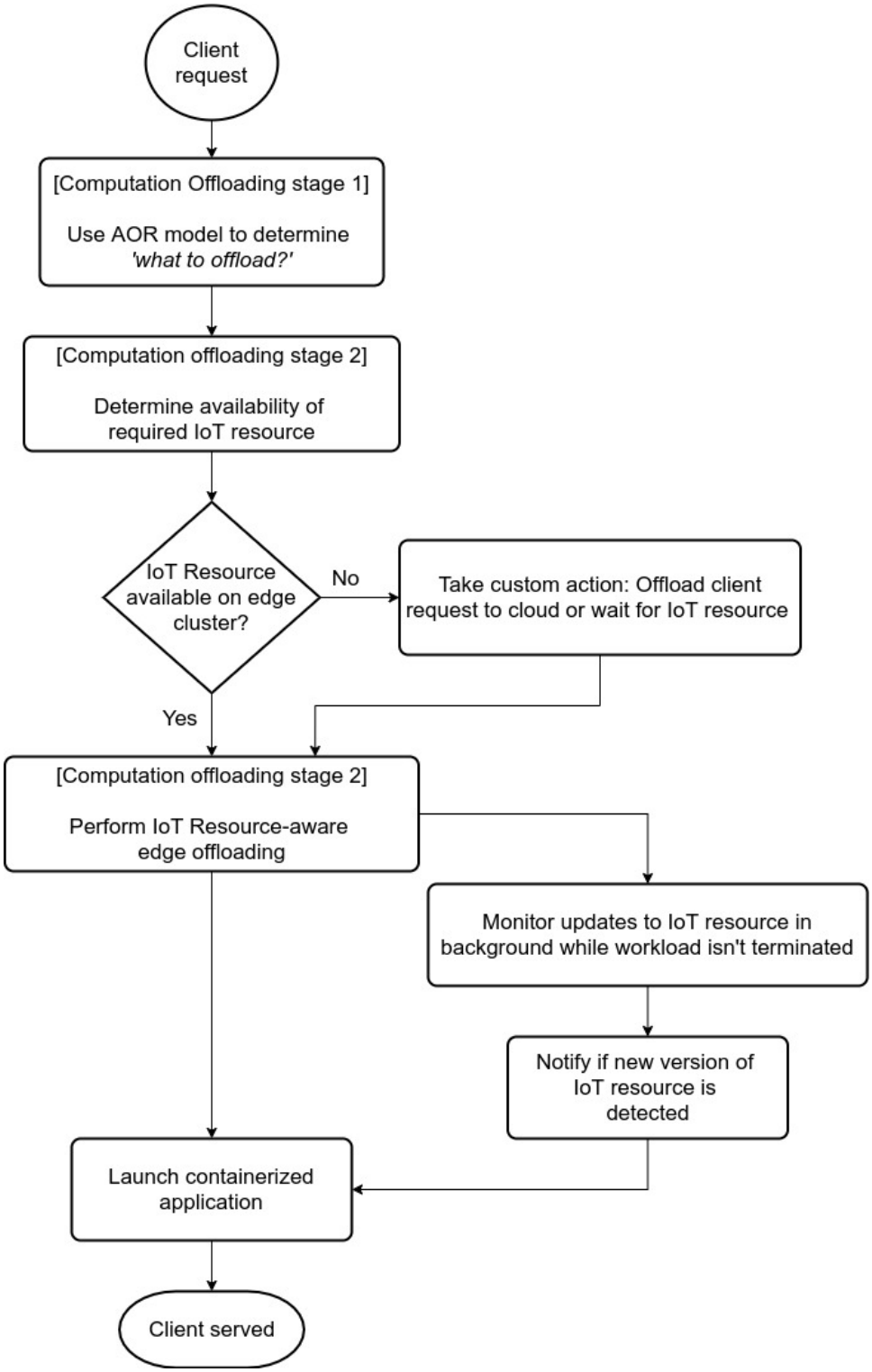


Figure 3.3: Edge Processing pipeline for servicing client requests

ing/decreasing the number of replicas of a running application instance, reschedule the workload to run on a different compute node, supplying a new base container image to be used by the application, limiting CPU and RAM usage, etc [4]. However, in a collaborative IoT environment, performing updates is often not a manual process but event driven. The interactions of the end users with the edge and the ongoing workloads on the edge forms the probable triggers for such updates. For example, availability of a newer version of an IoT resource, which is already being consumed by a workload, can be utilized to generate an end result that captures this latest update. Another example is where an intermediate or the end result produced by a workload on the edge may be utilized for another ongoing workload to reduce the computation time or improve the end result.

Handling such updates using a centralized orchestration model isn't feasible as the updates from across the edge cluster needs to be conveyed to the manager entity which then takes appropriate actions, thereby resulting in a high latency. To this, a per client dedicated service management feature is provided in EDIRO in the form of an edge processing pipeline. A unique edge processing pipeline is established on-demand in response to a client's service request and terminated post serving the client. Computation offloading marks the start of the pipeline followed by the execution of the workload using the underlying virtualization technology as shown in the Figure 3.3. In addition this, the pipeline also consists of a module dedicated for handling updates to the in-progress workloads in the form of utilization of new version of IoT resources. It utilizes the distributed resource management feature in EDIRO to continuously monitor for such updates which can then be performed as per a custom behavior defined. Hence, this dedicated service management feature per client is the key to the effective utilization of the resources and handling updates in a distributed orchestration framework such as EDIRO.

3.1.4 System Architecture

Figure 3.4 compares EDIRO with the state of the art in order to exhibit the benefits the proposed approach provides [26]. ***By proposing EDIRO, this thesis approaches the design and operation of an edge computing infrastructure in a bottom-up fashion as opposed to the typical top-down strategy in the existing approaches which ignores the interactions of the end users with the edge infrastructure and the related context information [26]***.

As shown in the Figure 3.5, EDIRO comprises of several individual modules which collectively provide its three main functionalities, i.e., resource management, service management and event parsing. EDIRO is designed to run on a single edge node and the distributed orchestration is facilitated when multiple of such similar edge nodes exist in an interconnected network. EDIRO is self sufficient for a single edge node to serve client requests considering IoT resources are available on the very same node. However, the edge infrastructure benefits from the presence of multiple edge nodes that run the same EDIRO framework. In such a scenario, each edge node runs an identical instance of EDIRO, thereby

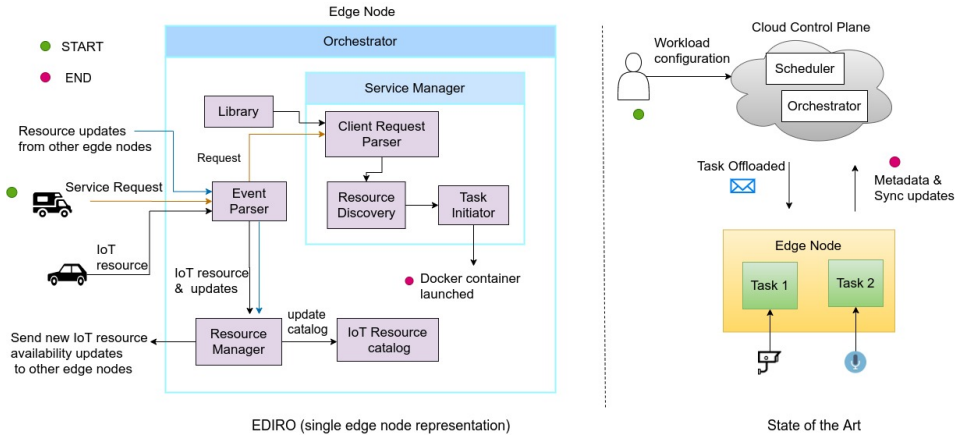


Figure 3.4: EDIRO vs State of the Art [26]

possessing an ability to orchestrate workloads in a capacity that is equal to any other edge node in the edge infrastructure. ***Hence, in this manner, an edge infrastructure consisting of several dissimilar edge nodes from hardware specification perspective, but running an instance of EDIRO, results in a flat hierarchy at the edge which eventually renders edge transparency to the end users.*** EDIRO uses containers as the virtualization technology for executing the workloads to serve the client requests for the reasons explained in Section 2.2.1. The description of the components that make up EDIRO is as follows.

- **Library** - Stores the information about the application packages corresponding to a particular client service request and the IoT resources corresponding to these applications in the form of key-value pairs. It is assumed that this information is made available by a knowledgeable party such as the service provider. For the practical evaluation of EDIRO, real containerized applications are used which are executed on the edge nodes in response to the on-demand client requests. Hence, The library stores the application packages by their corresponding container image labels.

- **Client Request Parser** - Parses the client service requests and fetches information about the corresponding application packages and the IoT resources required to serve the client request using the configuration library.

- **Event Parser** - The communication interface which enables the reception of client requests, IoT resource offloads and and sharing of the IoT resource availability updates to other edge nodes.

- **Resource Manager** - It performs the following tasks.

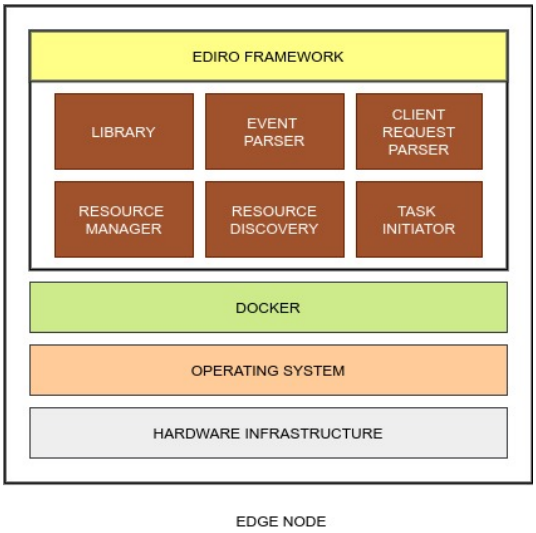


Figure 3.5: EDIRO System Architecture

1. Maintains a catalog of the IoT resource availability throughout the cluster in the edge node in form of the local state of the system.
2. Facilitate communication among the edge nodes to share the IoT resource availability metadata information each time an IoT resource is offloaded on the edge infrastructure. This includes the broadcast by an edge node to the edge cluster announcing the availability of a new IoT resource and the updating of its own local IoT resource catalog after hearing similar broadcast update requests from the other edge nodes.

- **Resource Discovery** - It performs the following tasks.

1. Determines the availability of the IoT resource needed by a workload and its location on the edge cluster by communicating with the resource manager.
2. Monitors any updates to the IoT resource in use and notifies the availability of a new version to take appropriate custom actions, for example, restarting task with the new version of IoT resource as the input.

- **Task Initiator** - It is responsible for the execution of the workloads using the underlying virtualization technology, i.e., container in this case. It translates the information provided by the Resource Discovery module into commands for the underlying container orchestration engine to launch the containerized tasks.

3.1.5 Workflow

Figure 3.4 shows the typical sequence of events involved in the distributed orchestration process in EDIRO. The process starts from an event or an interaction detected at the edge of the network on one of the edge nodes. The event parser module identifies the type of event as a client service request or an IoT resource offload and forwards it to the respective modules in EDIRO for subsequent handling. The client requests are processed in a edge processing pipeline as shown in the Figure 3.3 and the IoT resources are processed by the resource manager module as shown in the Figure 3.2. While a dedicated processing pipeline is established for each client request to facilitate dedicated service management, the IoT resources are processed together by a single resource management process that runs concurrently in the background. As explained earlier in Section 3.1.3, the edge processing pipeline relies on the latest IoT resource availability information rendered by the resource manager to orchestrate the client requests.

3.1.6 Challenges Involved

The first main challenge in developing EDIRO lies in implementing the distributed orchestration and maintaining a global and consistent state of events at the edge. As described earlier, the availability of the IoT resources on the edge infrastructure is captured and represented in the form of a system state by EDIRO. Each edge node maintains a local copy of this state that records the IoT resources available throughout the edge cluster which is eventually utilized for orchestrating the end user service requests by performing IoT resource-aware edge offloading. This challenge amplifies in the absence of a leader node(s) as the state on each edge node undergoes continuous modifications both by the host edge node in an event of a new IoT resource offloaded on it and also by an update request issued by other edge nodes announcing the arrival of a new IoT resource on them. In order to successfully serve the client requests, this state must be consistent throughout the edge cluster. A consistent state is said to have reached when the local state of each edge node is identical. However, the non uniform and dynamic distribution of the IoT resources on the edge cluster will lead to multiple simultaneous writes to the system state and the overhead that comes from the frequent communication among the edge nodes to synchronize these write operations. This overhead is expected to grow with increasing load on the system. Thus, it presents a challenge to ensure a sufficient QoS to the end users in this scenario.

The second main challenge is associated with the service management in EDIRO. As per the service management design principle described earlier, a unique edge processing pipeline is established on-demand for each client request. It comprises of multiple modules that collectively perform service orchestration. In a large scale scenario multiple clients are expected to access the edge servers simultaneously resulting in the creation of several of such pipelines and the increased communication among the edge nodes to serve these requests. This operation is

likely to consume computing resources (CPU, RAM, etc.) in an additive manner and may slow down the system eventually affecting the QoS perceived by the end users. This requires the edge infrastructure to possess a modular architecture and process several events in a concurrent and coordinated manner to avoid significant delays in servicing of client requests.

A key characteristic of EDIRO is the presence of several modules that work independently but simultaneously. For instance, the event parser and the resource management modules function in the background while the service management module orchestrates the client requests. Translating this design into a software framework presents a challenge to demonstrate a significant level of concurrency and interoperability of the constituent modules needed to achieve this functionality. At the same time, it needs to be ensured that the operational overhead does not affect the end user QoS drastically.

One of the goals of this work is to release EDIRO as open source to encourage collaboration with and experimentation by the research community. In this regard, special attention is paid to three main aspects of EDIRO as an open source project: ***ease of development***, ***extensibility*** and ***maintenance***. In an attempt to render the edge intelligent and autonomous, certain specific features have been embedded in EDIRO as part of its design in this work. However, several other features could be added to EDIRO in future to further enhance the edge intelligence. Hence, the implementation must consider a style of development which allows it to be flexible for such improvisations. While this thesis aims to integrate EDIRO into an existing container orchestration platform, it should also be able to run as standalone. This will facilitate the development and testing of EDIRO in isolation and not being limited by the design of the third party orchestration platform. Considering that the selection of specific software tools and techniques to achieve the above may impose certain limitations or impact the design goals of EDIRO, the challenge lies in making appropriate design choices for software prototyping that can help achieve an optimum trade-off.

3.2 Implementation

EDIRO is implemented as an executable software package. The source code for the same is written in the Go programming language [7] and lives as an open source project at [5]. This section is structured as follows. First, it introduces the building blocks that constitute the software prototype of EDIRO to provide a technical background for understanding the implementation of EDIRO. This is followed by describing the software architecture of EDIRO, the core components, their implementations, their interaction with each other and the method of solving design challenges identified earlier in Section 3.1.6. This also includes the reasoning for the choice of software prototyping tools were made to implement EDIRO.

3.2.1 Building Blocks

Goroutines

A goroutine is just another function that is specified to run concurrently to the calling function by prefixing it with the keyword 'go' in the program. Unlike threads in other programming languages, goroutines are scheduled by the Go runtime and not by the OS (operating system) such that multiple goroutines map to the same underlying OS thread [16]. Thus, goroutines are more lightweight than threads with an overhead in the range of a few KBs (Kilobytes) which facilitates running millions of goroutines on currently available computing systems [16].

Channels

Channels facilitate the execution synchronization of multiple goroutines in a program by allowing them to communicate with other go routines and exchange data [13, 16]. A channel is essentially a pipe to which a goroutine can pass data to and read from [16]. By default, channels are synchronous and blocking, i.e., once the sender writes data to a channel, it blocks until the receiver reads from the channel and vice versa. However, a buffered channel operates asynchronously, i.e., the sender or receiver need not wait for the other side to be ready until the channel is full. This is useful in the case when multiple inputs need to processed concurrently but separately from each other such as in the case of EDIRO where each client request is serviced separately in a unique edge processing pipeline established for it. Usage of a default channel otherwise will lead to other client requests waiting in the queue until the previous one is serviced.

gRPC

Remote Procedure Call (RPC) is a client-server based mechanism that enables a computer program to execute functions in a program residing on another computer in a different address space as if those functions resided locally [20]. Hence, this technique enables the design of distributed applications and forms an important part of the implementation of distributed orchestration in EDIRO, particularly, the resource management functionality. gRPC is utilized as the RPC framework for the implementation of distributed orchestration in EDIRO.

gRPC is an open source RPC framework that facilitates developing distributed services and applications in a variety of programming languages [9]. Developing a distributed application using gRPC involves defining the desired service, specifying the functions which can be remotely called, their input arguments and the output return types [9]. gRPC uses protocol buffers by default as the Interface Definition Language (IDL) for defining the desired service in the form of RPC methods and specifying the structured format for data that is exchanged in the process [9]. Protocol buffers are a language and platform neutral mechanism for serializing structured data [19]. The explanation of the detailed procedure for building

a distributed application using gRPC goes beyond the relevance of this chapter in the context of this thesis. While the detailed documentation can be accessed at [9], a brief overview is provided of the same below based on the procedure described in [9] in order to familiarize the reader with the general approach involved in this process. The below description mainly focuses on the aspects of the application that the developer needs to focus on and implement as the low-level details concerning the transmission of the bits on the wire are taken care of by gRPC.

Since gRPC is based on the client-server model, when building a distributed service using gRPC, the developer needs to first define the desired service by specifying the RPC methods that constitute it and the *request* and *response* message types they deal with. Request corresponds to the message that the client sends to the server while calling a RPC method and response is the message which the server responds to the client with after handling that particular RPC call. For example, a handshake service between two computers may define a RPC method named 'Hello' which takes in a request message by the name 'Hi' of the type string and emits a response by the message name 'Hi there' of the type string.

After the service definition, the client and the server side of the application needs to be developed in the choice of programming language. Development of the client side involves establishing a connection to the server, calling a RPC method, passing the data as an argument and optionally a custom action post receiving a response from the server. In order to facilitate this, gRPC provides the required APIs to and the client interfaces that contains the signature of the RPC defined in the previous step. These can be generated by compiling the service definition file in the choice of programming language. However, it's up to the developer to implement the RPC method from the provided signatures. The same approach holds good for the development of the server side as well where the developer needs to implement the core logic for handling of the client requests by the server which corresponds to the main part of the service.

3.2.2 Software Architecture

A modular design approach has been followed in the development of the software prototype of EDIRO. EDIRO is implemented as an executable software package that is composed of several modules that offer a specific functionality each. These modules and their respective functionality is mapped to the core components and their tasks as mentioned in the system architecture of EDIRO earlier in Section 3.1.4. These modules are implemented as individual library packages which combine together to compose a single software package for EDIRO.

EDIRO is designed and implemented as an event-driven architecture. The modules constituting EDIRO listen for events and act in response to deliver their respective functionality. The key events that trigger the system software of EDIRO are the end user interactions at the edge nodes that correspond to either offload of IoT resources or arrival of on-demand client requests. As described in the workflow in Section 3.1.5, following these events, concurrent processing branches are

created for service management and IoT resource management corresponding to the type of event. Each stage of the edge processing pipeline and the resource management results in the creation of new events that propagates through the respective branch and further trigger the functioning of the subsequent modules in EDIRO. Thus, each module in EDIRO listens and acts on a specific type of event depending on its designated role. All the modules constituting EDIRO are initialized to run as goroutines at software startup. At this point, each module is ready to process any events generated in the system. These concurrently running modules synchronize their operations and interact with each other using channels. While the precise and detailed explanation of the functionality of each of the modules is documented as comments in the source code which lives at [5], an overview of the implementation methodology for the modules in EDIRO and their interaction with each other is provided below.

3.2.3 Implementation Methodology for Core Components

For the software development of the modules constituting EDIRO, a common design template is followed as shown in the Listing 3.1. A module has a designated input and output channel dedicated to it using which it receives the events and writes post processing data to exchange with other modules respectively. The type of data that is exchanged over these channels vary only slightly per module as per its functionality but it typically represents and includes the details regarding the client's service request, corresponding workload to launch and the IoT resource.

Upon the initialization at software startup, each of the modules run concurrently and waits for the arrival of events by blocking on their respective input channels in an infinite loop. This behavior is comparatively more resource efficient than polling as the goroutine is blocked until there is some data on the channel. Upon the arrival of data on the channel, the Go runtime schedules the respective module enabling it to process the event and write the result on its output channel. Although the processing loop shown in the Listing 3.1 is sequential, the core processing part is delegated to another goroutine to facilitate concurrent processing of multiple events and avoid long waiting queues of events. Another point to note is that both the input and output channels are buffered which means that the sender or receiver side need not wait for the other end to be ready to handle the data passed over the channel. This holds particular significance here as buffered channels enable the sender and receiver to operate asynchronously, thereby allowing each module in EDIRO to operate asynchronously and independent from each other.

```
1
2  func module(input_channel, output_channel) {
3    for {
4
5          /* Read input from channel in an infinite loop.
6             Blocks until 'event' arrives on channel
7          */
8          input := <- input_channel
9
10         {
11            /* delegating the event processing task
12               to a goroutine to facilitate concurrent
13               processing of multiple events
14               and avoid queuing
15            */
16
17            go func(process_input)
18
19         }
20
21        /* Write result to output channel.
22           This will be the 'event' for another
23           module. Buffered channel allows
24           non-blocking write operation
25        */
26          output_channel <- result
27
28       }
```

Listing 3.1: Template for module implementation in EDIRO

3.2.4 End user Interactions With Edge Nodes

EDIRO is designed to react to and process the interactions that the end users have with the edge infrastructure in real life IoT scenarios. These interactions are the on-demand service requests and IoT resource offloads. However, for experimental purposes in this thesis, end user interactions at the edge nodes are simulated by representing them in a JSON format in a file and supplying it as an external input to EDIRO during testing. The event parser module in EDIRO parses the contents of this file and passes them to the other modules for subsequent processing in an appropriate structured data format as explained earlier in Section 3.2.3.

The rationale behind opting for the simulation based approach is that this thesis focuses on the feasibility of an edge-driven orchestration architecture purely on the basis of its capability to orchestrate the workloads and IoT resources that arrive on the edge nodes. The mode of wireless networking which the end users use to offload the IoT resources or issue service requests on the edge nodes assumes secondary importance in this regard. Hence, supporting a specific mode of wireless

or wired connectivity means in EDIRO for the end users to interact with the edge nodes wouldn't add any specific significance to EDIRO at this point. Therefore, a REST interface is not implemented and neither is a semantic language or an API developed which allows the end users to interact with the edge nodes in a sophisticated manner, although this could be a future work which could serve as an add-on to the current design of EDIRO.

Two separate sets of unique files are utilized for each edge node during a test run. These files contain the information regarding the on-demand client requests received and the IoT resources offloaded on an edge node respectively. Both the on-demand client requests and the IoT resources are represented as strings in these files. The rationale behind representing the client requests as strings is as follows. In a practical scenario of the target use case, an on-demand client request will ultimately be a form of a control message or a signal embedding a message that represents the desired service request. Moreover, in the context of EDIRO, the client requests are only used as keys which are plugged into the AOR (Application Orchestration Reference) model to determine the corresponding workload to be deployed on the edge nodes.

The rationale behind representing the IoT resources as strings is as follows. An IoT resource such as an image or sensor data offloaded on the edge node in a practical scenario will ultimately reside on the local disk storage of that edge node with its reference being its location on the file system, which is a string. Although the action of IoT resource offloads is simulated but real IoT resources in the form of sensor data sets and images are used for conducting experiments and evaluating EDIRO in this thesis. Before the start of a test run, each edge node is populated with a set of IoT resources. Their respective locations on the disk storage of that edge node is mapped to a string and captured in the file that is supplied as an external input.

3.2.5 Leveraging Underlying Container Orchestration Engine

EDIRO uses container as the virtualization technology for executing the workloads to serve the client requests for reasons explained in Section 2.2.1. While EDIRO implements the orchestration part, the following features are needed in order to deploy the containerized applications across the edge cluster.

1. Control and manage the deployment of applications on edge nodes in the cluster.
2. Execute the containerized workloads.
3. Access the current status of workloads on the edge cluster.

Docker is utilized as the container platform purely for the evaluation purposes of EDIRO. In addition to providing the means to build and run containerized applications, Docker also provides container cluster management features via the *Swarm*

Mode. This mode enables to turn a cluster of physical host machines into a single large pool of computing resources on which the containerized applications can be deployed [22]. It provides the features to control and manage the deployment of applications across these hosts machines or edge nodes, as required in the case of EDIRO. Docker must be installed on each edge node. In order to use the swarm mode, all nodes must be added to the swarm. Generally the swarm mode has some managers and some worker nodes but considering the goal of this thesis of possessing a flat hierarchy, an all manager mode is chosen so that enables each edge node to offload and execute containerized applications anywhere on the edge cluster.

The output of the orchestration process is the determination of what workload runs where. In order to actually run the workload, or leverage any additional service from Docker, contact to the Docker engine needs to be established. Docker possesses a client-server architecture in which the Docker engine is the server that exposes a REST API for the client to specify the desired action to be performed, for example, launching a container. The system command are constructed based on the information derived from the orchestration process to launch the containerized workloads and query the information about ongoing workloads.

3.3 Rationale for Choice of Software Prototyping Tools

This thesis makes specific choices for the software development tools and technologies used to develop the software prototype of EDIRO. These choices were driven by the goals of this thesis and the design goals set for EDIRO in particular. Three key ones are discussed in this regard, namely the choice of programming language, the RPC framework and the underlying container orchestration platform.

3.3.1 The Go Programming Language

A significant level of concurrency is the key to facilitating handing of several events at large scale that needs to be processed separately from each other. The Go programming language also known as Golang, offers support for concurrency in the form of goroutines and channels. As explained earlier in Section 3.2.1, the lightweight and low overhead of goroutines enables the application to spin up several go routines as compared to threads. Channels allow safe communication among go routines by sharing memory by communicating and not the other way around which is the case with threads [8]. Choosing Go as the programming language for development of EDIRO facilitates extending Docker's container orchestration system (also implemented in Go) as a future work identified in this thesis. Go also provides APIs for benchmarking the application which can help identify the sections of program that contribute to resource contention or consume considerable amount of CPU eventually slowing down the entire application. However, this feature is not utilized as part of this work.

3.3.2 gRPC

The rationale behind opting for gRPC as the RPC framework for implementing the distributed orchestration in EDIRO is based on the following characteristic features of gRPC.

- A development environment that facilitates building a distributed application by defining a service with a set of methods makes for a structured application development process. It improves readability of the software manifolds and renders a significant clarity to the application unlike the case with using TCP sockets which is primarily based on sending raw data over the wire.
- Self manages the low level details of the distributed application such as sending the bits over the wire. Thus, the developer needs to only focus on designing and implementing the core logic of the application and not deal with the engineering aspects of the underlying networking and data transfer mechanisms.
- It allows the application to be extended in future by facilitating updating the service definition with adding more features that could be supported on both the client and the server side.
- Supports application development in several programming languages to a level where the client and server side could be implemented in different languages. This facilitates building a distributed application for use case involving dissimilar hardware infrastructure. For example, developing a distributed edge orchestration framework involving edge nodes owned by different vendors running different operating systems and runtime environments could potentially benefit from this.
- Last but not the least, gRPC is backed by an active community with strong online presence and a well maintained open source repository with good documentation.

3.3.3 Docker

The choice of the underlying container platform was based on the following criterion.

- It should offer container cluster and management features required for experimental evaluation of EDIRO.
- Must possess a simplified architecture of the inbuilt container orchestrator whose source code is easily accessible and extendable.

Docker stood out as the clear choice as compared to Kubernetes. Simplicity in container deployment is one of the main reasons why Docker is preferred over

Kubernetes. Container deployment and cluster management in Kubernetes involves several steps of configuration of the infrastructure, setting up volumes, networks, etc. The other main reason behind opting for Docker as the choice of underlying container platform is the easy access to 'Swarmkit' which is the open source project that houses the source code for Docker's inbuilt orchestrator. Although the only documentation available for it is a few short design documents, the source code overall does not appear as very difficult to follow.

Chapter 4

Measurements and Results

This chapter presents the description of the experiments conducted and the analysis of the results obtained for the experimental evaluation of EDIRO. Section 4.1 describes the experimental set up. Section 4.2 describes the measurement methodology and the results obtained for the overhead from distributed orchestration in EDIRO. Section 4.3 describes the measurement methodology and results obtained for the client request servicing time in EDIRO. Section 4.4 describes the measurement methodology and results obtained for the computing resource utilization of EDIRO on different hardware devices. Section 4.5 correlates the different results obtained and discusses the implications of the same in the context of the research questions defined in the thesis.

4.1 Experimental Setup

This section describes the four main components of the test setup for the experimental evaluation of EDIRO, namely the *edge infrastructure*, *workload applications*, *client requests* and the *IoT resources*.

Edge Infrastructure

The test setup comprises of a set of interconnected computing devices emulating edge nodes in an edge computing infrastructure. For the experiments, three different types of devices are used as listed below.

1. An Intel NUC (Intel Core i3-8109U CPU@3.6GHz, 8GB RAM) running Ubuntu 18.04.3 LTS with Docker community edition version 19.03.6 installed.

2. A laptop PC (HP Probook 440G4, Intel Core i5-7200U CPU@2.5GHz, 8GB RAM) running Ubuntu 18.04.4 LTS with Docker community edition version 19.03.6 installed. This will be referred to as just 'PC' henceforth for simplicity.

3. A Raspberry Pi 3B+ (BCM2837B0 quad-core ARMv8 CPU@1.4GHz, 1GB RAM) running Raspbian GNU/Linux 10 (buster) with Docker community edition version 19.03.6 installed.

Two different variants of the test setup are created to emulate two edge computing infrastructures that differ vastly in terms of the computing, networking and storage resources available on each of them. This facilitates evaluating EDIRO on two distant ranges and types of edge infrastructures. To accomplish this, the aforementioned set of computing devices are paired with each other accordingly to form a two node edge computing setup. An Intel NUC interconnected with the PC over a 1Gbps Ethernet link forms one setup and two Raspberry Pi devices interconnected over a 100Mbps Ethernet link forms another.

Each edge node is populated with the EDIRO software package and added to a cluster initialized using Docker Swarm. The swarm mode in Docker turns a cluster of multiple separate physical Docker host machines into a logical single large pool of compute nodes where containers can be deployed and migrated across. The control and core control and management responsibilities are fulfilled by dedicated *manager* node(s) while the *worker* nodes are only limited to executing the workloads. Aligning with the ideology of a flat hierarchy at the edge, each edge node is assigned a manager role in this cluster. Furthermore, while adding the edge nodes in the cluster, a specific label is assigned to each edge node which acts as a unique identifier for that node. A unique identifier for each edge node facilitates deploying containerized workloads on a specific edge node in the cluster.

Simulation of the input traffic : Client requests and IoT resources

EDIRO is designed for IoT edge computing scenarios where the clients offload their on-demand requests to the edge nodes requesting a service in return. This service corresponds to the result produced by a corresponding workload application that executes on an edge node. Furthermore, the workload application requires an IoT resource such as sensor data, image, etc., as an input for its execution which is sourced from the IoT resources offloaded by other end users on the edge nodes in their vicinity. This is one of the underlying methodologies to accomplish collaborative processing in an IoT domain. A typical example for this is a vehicle querying an edge node installed on a nearby street light for the driving conditions in a certain part of the city. To serve the vehicle's request, the edge infrastructure collectively processes the IoT resources ,i.e., images, HD maps and sensor data offloaded by other vehicles on the edge nodes in their vicinity [108]. This application scenario of collaborative processing for connected vehicles in a Smart city domain is also the target use case for this work.

For experimental purposes in this thesis, this input traffic to the edge nodes is simulated. This is accomplished by simulating the event of offloading client requests and IoT resources to the edge nodes. The rationale behind opting for the simulation based approach is that this thesis focuses on the feasibility of an edge-

driven orchestration architecture purely on the basis of its capability to orchestrate the workloads and IoT resources that arrive on the edge nodes. The mode of wireless networking which the end users leverage to offload the IoT resources or service requests on the edge nodes assumes secondary importance in this regard.

The client requests and the IoT resources are represented as strings and the event of offloading either of them on an edge node is represented by a JSON notation in a file as explained earlier in the Section 3.2.4. This file represents a uniform distribution of these events across the two edge nodes. Two separate sets of unique files are utilized for each edge node which embed the information of the on-demand client requests received and the IoT resources offloaded on an edge node respectively. Multiple of such sets of files are created to simulate the application of varying amounts of traffic at the edge nodes for different test runs. The rationale behind this form of input representation is as follows.

In a real life scenario involving actual vehicles on the road, the events of offloading service requests and IoT resources on the edge nodes by the end users can be characteristically described as the following.

- Client requests are typically of the form of a control message embedding the service request metadata [103]. Upon their reception by an edge node, they will be processed and a workload application will be deployed for execution in response that will produce an output that serves the particular client request.
- An IoT resource such as an image or sensor data offloaded by an end user on the edge node will ultimately reside on the local disk storage of that edge node.

Since, in the context of EDIRO, the client requests are only used as keys which are plugged into the AOR (Application Orchestration Reference) model to determine the corresponding workload to be deployed on the edge nodes, they are represented as a set of strings in the simulated input traffic. And, an IoT resource offloaded on an edge node can be referenced by its location on the disk storage, thus it is represented as a string too. However, although the action of offloading an IoT resource is simulated, actual IoT resources in the form of sensor data sets and images are used for conducting experiments and evaluating EDIRO in this thesis.

Traffic Index

The term *Traffic Index* is used to specify the amount and the type of traffic in the form of incoming client requests and IoT resources that is applied to the edge nodes running the EDIRO framework. Traffic Index is represented either by an ordered pair (A,B) or an ordered triplet (A,B,C) with the description of the notations as given below.

- 'A' denotes the number of client requests received by one edge node.

- 'B' denotes the number of IoT resources offloaded on one edge node.

- 'C' denotes the inter-arrival time between the incoming client requests on an edge node. *An inter-arrival time of zero seconds between the consecutive client requests signifies a bursty traffic* which is characterized by multiple simultaneous client requests issued towards the edge nodes. *A finite inter-arrival time between the consecutive client requests denotes a normal or regular traffic scenario*. For the experiments, this finite inter-arrival time is fixed to 3 seconds.

Workload Applications

In order to serve the multiple different client requests, as many number of different applications need to be executed on the edge nodes in response. As stated earlier in Section 3.1.3, an imperative approach has been followed to map each of the incoming client requests to a specific workload application that needs to be deployed on the edge nodes to serve that request. While making the selection of the aforementioned applications for this evaluation, the relevance of a logical mapping of the applications with the client requests is not considered as the aim of this experimental evaluation is to only show the feasibility to orchestrate a variety of applications and requests regardless of their logical significance to each other. A set of different applications are mapped to a set of different client requests in order to simulate an environment in which a set of unique end users request a specific workload to be deployed on the edge nodes.

The workloads are executed as containerized applications on the edge nodes. Docker is used as the container platform to facilitate the creation of the application images and their deployment on the edge nodes. Two sample IoT applications are built and packaged as container images, i.e., Docker images to use for this experimentation as mentioned below.

- An image resizing application that takes an image of approximately 1.2MB in size as an input, resizes it to a lower resolution and returns a resized version of the image as the output which is approximately 250KB in size. The Docker image for this application is approximately 1.26GB in size.

- An application that performs a quicksort on a set of multiple sensor readings such as temperature, humidity and pressure. The sensor readings are stored in a JSON format in form of a dataset. The application takes this dataset as an input, performs quicksort on each of the set of sensor readings in it and renders the sorted set of values for each type of sensor as the output. The Docker image for this application is approximately 100MB in size.

In this case, the IoT resources for the above two applications, i.e., the image for the image resizing application and the sensor data set for the quicksort application,

are embedded in the container images of these application itself. In order to emulate the provisioning of several different applications on the edge nodes, different versions of these applications images are created, which upon deployment runs as individual containers, although the underlying core application is the same in each of them.

Since EDIRO aims to study the feasibility of an edge architecture that operates autonomously with minimal or no dependency on the cloud, it implicitly means that the application images must reside somewhere on the edge cluster and made available in advance. An option is to store the images in a registry server placed on the edge cluster which is reachable by each of the edge nodes. However, the approach of populating each edge node with all the application images is taken. Although this approach is not an ideal option in field scenarios considering the limited storage capacity available on an edge nodes, it is chosen and relied upon only as a temporary solution for this experimentation phase. The rationale behind the same is that it does not become a bottleneck from storage perspective as different versions of the same base image are created. The layered file system in containers ensures that the total space on the disk occupied remains constant and equivalent to the base image regardless of the several different versions of it that are created.

Measurements Overview

The system is profiled under different combinations of computing devices, workload applications and traffic conditions. The goal of the experimentation phase is to measure and analyze the performance trend of EDIRO with respect to the aforementioned parameters. For the experiments, the edge cluster topology is restricted to a two node edge computing setup to facilitate precise measurement and comparison of the performance of EDIRO in varying traffic load conditions on different types of computing devices. Opting for a two node topology consisting of similar devices facilitates stress testing in this scenario as the amount of traffic flowing into an edge node can be controlled with better ease which enables to study the performance trend precisely. Performance evaluation on a large topology consisting of similar devices on the other hand would not prove to be an ideal testing ground for this purpose considering that modern edge computing infrastructure often consists of edge servers owned by multiple vendors that vary in hardware specifications. Although it can help gain insights into specific use cases such as performance on an edge infrastructure consisting of dissimilar edge nodes, it is identified as a future work. In such scenario, specific type of edge node is bound to become a bottleneck in the system and thus would require an algorithm to enhance and adapt the orchestration policy accordingly.

The three main parameters measured in order to evaluate EDIRO are presented below along with the rationale behind their choice.

- ***Time taken to serve an on-demand client request***, especially in a large scale scenario where multiple of such requests arrive simultaneously. The end user QoS is the most important and ultimate objective of any edge computing solution. This measurement will help determine how does the client request servicing time vary with increasing traffic in large scale scenario and render insights into EDIRO's response to varying load.
- ***Overhead incurred from the distributed orchestration approach in EDIRO*** that facilitates servicing of the client requests. Measurement of this parameter will convey how much the overhead grows with increasing traffic and adds to the client servicing time of EDIRO.
- ***Computing resource utilization , i.e., the CPU and RAM usage of EDIRO*** when running on different computing devices. This is measured to get an idea of resource consumption pattern keeping in mind the resource constrained edge nodes for which EDIRO is targeted.

All the measurements are averaged over five runs. Mean is chosen as the metric instead of median because the data set of measured values of a parameter across multiple runs demonstrated a uniform trend in which the individual values in the dataset were comparable and outliers were absent.

4.2 Overhead from Distributed Orchestration

Measurement Methodology

The parameter measured here is the average time it takes for the edge infrastructure to reach a consistent state after an IoT resource is offloaded anywhere on the edge cluster. Reaching a consistent state is crucial for the maximization of the serviceability of a future client request irrespective of the edge node it is offloaded on. The edge infrastructure is said to have reached a consistent state when the information about the availability of an IoT resource offloaded on one of the edge nodes in the edge cluster is shared with all the other edge nodes such that each edge node bears an identical local record corresponding to that IoT resource. This process repeats post the offload of any new IoT resource on the edge cluster. Hence, each time a consistent state is reached for a particular IoT resource, the entire edge cluster possesses the knowledge of its availability information which is needed to serve a future client request that is dependent on it. Figure 4.1 illustrates this process for a two node edge computing setup. Step 5 specifies the achievement of the consistent state in the edge cluster. The overhead that is being referring to here corresponds to the time it takes to execute step 1 through step 5 in order.

For the measurement, the events of several IoT resources being offloaded simultaneously to the two edge nodes is simulated by subjecting the test setup to an input traffic that embeds the IoT resources information in it. This bursty traffic is applied to the edge nodes in a manner that equally distributes the number of IoT

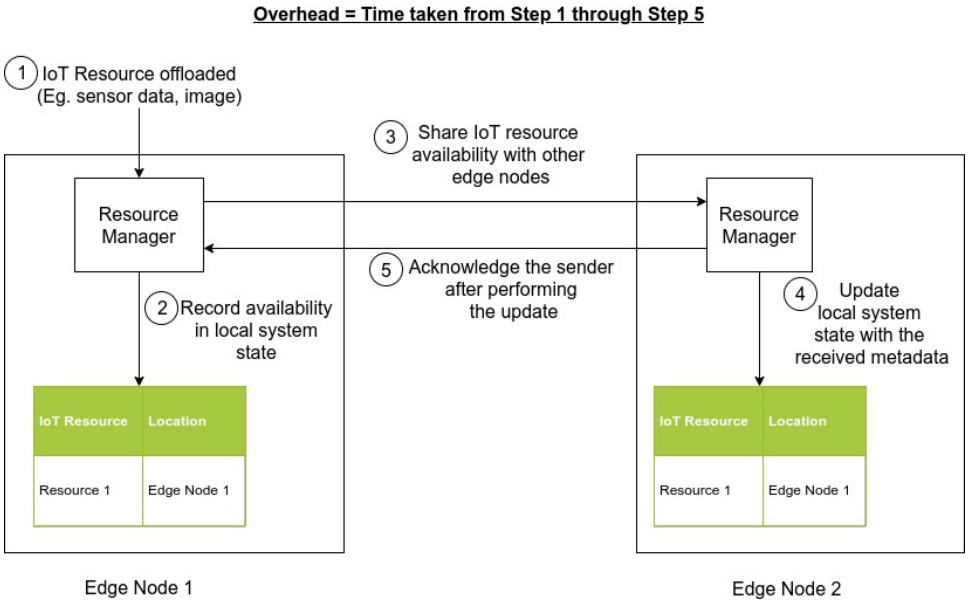


Figure 4.1: Methodology for measuring the overhead due to distributed orchestration in EDIRO to serve a client request

resources among the edge nodes. The number of IoT resources offloaded by this traffic burst is increased by 10 in every subsequent iteration. There are no client requests generated towards the edge nodes during this process as the aim is to measure the overhead caused by the inter-edge communication that takes place post the reception of an IoT resource by the edge node. Through a dedicated measurement logic embedded in the EDIRO software, the time taken to reach a consistent state for each IoT resource received by an edge noes in each iteration is measured and recorded. To convey the final result, the average of the individual values recorded for every test run is taken.

Assumption

It is assumed that the IoT resources required to serve a client request are available prior to the arrival of the request. Hence, during the test runs, the input traffic that EDIRO is subjected to is applied in a manner such that the client requests appear after the IoT resources with a certain inter-arrival time in between. The rationale behind this assumption is that in an event of unavailability of an IoT resource, either the client request will have to wait for a certain time out period before its service period expires or it would have to be offloaded to the cloud eventually. The bottleneck in each of the two cases is the time it takes for the IoT resource to be offloaded onto the edge cluster after a client request has arrived. Considering these IoT resources will be provided by other clients in the vicinity, this is an external factor and the additional latency experienced in serving the client request

will be dependent on it. Since, this does not help in gaining any insights specific to EDIRO's performance or related aspects, this assumption is made.

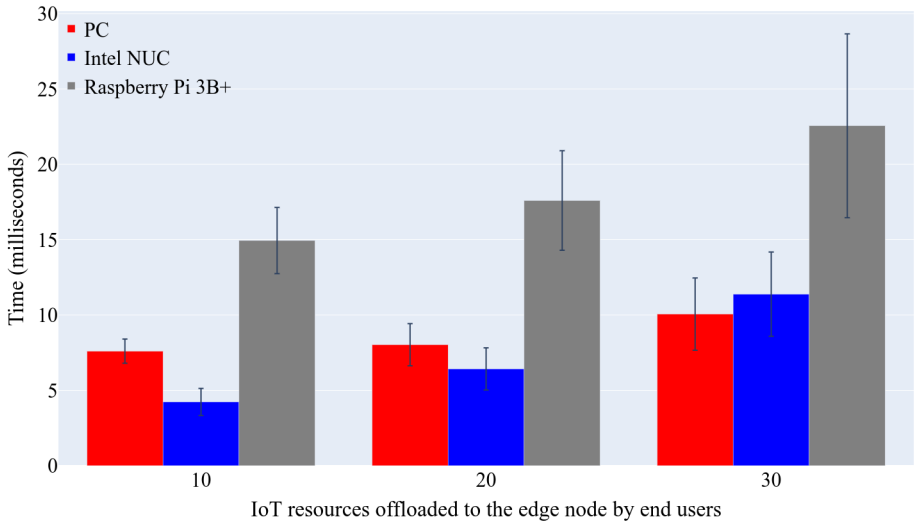


Figure 4.2: Comparison of the average overhead incurred in serving a client request due to distributed orchestration across different hardware

Results Analysis

Figure 4.2 captures the trend of the average overhead in serving a client request due to distributed orchestration across three different types of computing devices. This is the worst case scenario and hence the values indicated here forms the benchmark for the upper limit on the latency experienced in the system in the specific traffic bursts sizes. The results are interpreted in the following manner. When 10 IoT resources are offloaded simultaneously on NUC, a future client request dependent of any one of these resources can be guaranteed a successful service if it arrives after 4.2 milliseconds after this. For PC and Raspberry Pi, this value is 7.6 milliseconds and 15 milliseconds respectively.

The following key observations are made:

1. The first impression from the result is that the average overhead in serving a client request increases with the number of IoT resources offloaded on the edge node. The rate of increases is gradual as compared to the rate of increase in the traffic initially but grows steeply afterwards. Differences in computing and networking capabilities of the devices largely influences this overhead as the ability to process a traffic burst quickly and communicate with other edge nodes is tested.

2. This overhead is the result of the flat hierarchy at the edge which enables the end users to offload IoT resources anywhere on the edge cluster. However, the comparatively steep rise of the overhead with increasing traffic across hardware is an indication that at some point this overhead will outweigh the benefits of the flat hierarchy and hence it is a trade-off of this edge-driven orchestration architecture.

3. Although the multiple IoT resources embedded in the traffic burst arrive simultaneously on the edge nodes, there is a certain degree of fairness observed in the manner each of them is processed as the input is not queued for long periods waiting to be processed. This is evident from a low standard deviation as indicated in the plot which increases gradually as the size of the traffic bursts increases. This is an encouraging sign especially for traffic bursts scenario which implies that when the same number of IoT resources are offloaded with an inter-arrival time in between, the performance is bound to be better than the worst case.

4.3 Client Request Servicing Time

Measurement Methodology

The procedure of serving a client request in EDIRO involves the coordinated working of multiple concurrent processes. Hence, the time taken to serve a client request is the sum total of the time taken by each of these processes. The processes that are being referring to here are the following.

- ***Distributed orchestration to serve a client request.***
  This refers to the distributed IoT resource management process which involves the collaboration of the edge nodes to serve a client request by sharing the availability of the IoT resources on the edge cluster among each other. Since frequent inter-edge communication is the main characteristic of this process, it accounts for an overhead to the overall client request servicing time. This is another measurement parameter as already described in Section 4.2.

- ***Time taken to execute the edge processing pipeline for a single client request.***
  EDIRO establishes an edge processing pipeline to processes each incoming client request as they are received by the edge nodes. As described earlier in Figure 3.3, the sequence of operations constituting this pipeline includes performing the computation offloading, triggering of the resource monitoring feature and initiation of the containerized workload application that executes on the edge node.

The above list of processes omits the process corresponding to measuring the time it takes for the containerized application to complete its execution after it has

been launched. The rationale behind the same is that the aforementioned time will vary depending on the application and it's an aspect which does not include the involvement of EDIRO in any capacity. Hence, measurement of this time does not render any insights into any specific aspect of the EDIRO.

4.3.1 Execution time of the edge processing pipeline

Measurement Methodology

As highlighted in the Figure 3.3, at the last stage of the pipeline EDIRO contacts the underlying Docker engine to launch the containerized workload. Until this point in the pipeline, the execution control resides with EDIRO but at this point, it is transferred to the underlying Docker engine which is external to EDIRO. The measurement of the execution time of the edge processing pipeline is split into two parts in order to collect insights specific to the performance of EDIRO and study the overhead, if any, introduced by Docker. The first part of this measurement precisely measures the pipeline execution time until the point where the containerized workload is launched. The second half measures the time it takes for the Docker Engine to launch the containerized workload.

For the measurement, the events of several client requests and IoT resources being offloaded to the two edge nodes is simulated by subjecting the test setup to an input traffic that embeds the information about the client requests and the IoT resources in it. Two types of traffic is applied to the edge nodes which differs in the manner in which the client requests are offloaded to the edge nodes. This is accomplished by specifying the inter-arrival time between two consecutive client requests in the traffic index. For an input traffic consisting of multiple client requests, the time to serve each client request is measured separately. These individual values are then used to calculate the average that represents the mean time it takes for a client request to be served for a specific size of traffic burst.

Results Analysis

The trends of edge processing pipeline execution time in bursty and regular traffic scenarios for NUC, PC and Raspberry Pi are captured in the Figure 4.3, Figure 4.4 and the Figure 4.5 respectively.

The following key observations are made:

1. For the case of bursty traffic with simultaneous incoming client requests, a trend similar to the one observed for the distributed orchestration overhead is observed which shows a steep increase in the time it takes to serve a client request as the size of the traffic burst increases. This indicates that even in a large topology comprising of multiple nodes, it is possible for the end users to experience a varying QoS for similar service demands considering that a particular edge node(s) may be overloaded with serving other clients. Thus,

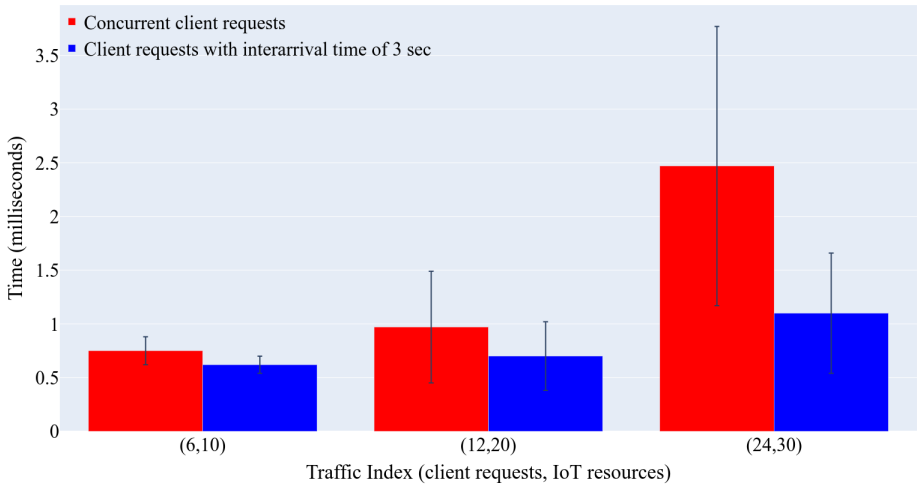


Figure 4.3: Comparison of time taken to execute the edge processing pipeline on Intel NUC in bursty and normal traffic scenarios

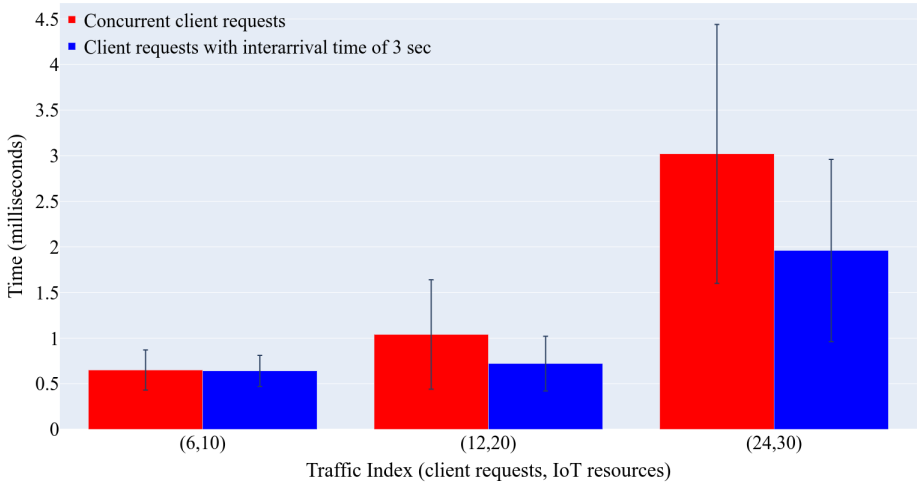


Figure 4.4: Comparison of time taken to execute the edge processing pipeline on PC in bursty and normal traffic scenarios

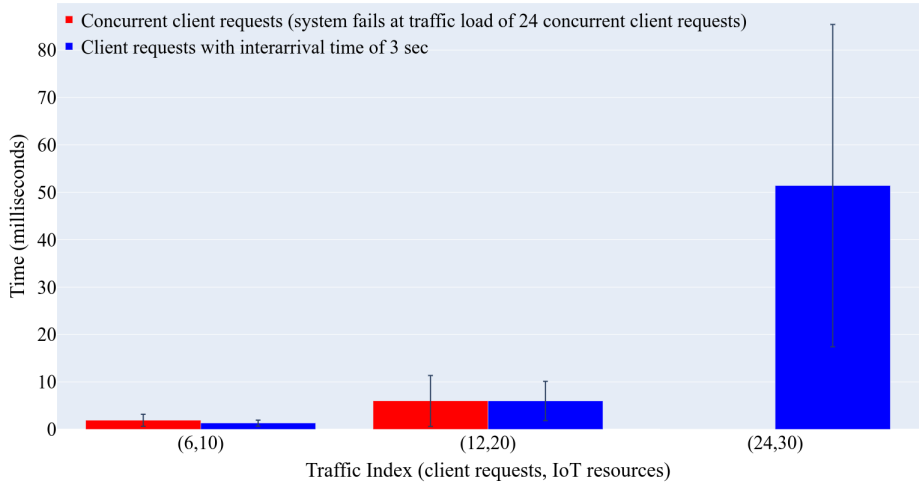


Figure 4.5: Comparison of time taken to execute the edge processing pipeline on Raspberry Pi in bursty and normal traffic scenarios

distribution of client requests in such a scenario becomes crucial but when a flat hierarchy at the edge is offered to the end users, the distribution of incoming client requests can't be controlled directly. Offloading the client request to run on another edge server could be a potential option but that would involve offloading the IoT resource, i.e, the data as well which may add an additional overhead resulting from data transmission. Hence, this is another trade-off that must be considered when using such an edge-driven architecture.

2. The latency improvement in serving the client requests with an inter-arrival time over the client requests that arrive in a traffic burst becomes significant as the traffic increases. This is an encouraging sign especially in devices such as NUC and PC because of the following reason. Since each incoming client request results in the execution of a containerized workload, the processing of every subsequent client request is subject to suffer an added latency due to the computing resources occupied by the ongoing containerized workloads of previous requests executing in the background. However, the service management design principle of EDIRO that processes each client request concurrently and separately, facilitates compensating for this latency.

3. An interesting observation is that the standard deviation in the client request servicing time for the requests that are part of the same traffic burst, increases steeply with the increase in traffic, i.e, the variation in the time taken to sever client requests that are part of the same traffic burst differ increasingly. The reason behind the same is the continuous variation in the amounts of comput-

ing resources available on the edge node due to the launch and termination of the containerized workloads that affects the processing of the subsequent client requests to come. As a result, some client requests, are processed quicker than the others, thereby only slightly affecting the average. This is later related with the findings from the computing resource utilization results. This phenomena is highlighted as an aspect of an edge-driven orchestration architecture that renders the measurement and analysis challenging due to dynamically changing parameters in the system.

4. Hardware differences influence the client request servicing time more than it influenced the overhead incurred from distributed orchestration. This is due to the reason that in order to handle multiple client requests concurrently, the system demands resources dynamically from the underlying CPU to run the required processes. This takes a toll on devices such as Raspberry Pi which failed to operate when it was subject to a traffic burst with a traffic index of (24,30,0), thereby making it an upper bound for such a device.

4.3.2 Overhead from Docker

Figure 4.6, Figure 4.8 and Figure 4.7 presents the overhead incurred due to Docker in the process of serving a client request.

Results Analysis

When launching of the containerized workload is taken into consideration for a client request, there is no visible latency improvement for client requests that arrive with an inter-arrival time over the ones that arrive simultaneously. This behavior is unlike what was observed in the case that excludes the initiation of the containerized workload initiation. This is attributed to the fact that the latency gain margin in the latter case is compensated by the delay involved in launching of the containers by the Docker engine that already manages the containerized workloads from previous requests. However, overall, this overhead maintains an approximately constant trend of variation with increasing load unlike the steep increase noticed in previous cases.

Average time taken to serve a client request

Having measured each of the processes as listed earlier in Section 4.3, the total time it takes for EDIRO to serve a client request can be calculated and is represented by the following expression.

$$\textbf{Average time to serve a client request} = A + B + C$$

where A, B and C are the following.

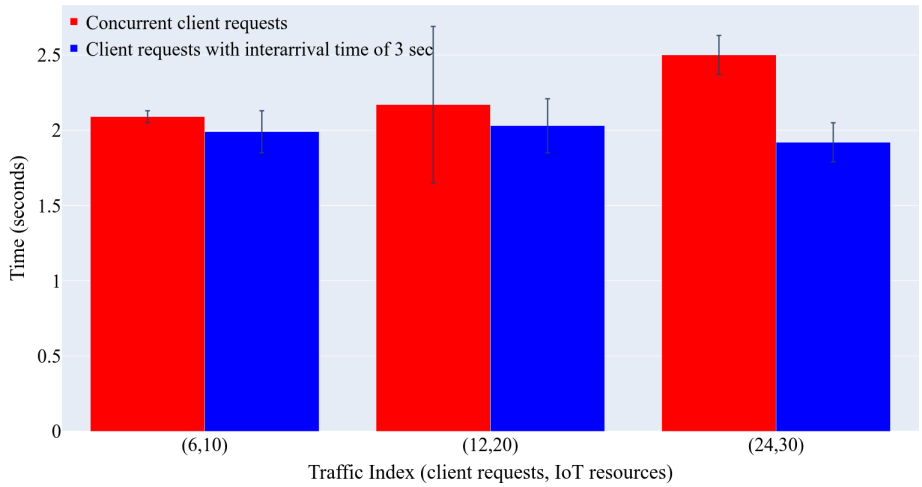


Figure 4.6: Docker overhead in orchestration of client request in Intel NUC

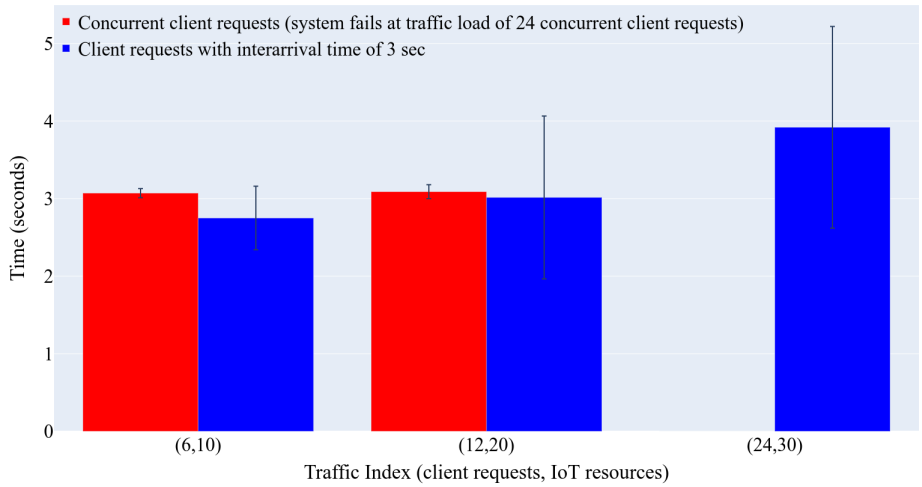


Figure 4.7: Docker overhead in orchestration of client request in Raspberry Pi

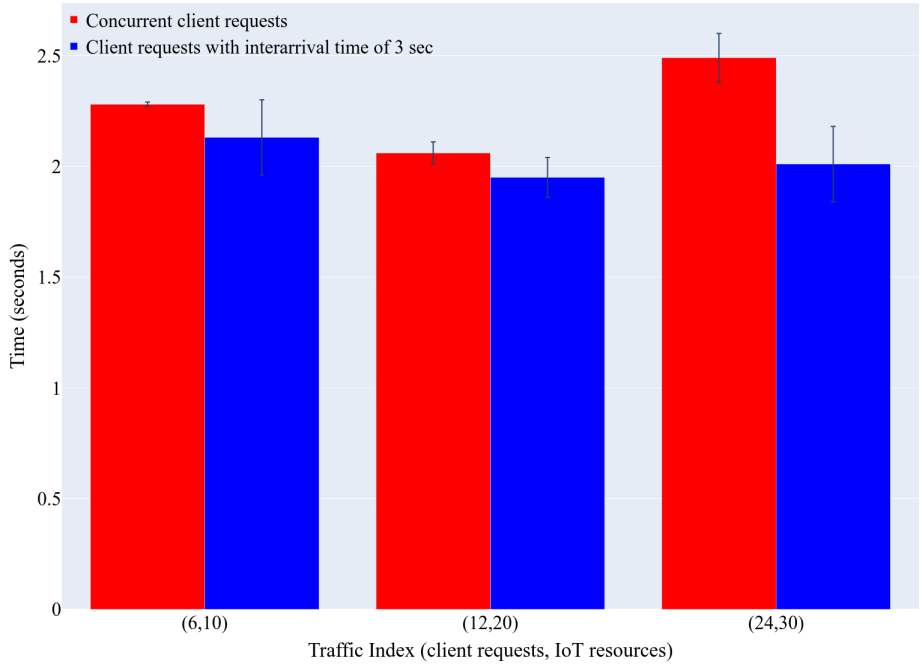


Figure 4.8: Docker overhead in orchestration of client request in PC

A: Average overhead from distributed orchestration to serve a single client request

B: Average execution time of the edge processing pipeline for a single client request excluding workload initiation

C: Average overhead from Docker to launch the containerized workload for a single client request

Results Analysis

Given that the time taken by A and B are in the order of tens of milliseconds and that by C is in the order of seconds, the average time taken to serve a client request is ultimately determined by C, i.e., the overhead from Docker. Figure 4.9 and Figure 4.10 presents the trend of the total time it takes to serve a client request in bursty and regular traffic conditions respectively.

The following key observations are made:

1. For a bursty traffic scenario, NUC and PC show similar trends of variation of the average client request servicing time with increasing size of traffic

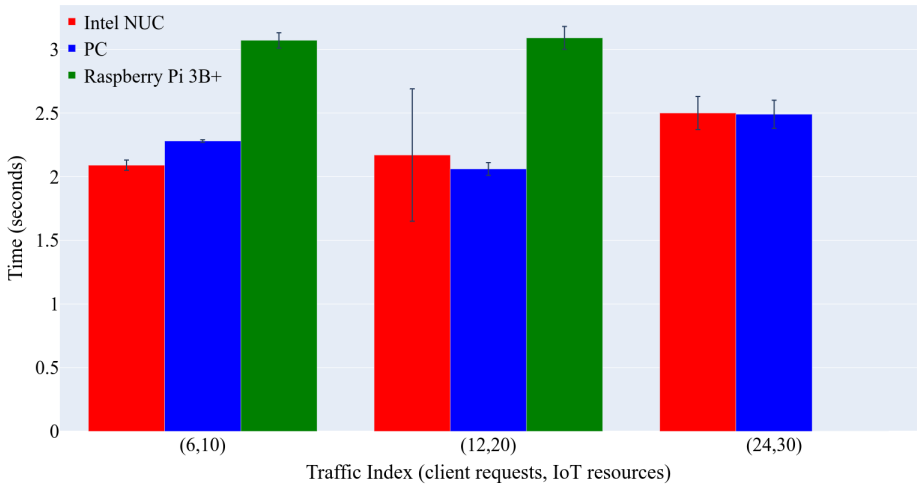


Figure 4.9: Comparison of time taken by EDIRO to serve a client request in bursty traffic scenarios (Raspberry Pi fails to handle a load of 24 concurrent client requests).

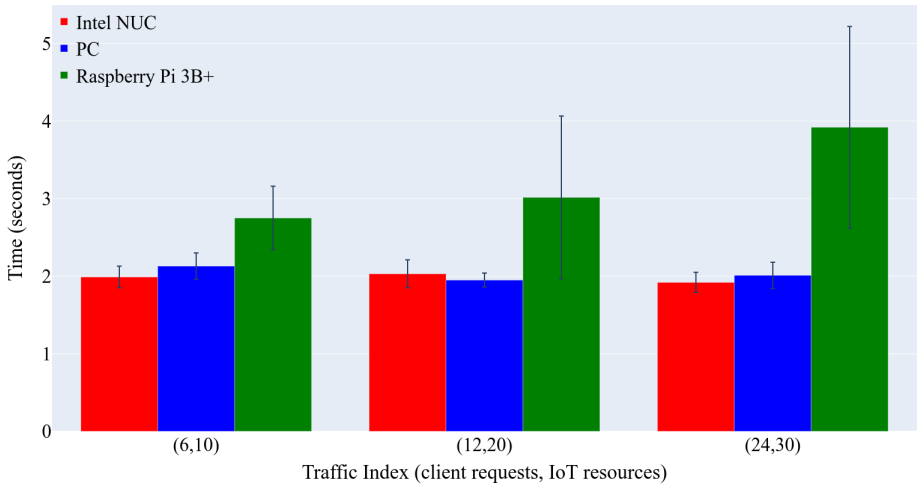


Figure 4.10: Comparison of time taken by EDIRO to serve a client request in regular traffic scenarios

bursts. When the number of simultaneous client requests that the edge nodes are subjected to, are doubled from 10 to 20 and subsequently to 30, the service time increases by 3.8% and 19.6% respectively. *This indicates that the EDIRO adapts well to a sudden increase even in a bursty traffic and controls the increase in the time it takes to serve a client requests.*

2. In the case of a regular traffic scenario for NUC and PC, no such clear increasing trend in the average client request servicing time is observed as the size of the traffic burst is increased. This can be explained in the following manner. An increase in the number of client requests implies an equal increase in the number of containerized workload to be deployed on the edge nodes. But since in a regular traffic scenario, the client requests arrive with a certain time interval in between, it gives that much additional time window for the previous workloads to be terminated and release the computing resources to be consumed by the next workloads in line, thereby reducing the load on the container deployment engine as compared to the case of the bursty traffic.

3. In the case of Raspberry Pi, the aforementioned phenomena is not observed and a distinct trend of increase in the average client request servicing time is observed as the traffic is increased. This is attributed to the lack of sufficient computing resources on low cost and portable embedded computing devices such as the Raspberry Pi. As evident from Figure 4.7, a container platform such as Docker consumes a significant amount of the computing resources when in operation. Hence, the contention for the computing resources by the workloads is continuously present leading to a development of a potential queuing effect which builds up the load on the container deployment engine resulting in an increase in the client request servicing time as seen in the Figure 4.10.

4.4 Computing Resource Utilization

Measurement Methodology

In order to determine the runtime computing resource utilization of EDIRO, the CPU and the RAM usage of EDIRO are measured during its execution on each of the edge nodes. These parameters are measured using the Linux command 'top' with a sampling interval of 1 second. The measurement of the percentage of CPU usage by EDIRO during its runtime considers the amount of CPU used by the processor to run both the user space processes as well as the Kernel space processes because the user space processes demand dynamic memory allocation and involves execution of system commands. While the RAM usage measured using the top command provides a comprehensive view of the memory usage in runtime, the memory utilization specific to the EDIRO software is also measured.

To accomplish this, the 'runtime' package of Golang is used that provides functions and structures to fetch runtime memory statistics. The functionality to collect the runtime memory statistics is embedded in the EDIRO software with a sampling interval of 1 second. The parameters measured here include the total memory reserved from the OS by the Go runtime, the current number of goroutines running and the stack in use, etc. Out of the total RAM utilized for its execution, EDIRO reserves 10 MB and 68MB of memory from the OS on Raspberry Pi and on each of NUC and PC respectively. This memory is reserved specifically for the program by the GO runtime at the beginning of its execution.

Results Analysis

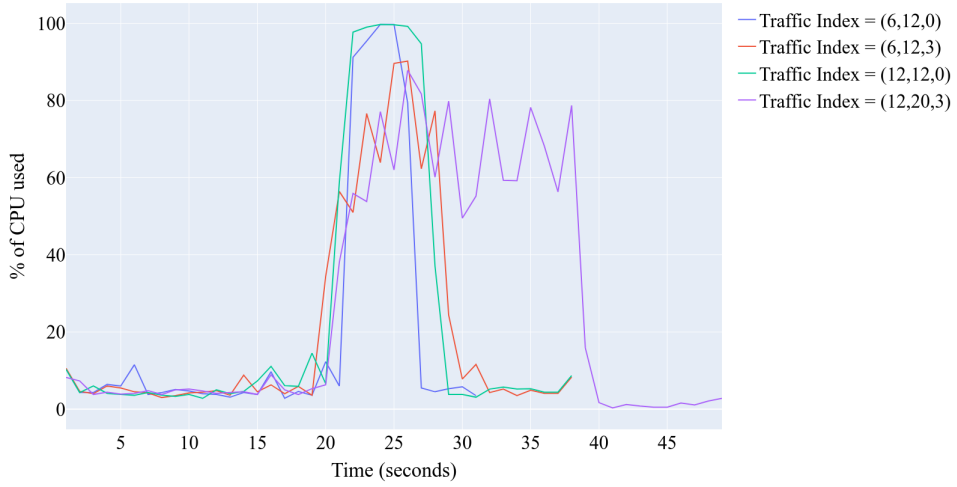
Figure 4.11a, Figure 4.12b and Figure 4.13a presents the trend of CPU and RAM utilization in runtime across the three devices. Although both NUC and PC possess similar hardware specifications, the RAM utilization in the former is almost half of that in the later. This is not due to any specific performance trend in PC as is evident from other other results where both show comparable performance, but due to some background processes running on the PC apart from EDIRO and Docker.

The following key observations are made:

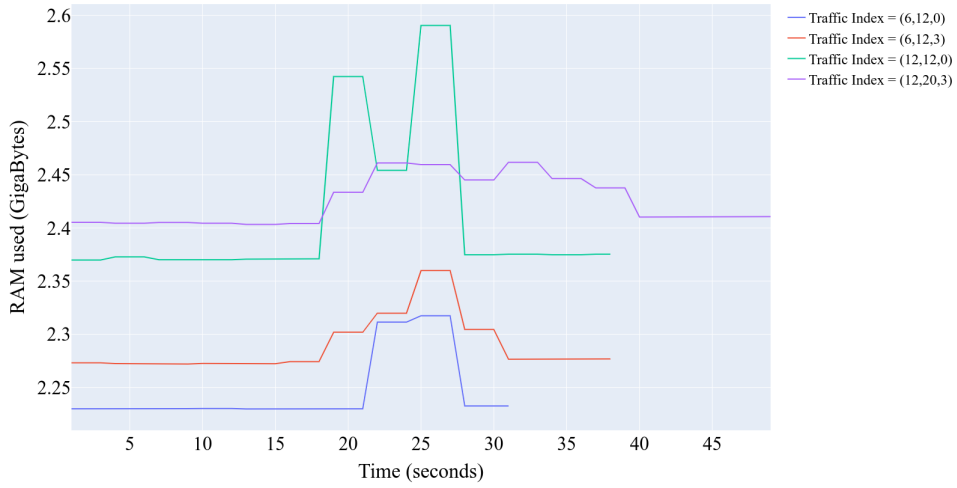
1. In the case of the client requests that come with an inter-arrival time in between, the behavior of dynamic consumption and release of computing resources is observed which is captured in the form of continuous pattern of high and low peaks over a period of time in the result plots. This is due to the combination of the several events that happen on the edge node during the execution of EDIRO while the input traffic is being applied. The high peaks correspond to the events such as starting of a containerized application and establishment of an on-demand edge processing pipeline for a client request which internally requests for memory from the CPU. The low peaks correspond to events such as termination of a containerized application, finish serving a client request, etc. However, in the case of bursty traffic, the same events keep the CPU busy all the time which is reflected in the plots by a flat line extending a short period of time.

2. A measurable difference is noticed in the CPU utilization for the client requests that arrive at the edge nodes with an inter-arrival time and the ones that arrive simultaneously as pat of a traffic burst. This difference ceases to be visible in devices such as Raspberry Pi. On an average, the CPU utilization for bursty traffic is higher by approximately 11% in NUC and 15% in PC than the traffic in which the client requests are separated by an inter-arrival time.

3. Overall, the computing resource utilization of EDIRO indicates that although the CPU resource utilization is slightly on the higher side, it is able to deliver

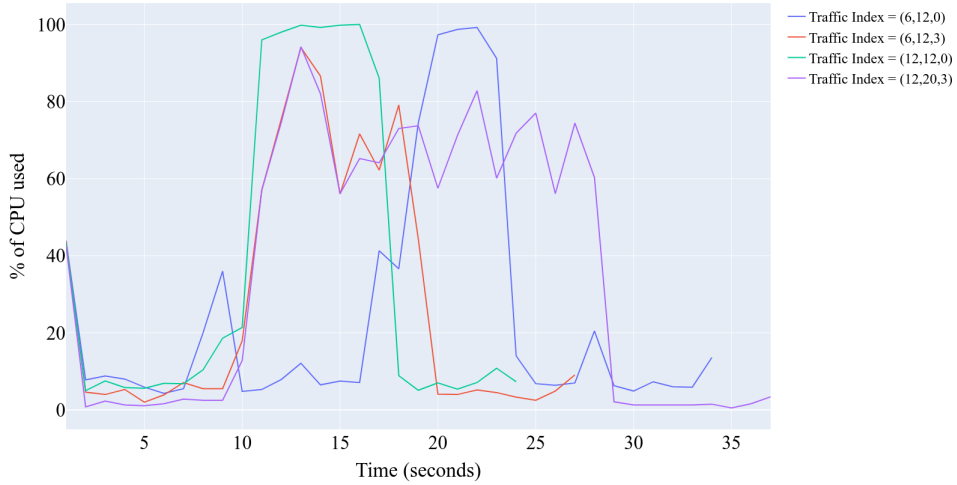


(a) CPU utilization in NUC

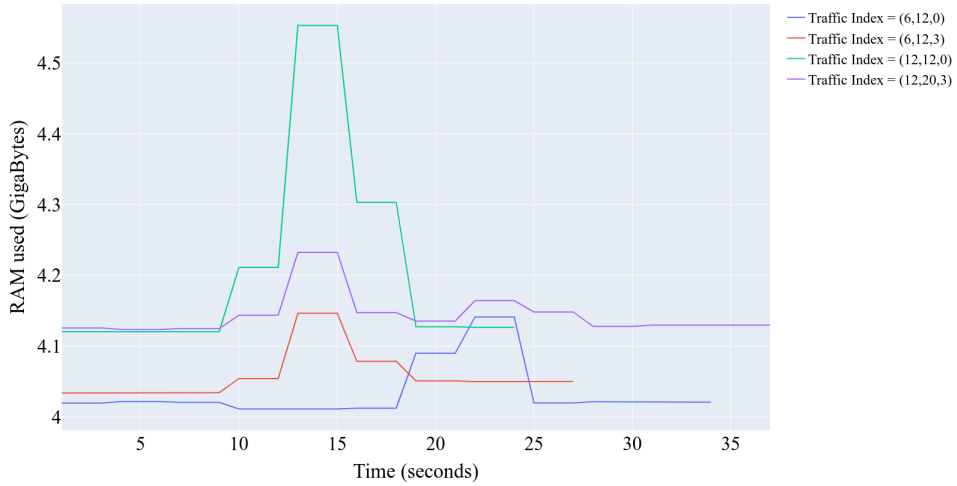


(b) RAM utilization in NUC

Figure 4.11: Computing resource utilization of EDIRO on Intel NUC

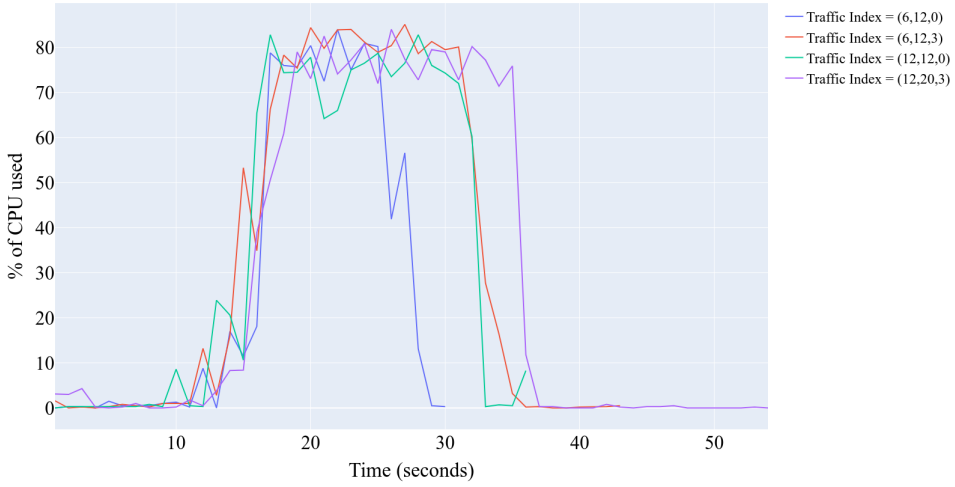


(a) CPU utilization in PC

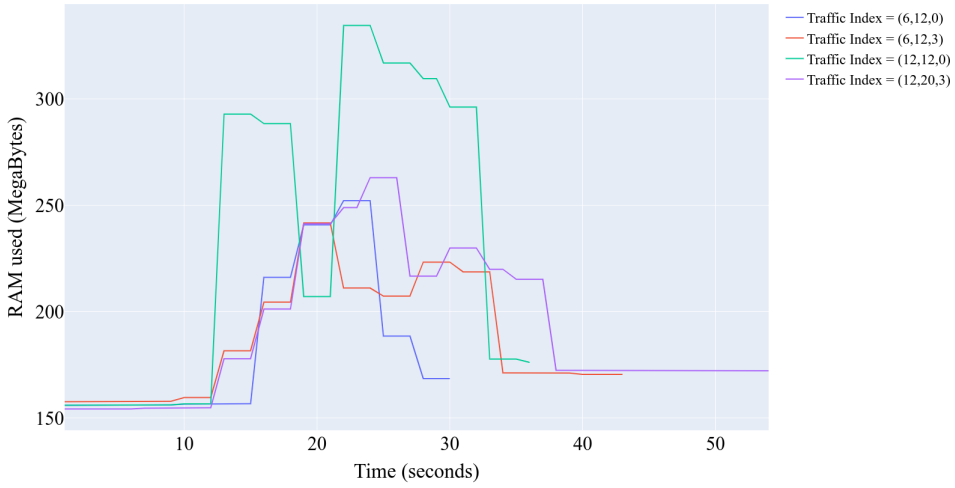


(b) RAM utilization in PC

Figure 4.12: Computing resource utilization of EDIRO on PC



(a) CPU utilization in Raspberry Pi 3B+



(b) RAM utilization in Raspberry Pi 3B+

Figure 4.13: Computing resource utilization of EDIRO on Raspberry Pi 3B+

the resources required by the constituent modules of EDIRO to facilitate handling of multiple events via concurrent processing. Moreover, analyzing the CPU utilization trend usage facilitates reasoning for the results obtained for other parameters such as the time to serve client request and the overhead due to Docker. For example, the phenomena of dynamic consumption and release of computing resources that affects the aforementioned parameters differently.

4.5 Discussion

Although the benefits of an edge-driven approach as offered via EDIRO are manifolds as described earlier in Section 2.4, there are certain overheads and limitations that come with them. These are a by product of the features and benefits offered by such an edge-driven approach. Hence, certain trade-offs need to be considered. From the results, four such sets of trade-off parameters are identified and an analysis of each of them is provided below.

Trade-off 1 : Benefits offered by the flat hierarchy or transparency at the edge vs the latency experienced by the end users.

Rendering the edge infrastructure as transparent ,i.e., enabling the end users to offload the data and requests to any of the constituent edge nodes directly implies that the same set of edge nodes need to communicate frequently in order to exchange and spread that information throughout the edge cluster. Since the knowledge of the data (IoT resources) availability on the edge cluster maximizes the chances of the edge to be able to serve the client request, this action becomes crucial. This results in an overhead and as noticed from the results, it increases steeply with the increase in the size of traffic bursts. This ultimately degrades the end user QoS due to increased latency experienced by them. Hence, this approach would tend to incline towards being infeasible for a scenario involving a large topology of edge nodes that receives a high amount of data and client requests in short bursts. However, the results in this work are generated for the worst case scenario that captures the system's response to traffic bursts of increasing sizes. Considering that this is the upper bound of the latency overhead that can be incurred by EDIRO, it can be expected that a comparatively lower overhead could be induced by the system in scenarios where consecutive IoT resources offloaded to the edge nodes are separated with a certain inter-arrival time even when the number of participating nodes in the system are on the higher side. However, an experimental evaluation of the same would be the ideal way to analyze and confirm the same.

Trade-off 2 : Number of multiple unique applications to be provisioned at the edge vs the storage resources required for the same as per the imperative approach employed in this work.

In this work, an imperative approach has been utilized for determining the application that should be executed or provisioned on the edge servers in order to serve a particular client request. This approach assumes that the dependency of the client requests with the applications is known beforehand. The imperative approach does offers benefits as applications can be launched quickly but there is a downside to it as well. Since EDIRO is an autonomous edge-driven architecture with minimal or no dependence on the cloud, the application packages must be made available at a location on the edge cluster that is reachable by each edge node. An increase in the number of unique applications required to be provisioned on the edge implies an increase in the storage resources available on the edge. Considering that a typical container image is of several hundred Megabytes or even a few Gigabytes in size, it will lead to an overkill of the resources on the edge. In the experiments conducted for this work, 24 unique applications on the edge cluster are executed. However, for experimentation purposes on handheld computing devices, several distinct replicas of two base application images are made to emulate the presence of multiple applications. To put things into perspective, in a field scenario, provisioning 24 unique applications on the edge cluster would require 24GB of storage space in form of a registry server placed somewhere on the edge.

However, recent studies have investigated and proposed an architecture that can not only solve the aforementioned problem of lack of storage space to store several different images, but also reduce the start time for the containerized workloads. Mahajan et. al propose a peer-to-peer network based container deployment system in [66] that exploits the similarity between the data blocks that comprise a container image across multiple compute nodes. The underlying idea is based on their observation that several applications use certain popular libraries for machine learning, image processing, etc. which can be referred to as a data block in the context of their proposed approach and utilized across multiple containers, thereby avoiding the need for storing a full image for each application. Hence, such a container deployment system has the potential to further improve the feasibility of EDIRO for practical scenarios.

Trade-off 3: Overhead from Docker vs the client request servicing time

The average time taken by EDIRO to serve a client request is a combination of several parameters that includes the overhead from the distributed orchestration, the overhead from Docker and the time taken to run the client request through the edge processing pipeline. An insightful observation is made after correlating the results obtained for the average time taken for edge processing pipeline execution for a client request and the average time taken to serve a client request, specifically for NUC and PC. From the Figure 4.3 and the Figure 4.4, it is observed that the

edge processing pipeline execution time follows a trend that increases steeply as the size of the traffic burst is increased irrespective of whether the client requests arrive simultaneously or with a certain inter-arrival time between them. Moreover, the similar increasing trend shown by the standard deviation in the plots indicates that the time it takes for the system to process each client request that is part of the same traffic burst varies significantly. Merging this together with the overhead from Docker to launch the containerized workloads for each client request, the average time taken by the system to serve the client request is determined as shown in the Figure 4.9 and the Figure 4.10.

Contrary to the trend of steep increase with increasing traffic as highlighted above, the average time taken to serve a client request exhibits a trend that increases rather gradually in the case of traffic burst scenarios for NUC and PC. However, the response in the case of regular traffic scenario makes for an interesting discussion point. When the edge nodes are subjected to a traffic in which the consecutive client requests are separated with an inter-arrival time, an overhead of approximately 2 seconds is added by Docker but no distinct increasing trend in the average client request servicing time is observed as the size of the traffic burst is increased. This is unlike the case when Docker overhead measurement is excluded. This indicates that overall EDIRO's performance scales well with the increase in the size of the traffic burst in the incoming traffic.

Trade-off 4 : A hybrid IoT resource-aware and computing resource-aware offloading approach vs the cost of migrating data within the edge cluster

The underlying concept of the IoT resource-aware edge offloading approach proposed in this thesis is data locality aware edge offloading which aims to bring application close to the data instead of transporting the data to the application that resides on a specific host machine [36, 71]. Furthermore, in EDIRO the edge nodes exchange the metadata about the IoT resources offloaded on them instead of the data itself to maintain a low transmission overhead. It could be possible that an edge node(s) may run out of computing resources to run the future workloads or may not guarantee the desired QoS to the end users given the scarcity of the computing resources. A potential solution in this case is to offload or migrate (if already running) the workload to another edge node that posses sufficient resources as per the concept of computing resource-aware orchestration. Since EDIRO deals with orchestrating workloads that consume an IoT resource or data, this would involve migrating the data too, which may result in transmission overhead depending upon its size.

Hence, a hybrid of IoT resource-aware and computing resource-aware orchestration approach is an ideal approach to deal with such scenarios but a runtime consideration should be made of the cost involved in terms of the overhead in the process of data transmission across the edge nodes. Such a hybrid approach in this work has not been implemented as the goal is to specifically study the behavior of IoT resource-aware offloading precisely. However, with the planned integration

of EDIRO with Docker in future, this feature could be implemented by leveraging Docker's inbuilt computing resource-aware orchestrator which would be valuable addition in terms of the fault management capabilities of EDIRO.

Chapter 5

Conclusions and Future Work

This chapter concludes the thesis by summarizing the contributions, providing the answers to the research questions defined for the thesis, making recommendations for future work and presenting the concluding remarks.

5.1 Contributions

This thesis presents the work on an edge-driven orchestration based edge computing architecture for the Internet of Things (IoT). It first investigates the feasibility of the state of the art in edge computing for IoT use cases by reviewing the current research, identifying its limitations for the upcoming and evolving trends in the IoT domain and consider the end user and service provider perspective of the utilization of an edge infrastructure to understand the knowledge and research gap that exists in this domain. In this regard, the thesis presents a detailed literature review (Chapter 2) and an extensive gap analysis (Section 2.4). These studies provide valuable insights into the current state of affairs with respect to edge computing for IoT. The gap analysis in particular provides a structured description of the key findings from the literature survey that renders insights into the shortcomings of the current research in this domain and lays emphasis on the need for an alternative architecture for edge computing through relevant justifications and examples. This phase of the thesis lays a strong foundation for the subsequent design, development and evaluation phases of the thesis.

The gap analysis also provides the answer to the below research question defined in the thesis.

- RQ1: *What is the significance of and the need for an edge-driven orchestration based edge computing architecture for IoT?*

Answer: The key finding from the literature review is that the state of the art in edge computing possesses a master-slave architectural model and operates in a top-down fashion ,i.e., the master (often located in the cloud) performs the orchestration and other control plane operations, while a set of slave compute nodes at

the edge of the network simply executes the workload deployed on them by the master. This renders the edge nodes oblivious to the events at the edge of the network and the direct interactions of the end users with them. These interactions are essentially the channel for the end users to perform computation offloading or issue on-demand service requests on the edge nodes in the context of ubiquitous computing and IoT. In order to actively respond to these interactions, the state of the art would require frequent round trips between the slave nodes and the master for the former to communicate these updates to the latter. The resulting latency from these round trips and the strong dependency on an active network connectivity between the two ends renders the state of the art infeasible for IoT use cases. Moreover, such an architecture requires the end users to be aware of the topology at the edge and restricts them from offloading their computations or placing service requests on the edge servers ubiquitously. This degrades the ease of use of the edge infrastructure by the end users and the resulting drop in its utilization subsequently affects the service provider.

An edge-driven architecture for edge computing as proposed in this thesis is able to solve the above problems. Since the edge nodes are in the vicinity of the events and the end user interactions at the edge of the network, it is efficient for the edge to drive the workload orchestration by itself. This approach can exploit opportunities of collaborative processing in an IoT domain by utilizing the IoT resources such as sensor data or images, that are either offloaded by the end users or sensed by the edge nodes from the surroundings, to serve the future client requests. On a large scale, through multiple of such edge nodes, this edge-driven approach can improve the ease of use of the edge infrastructure for the end users by offering a transparency at the edge, i.e., a flat hierarchy that allows the end users to interact with the edge nodes ubiquitously.

Based on the research gaps identified from the gap analysis in the literature review, this thesis proposes EDIRO, which is an edge-driven IoT resource-aware orchestration framework for edge computing. EDIRO is a distributed orchestration framework that facilitates service orchestration for serving the on-demand client requests through the collaboration of the multiple edge nodes in an edge cluster. Through EDIRO, it is demonstrated how to achieve an autonomous edge infrastructure that operates with minimal or no dependence on the cloud.

The design of EDIRO is discussed in reference to its application to connected vehicles in a Smart City scenario, which is also the target use case of this work. In such a domain, the clients (vehicles) contact the edge nodes embedded in the roadside infrastructure (street lights) to request location-based information queries which are served by executing specific ephemeral workloads on the edge servers. Contribution of IoT resources such as sensor data, images, etc., to the edge nodes by the same set of end users is another characteristic of such an environment. EDIRO utilizes these IoT resources to serve the client requests to simultaneously exploit and promote collaborative processing opportunities available in such scenarios that is infeasible for the state of the art. In this regard, this thesis presents the

EDIRO framework design (Section 3.1) providing detailed description, reasoning, and implications of the design choices along with the challenges that are foreseen, a prototype implementation (Section 3.2) and the rationale behind the choices of the tools used for the same (Section 3.3). This part of the thesis provides the answer to the below research question in the following manner.

- RQ2: *How to orchestrate the workloads corresponding to the on-demand service requests by utilizing the local knowledge available at the edge in the form of IoT resources?*

Answer: EDIRO is designed as an event-driven architecture that comprises of different but concurrent processing paths for handling the events ,i.e., the incoming client requests and the IoT resources offloaded to the edge nodes. A unique edge processing pipeline is established on-demand per incoming client request to process it. The management of the IoT resources is performed in a distributed fashion involving multiple edge nodes in an equal capacity. In order to maximize the chances to be able to serve the future client requests, the edge nodes work in collaboration towards maintaining a global consistent state in the edge cluster that holds the information regarding the availability of the various IoT resources on the edge cluster.

To serve the client requests, an imperative approach is opted to determine the corresponding application and the IoT resource, following which, the requests are offloaded to the edge nodes which possess the required resources. The main challenge that is solved here is approaching the system design in a manner which enables the edge nodes to concurrently perform control and management plane operations in addition to the workload execution. These tasks are embedded into dedicated modules to achieve a concurrent and coordinated functioning on the whole, especially in peak load scenarios when the utilization of the edge infrastructure goes up significantly as a result of simultaneous access by several end users.

The experiments conducted for the evaluation of EDIRO to determine its feasibility for practical IoT use cases provides the answer to the below and the primary research question defined in the thesis.

- RQ3: *What is the feasibility and the associated trade-off of this edge-driven approach for practical IoT use cases?*

Answer: Through prototype implementation and experimentation on low and mid range embedded computing devices it is demonstrated how a static edge computing architecture can be converted to one that functions autonomously with the collaboration of multiple constituent edge nodes. Experiments on a two node setup running the EDIRO framework are conducted to analyze the performance in varying traffic scenarios. The results indicate that the system adapts well to the sudden increase in the traffic load applied on it. The average client request servicing

time (including the overhead from distributed orchestration) increases by 3.8% and 19.6% when the number of multiple simultaneous clients that are trying to access the edge servers are doubled and tripled from 10 to 20 and 30 respectively. In the case where the incoming client requests were separated in time, no such increase in client serving time is observed and this trend remains approximately constant. However, this comes at the cost of an overhead contributed by the underlying container platform ,i.e., Docker in this case, which itself showed a similar trend with the increasing load on the system. This is one of the trade-offs of this edge-driven approach.

Insights from the results obtained are shared and a detailed trade-off analysis is also provided that weighs the benefits offered by the proposed edge-driven architecture against the overhead and limitations contributed by it (Section 4.5). The key points are highlighted here to summarize the same. Firstly, the benefits that a flat hierarchy or transparency at the edge offers to the end users in terms of facilitating a ubiquitous utilization of the edge infrastructure, also leads to an overhead that results from the communication among the increased number of edge nodes which can potentially receive client requests or offloaded computations. Secondly, since such an edge-drive architecture operates with minimal or no dependence on the cloud, an increase in the number of different applications that needs to be provisioned on the edge servers demands for that much extra storage space on the edge cluster to store the application packages.

The experimental evaluation in this work is focused on evaluating EDIRO on low range hardware and basic topologies to get a first view of the feasibility of this approach instead of evaluating it over large scale scenarios involving many nodes. The rationale behind the same is that such topologies with sophisticated hardware aren't realistic from edge computing scenarios perspective. Unlike cloud computing, edge computing deployments feature dissimilar nodes owned by different parties. In this regard, from the experiments it is shown that a device with low computing resources such as Raspberry Pi will become a bottleneck for such a scenario. Although the results from the measurements on a two node topology can be opportunistically extrapolated to qualitatively project the trend for topologies that grow in size, a practical evaluation will definitely give concrete information about the same and is a part of the future work. Hence, with the current state of the work and research findings it is concluded that such an edge-driven approach is feasible for practical IoT use cases considering the trade-offs associated with it.

5.2 Recommendations for Future Work

EDIRO is not yet utilized at its full potential. Below are the recommendations and suggestions through which this framework and the edge-driven architecture can be further extended.

1. Lately, the demand for performing machine learning and Artificial Intelligence at the edge has been on the rise. An edge-driven approach as proposed

in this thesis can benefit the applications and systems that employ machine learning algorithms on the edge nodes by facilitating them to sense and process the events at the edge of the network, thereby extracting the latest context from such events to improve the decisions making process.

2. An interesting future research topic concerns the application of the trade-offs listed in Section 4.5 and achieving optimization for a specific parameter such as latency, cost, energy, etc. A detailed analysis of this for specific target applications can yield further valuable insights into this edge-driven approach.

3. The inbuilt container orchestrator in Docker orchestrates the workloads based on the availability of the computing resource on the host machines. At present EDIRO runs as a standalone application on the edge nodes. An immediate future work is to integrated it with the internal container orchestration engine in Docker to further make it IoT resource-aware and extend its functionality. The outcome will be a customized binary for Docker that embeds EDIRO.

4. Currently, the overhead from using containers as the virtualization technology is on the higher side. As a future work, utilization of a lightweight virtualization technology such as unikernel for the application deployment could be considered to bring down this overhead. However, it would require the development of an additional basic orchestration and cluster management system for launching and managing unikernel deployment across the edge cluster.

5.3 Concluding Remarks

The existing edge computing solutions for IoT are unable to meet its evolving demands and adapt to the current trends as they lack to incorporate the key aspects that govern the utilization of an edge computing infrastructure. These aspects concern with identifying the specific type of edge computing services offered in an IoT domain, the opportunities to exploit collaborative processing among a group of end users and enhancing the ease of use of the edge computing infrastructure for the end users. An architectural style and an operational methodology that is significantly dominated by cloud computing is one of the primary reasons that hinders the state of the art in edge computing from addressing the aforementioned aspects.

This thesis proposes an edge-driven orchestration based edge computing architecture that bridges the gaps left behind by the literature in their efforts to realize edge computing solutions for IoT. The feasibility of this proposed approach for practical scenarios is shown through a proof of concept development and experimentation. The thesis shares the insights gained into this edge-driven approach by highlighting the key considerations that should be made in the form of a detailed trade-off analysis which discusses the associated benefits and the limitations.

Based on the research findings, the conclusion drawn from this thesis is that an edge-driven approach in edge computing is feasible and possesses a promising scope for future edge computing solutions in the IoT domain. Further enhancements and customization to this work as per the documented findings could help in the exploitation of this approach to its full potential or discovering further insights into the same. Several emerging trends such as ubiquitous computing, Machine learning and Artificial Intelligence at the edge can benefit from the proposed edge-driven approach in particular due to its distributed, autonomous and bottom-up style of operation. In this regard, the insights shared from this work can prove as valuable guidelines that lay the foundation over which an edge computing solution can be implemented for the desired application or use case. The EDIRO framework itself is a useful asset which is released as open source to encourage further extension and collaboration.

Bibliography

[1] Aws iot greengrass: Bring local compute, messaging, data management, sync, and ml inference capabilities to edge devices. https://aws.amazon.com/greengrass/.

[2] Azure iot edge: Cloud intelligence deployed locally on iot edge devices. https://azure.microsoft.com/en-us/services/iot-edge/.

[3] Azure iot edge source code. https://github.com/Azure/iotedge.

[4] Docker service update options. https://docs.docker.com/engine/reference/commandline/service_update/.

[5] Ediro source code. https://github.com/niketagrawal/EDIRO.

[6] Firecracker: Secure and fast microvms for serverless computing. https://firecracker-microvm.github.io/.

[7] The go programming language. https://golang.org/.

[8] Goroutines vs threads. http://tleyden.github.io/blog/2014/10/30/goroutines-vs-threads/.

[9] grpc: A high performance, open-source universal rpc framework. https://grpc.io/.

[10] Ieee 1934-2018 - ieee standard for adoption of openfog reference architecture for fog computing. https://standards.ieee.org/standard/1934-2018.html.

[11] Includeos - run your application with zero overhead. https://www.includeos.org/.

[12] Industrial internet consortium. https://www.iiconsortium.org/index.html.

[13] An introduction to programming in go. https://www.golang-book.com/books/intro.

[14] Kubeedge: An open platform to enable edge computing. https://kubeedge.io/en/.

[15] Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

[16] The little go book. https://www.openmymind.net/assets/go/go.pdf.

[17] Mirage os: A programming framework for building type-safe, modular systems. https://mirage.io/.

[18] Moby: An open framework to assemble specialized container systems without reinventing the wheel. https://mobyproject.org/.

[19] Protocol buffers. https://developers.google.com/protocol-buffers.

[20] Remote procedure call. https://en.wikipedia.org/wiki/Remote_procedure_call.

[21] Swarm mode key concepts. https://docs.docker.com/engine/swarm/key-concepts/.

[22] Swarm mode overview. https://docs.docker.com/engine/swarm/.

[23] Unikernels: Rethinking cloud infrastructure. http://unikernel.org/.

[24] What is a container?: A standardized unit of software. https://www.docker.com/resources/what-container.

[25] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. Mobile edge computing: A survey. *IEEE Internet of Things Journal*, 5(1):450–465, Feb 2018.

[26] Niket Agrawal, Jan Rellermeyer, and Aaron Yi Ding. Iot resource-aware orchestration framework for edge computing. In *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '19, page 62–64, New York, NY, USA, 2019. Association for Computing Machinery.

[27] A. Ahmed and G. Pierre. Docker container deployment in fog computing infrastructures. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 1–8, July 2018.

[28] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015.

[29] Abdulazaz Ali Albalawi, Asit Chakraborti, Cedric Westphal, Dirk Kutscher, Jeffrey He, and Quinton Hoole. Inca: An architecture for in-network computing. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms*, ENCP '19, pages 56–62, New York, NY, USA, 2019. ACM.

[30] Z. Amjad, A. Sikora, B. Hilt, and J. Lauffenburger. Low latency v2x applications and network requirements: Performance evaluation. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 220–225, June 2018.

[31] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[32] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, Oct. 2010.

[33] G. Avino, M. Malinverno, F. Malandrino, C. Casetti, and C. F. Chiasserini. Characterizing docker overhead in mobile edge computing scenarios. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet '17, pages 30–35, New York, NY, USA, 2017. ACM.

[34] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 250–257, Nov 2015.

[35] D. Bruneo, S. Distefano, F. Longo, G. Merlino, A. Puliafito, V. D'Amico, M. Sapienza, and G. Torrisi. Stack4things as a fog computing platform for smart city applications. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 848–853, April 2016.

[36] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. Fades: Fine-grained edge offloading with unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet '17, pages 36–41, New York, NY, USA, 2017. ACM.

[37] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. Edge chaining framework for black ice road fingerprinting. In *Proceedings of the 2Nd International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '19, pages 42–47, New York, NY, USA, 2019. ACM.

[38] M. S. de Brito, S. Hoque, T. Magedanz, R. Steinke, A. Willner, D. Nehls, O. Keils, and F. Schreiner. A service orchestration architecture for fog-enabled infrastructures. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 127–132, May 2017.

[39] T. V. Doan, G. T. Nguyen, H. Salah, S. Pandi, M. Jarschel, R. Pries, and F. H. P. Fitzek. Containers vs virtual machines: Choosing the right virtualization technology for mobile edge cloud. In *2019 IEEE 2nd 5G World Forum (5GWF)*, pages 46–52, Sep. 2019.

[40] H. El-Sayed, S. Sankar, M. Prasad, D. Puthal, A. Gupta, M. Mohanty, and C. Lin. Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment. *IEEE Access*, 6:1706–1717, 2018.

[41] D. Evans. Mobile edge computing: A key technology towards 5g. White paper, ETSI, `https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf`, September 2015.

[42] H. Flores, Xiang Su, V. Kostakos, A. Y. Ding, P. Nurmi, S. Tarkoma, P. Hui, and Y. Li. Large-scale offloading in the internet of things. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 479–484, March 2017.

[43] Julien Gedeon, Jeff Krisztinkovics, Christian Meurisch, Michael Stein, Lin Wang, and Max Mühlhäuser. A multi-cloudlet infrastructure for future smart cities: An empirical study. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*, EdgeSys'18, pages 19–24, New York, NY, USA, 2018. ACM.

[44] J. Gedeon, M. Stein, J. Krisztinkovics, P. Felka, K. Keller, C. Meurisch, L. Wang, and M. Mühlhäuser. From cell towers to smart street lamps: Placing cloudlets on existing urban infrastructures. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 187–202, Oct 2018.

[45] M. Gerla, E. Lee, G. Pau, and U. Lee. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 241–246, March 2014.

[46] OpenFog Consortium Architecture Working Group. Openfog reference architecture for fog computing. `https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf`, February 2017.

[47] C. Harrison, B. Eckman, R. Hamilton, P. Hartswick, J. Kalagnanam, J. Paraszczak, and P. Williams. Foundations for smarter cities. *IBM Journal of Research and Development*, 54(4):1–16, July 2010.

[48] W. He, G. Yan, and L. D. Xu. Developing vehicular data cloud services in the iot environment. *IEEE Transactions on Industrial Informatics*, 10(2):1587–1595, May 2014.

[49] S. Hoque, M. S. d. Brito, A. Willner, O. Keil, and T. Magedanz. Towards container orchestration in fog computing infrastructures. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 294–299, July 2017.

[50] Y. Hsieh, H. Hong, P. Tsai, Y. Wang, Q. Zhu, M. Y. S. Uddin, N. Venkatasubramanian, and C. Hsu. Managed edge computing on internet-of-things devices for smart city applications. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–2, April 2018.

[51] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. Quantifying the impact of edge

computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[52] S. Ibrahim, B. He, and H. Jin. Towards pay-as-you-consume cloud computing. In *2011 IEEE International Conference on Services Computing*, pages 370–377, July 2011.

[53] C. Jiang, X. Cheng, H. Gao, X. Zhou, and J. Wan. Toward computation offloading in edge computing: A survey. *IEEE Access*, 7:131543–131558, 2019.

[54] Y. Jiang, Z. Huang, and D. H. K. Tsang. Challenges and solutions in fog computing orchestration. *IEEE Network*, 32(3):122–129, May 2018.

[55] J. Jin, J. Gubbi, S. Marusic, and M. Palaniswami. An information framework for creating a smart city through internet of things. *IEEE Internet of Things Journal*, 1(2):112–121, April 2014.

[56] Gorkem Kar, Shubham Jain, Marco Gruteser, Fan Bai, and Ramesh Govindan. Real-time traffic estimation at vehicular edge nodes. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 3:1–3:13, New York, NY, USA, 2017. ACM.

[57] K. Kritikos and P. Skrzypek. A review of serverless frameworks. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168, Dec 2018.

[58] M. T. Lazarescu. Design of a wsn platform for long-term environmental monitoring for iot applications. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 3(1):45–54, March 2013.

[59] L. Lin, X. Liao, H. Jin, and P. Li. Computation offloading toward edge computing. *Proceedings of the IEEE*, 107(8):1584–1607, Aug 2019.

[60] C. Long, Y. Cao, T. Jiang, and Q. Zhang. Edge computing framework for cooperative video processing in multimedia iot systems. *IEEE Transactions on Multimedia*, 20(5):1126–1139, May 2018.

[61] G. Luo, Q. Yuan, H. Zhou, N. Cheng, Z. Liu, F. Yang, and X. S. Shen. Cooperative vehicular content distribution in edge computing assisted 5g-vanet. *China Communications*, 15(7):1–17, July 2018.

[62] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 162–169, Dec 2017.

[63] Lele Ma, Shanhe Yi, and Qun Li. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 11:1–11:13, New York, NY, USA, 2017. ACM.

[64] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[65] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30–44, Dec. 2013.

[66] Kunal Mahajan, Saket Mahajan, Vishal Misra, and Dan Rubenstein. Exploiting content similarity to address cold start in container deployments. In *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '19, pages 37–39, New York, NY, USA, 2019. ACM.

[67] Jonathan McChesney, Nan Wang, Ashish Tanwer, Eyal de Lara, and Blesson Varghese. Defog: Fog computing benchmarks. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, pages 47–58, New York, NY, USA, 2019. ACM.

[68] G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, June 2017.

[69] N. Mohamed, J. Al-Jaroodi, S. Lazarova-Molnar, I. Jawhar, and S. Mahmoud. A service-oriented middleware for cloud of things and fog computing supporting smart city applications. In *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–7, Aug 2017.

[70] R. Morabito. Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access*, 5:8835–8850, 2017.

[71] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott. Consolidate iot edge computing with lightweight virtualization. *IEEE Network*, 32(1):102–111, Jan 2018.

[72] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 20:1–20:13, New York, NY, USA, 2017. ACM.

[73] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos. A comprehensive survey on fog computing: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 20(1):416–464, Firstquarter 2018.

[74] M. Mukherjee, L. Shu, and D. Wang. Survey of fog computing: Fundamental, network applications, and research challenges. *IEEE Communications Surveys Tutorials*, 20(3):1826–1857, thirdquarter 2018.

[75] M. Naphade, G. Banavar, C. Harrison, J. Paraszczak, and R. Morris. Smarter cities and their innovation challenges. *Computer*, 44(6):32–39, June 2011.

[76] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee. A container-based edge cloud paas architecture based on raspberry pi clusters. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 117–124, Aug 2016.

[77] C. Pahl and B. Lee. Containers and clusters for edge cloud architectures – a technology review. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 379–386, Aug 2015.

[78] J. Pan and J. McElhannon. Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal*, 5(1):439–449, Feb 2018.

[79] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. Avr: Augmented vehicular reality. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 81–95, New York, NY, USA, 2018. ACM.

[80] Thomas Rausch. Message-oriented middleware for edge computing applications. In *Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference*, Middleware '17, pages 3–4, New York, NY, USA, 2017. ACM.

[81] B. P. Rimal, E. Choi, and I. Lumb. A taxonomy and survey of cloud computing systems. In *2009 Fifth International Joint Conference on INC, IMS and IDC*, pages 44–51, Aug 2009.

[82] Y. Sahni, J. Cao, S. Zhang, and L. Yang. Edge mesh: A new paradigm to enable distributed intelligence in internet of things. *IEEE Access*, 5:16441–16458, 2017.

[83] D. Santoro, D. Zozin, D. Pizzolli, F. D. Pellegrini, and S. Cretti. Foggy: A platform for workload orchestration in a fog computing environment. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 231–234, Dec 2017.

[84] M. Sapienza, E. Guardo, M. Cavallo, G. La Torre, G. Leombruno, and O. Tomarchio. Solving critical events through mobile edge computing: An approach for smart cities. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–5, May 2016.

[85] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, Aug 2001.

[86] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, Jan 2017.

[87] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009.

[88] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[89] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, May 2016.

[90] K. Su, J. Li, and H. Fu. Smart city and the applications. In *2011 International Conference on Electronics, Communications and Control (ICECC)*, pages 1028–1031, Sep. 2011.

[91] Y. Sun, H. Song, A. J. Jara, and R. Bie. Internet of things and big data analytics for smart and connected communities. *IEEE Access*, 4:766–773, 2016.

[92] T. Taleb, S. Dutta, A. Ksentini, M. Iqbal, and H. Flinck. Mobile edge computing potential in making cities smarter. *IEEE Communications Magazine*, 55(3):38–43, March 2017.

[93] M. Taneja. A mobility analytics framework for internet of things. In *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 113–118, Oct 2015.

[94] Selome Kostentinos Tesfatsion, Cristian Klein, and Johan Tordsson. Virtualization techniques compared: Performance, resource, and power usage overheads in clouds. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 145–156, New York, NY, USA, 2018. ACM.

[95] L. Tong, Y. Li, and W. Gao. A hierarchical edge cloud architecture for mobile computing. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[96] P. Tsai, H. Hong, A. Cheng, and C. Hsu. Distributed analytics in fog computing platforms using tensorflow and kubernetes. In *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 145–150, Sep. 2017.

[97] A. J. J. Valera, M. A. Zamora, and A. F. G. Skarmeta. An architecture based on internet of things to support mobility and security in medical environments. In *2010 7th IEEE Consumer Communications and Networking Conference*, pages 1–5, Jan 2010.

[98] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup. Serverless is more: From paas to present cloud computing. *IEEE Internet Computing*, 22(5):8–17, Sep. 2018.

[99] J. Wan, D. Zhang, S. Zhao, L. T. Yang, and J. Lloret. Context-aware vehicular cyber-physical systems with cloud support: architecture, challenges, and solutions. *IEEE Communications Magazine*, 52(8):106–113, Aug 2014.

[100] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos. Fog orchestration for internet of things services. *IEEE Internet Computing*, 21(2):16–24, Mar 2017.

[101] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, Oct 2018.

[102] M. Yannuzzi, F. van Lingen, A. Jain, O. L. Parellada, M. M. Flores, D. Carrera, J. L. Pérez, D. Montero, P. Chacin, A. Corsaro, and A. Olive. A new era for cities with fog computing. *IEEE Internet Computing*, 21(2):54–67, Mar 2017.

[103] 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)*, pages 58–65, June 2017.

[104] A. Yousefpour, G. Ishigaki, and J. P. Jue. Fog computing: Towards minimizing delay in the internet of things. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 17–24, June 2017.

[105] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang. A survey on the edge computing for the internet of things. *IEEE Access*, 6:6900–6919, 2018.

[106] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, Feb 2014.

[107] K. Zhang, Y. Mao, S. Leng, Y. He, and Y. ZHANG. Mobile-edge computing for vehicular networks: A promising network paradigm with predictive off-loading. *IEEE Vehicular Technology Magazine*, 12(2):36–44, June 2017.

[108] Pengyuan Zhou, Wenxiao Zhang, Tristan Braud, Pan Hui, and Jussi Kangasharju. Arve: Augmented reality applications in vehicle to edge networks. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*, MECOMM'18, pages 25–30, New York, NY, USA, 2018. ACM.

[109] P. Zhou, W. Zhang, T. Braud, P. Hui, and J. Kangasharju. Enhanced augmented reality applications in vehicle-to-edge networks. In *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 167–174, Feb 2019.