2-11-2021 10:30 AM

# Load Balancing and Resource Allocation in Smart Cities using Reinforcement Learning

Aseel AlOrbani, *The University of Western Ontario*

Supervisor: Bauer, Michael A., *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

# Abstract

Today, smart city technology is being adopted by many municipal governments to improve their services and to adapt to growing and changing urban population. Implementing a smart city application can be one of the most challenging projects due to the complexity, requirements and constraints. Sensing devices and computing components can be numerous and heterogeneous. Increasingly, researchers working in the smart city arena are looking to leverage edge and cloud computing to support smart city development. This approach also brings a number of challenges. Two of the main challenges are resource allocation and load balancing of tasks associated with processing data from sensors, etc. This can be particularly challenging depending on the frequency that tasks come and go, their complexity, the level of resources, etc. This is a dynamically changing environment and static allocation strategies are not effective. This thesis investigates a reinforcement learning approach to dynamically allocate tasks to resources and try to ensure balanced loads on processing elements. The agent follows a Multi-Observation Single-State (MOSS) model which allows it to observe processed features from multiple sources at a single step. Those features represent multiple registered virtual machines (executors). The agent tries to orchestrate the arriving task to one of the executor candidates based on the task's characteristics and current condition of the executor. We introduce a model of a smart city computational infrastructure, describe our approach to reinforcement learning and present our algorithm for task allocation. We illustrate the agent behavior through simulations and show how its performance improves as it learns the environment.

**Keywords:** Load balancing, resource allocation, smart cities, reinforcement learning, features engineering, edge computing

# Summary for Lay Audience

Cities continue to grow in population, size, services and complexity. City governments need smarter ways to accommodate the needs of their citizens and are looking at ways to make their cities smarter. Small sensing and computing devices coupled with powerful computers are being used to create smart city systems. While these systems have the potential to improve city-wide operations and services and to benefit citizens, they also present challenges in their operation. These resources are costly to acquire and to operate, and need to be managed wisely to prevent excessive costs. This also means that use of these computers to execute tasks needs to be as efficient as possible, to avoid over loading and wasting resources. This is the problem of resource allocation and load balancing. In this thesis, we introduce a reinforcement learning method for resource allocation and load balancing and evaluate its effectiveness.

# Acknowledgements

Throughout my journey in researching and writing my thesis, I had valuable support and assistance. First of all, I would like to express my gratitude to my supervisor, Prof. Michael A. Bauer. His expertise and knowledge played a great role in formulating this thesis. Western helped a great deal in making me feel home and supporting me morally and financially. Many thanks goes to my friends and advisors: Dr. Yehia Kotb and Mohammad Kotb, who helped in revising and reviewing my thesis. As always, my father is my all-time supporter, as he always believed in me, thank you dad. Thank you mom for having faith in me, and thank you for your encouragement. I would also like to thank my sisters, friends, and colleagues for their continuous support throughout my journey. Finally, I could not have completed this thesis without the unconditional help of my best friend and fiance Eng. Ali Abdelali, thank you for your contribution and for believing in me.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

SMART city has been a captivating topic for the last decade [79, 18]. Like many new concepts, there is no agreed upon definition of a ”smart city” among researchers and scientists [21]. This is because many researchers use their own definition based on what they expect from the concept of **smart city**. Some see it as a collection of e-services [8], others see it as a green city [25]. These are just two views among many others [21]. Regardless of what the definition is, due to the complexity of the system and the potentially huge amount of data that can be streamed to servers, the computational infrastructure of the smart city is commonly recognized as involving the following: Fog computing [93], artificial intelligence [39], Internet of Things (IoT) [27], and big data [9] (see Figure 1.1). Moreover, applications associated with smart city initiatives span a range of domains from healthcare [63, 12] to robotics [46]. Figure 1.1 is a Venn diagram that illustrates that a smart city is the intersection of those four big concepts.

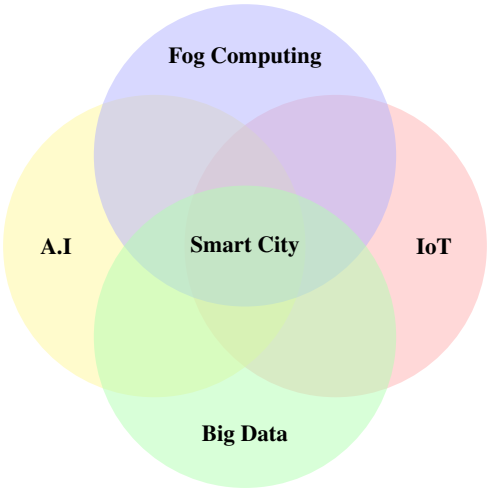


Figure 1.1: Overlapping Technologies for Smart Cities

The idea of a smart city is directly related to the government, administrators, and citizens of cities. The cost of administrating cities is increasing with the increase of their population and citizen expectations. For a better life quality, people migrate to advanced cities which provide

enhanced security, healthier living and working environments, excellent response system to emergencies, and smart houses. An autonomous smart city system can predict, recognize, analyse, and response to different city events. Since each city has its own type of common events, smart city systems should be general and flexible to be tailored to specific types of events.

## 1.1 Problem Statement

This thesis focuses on smart cities based on fog computing. Fog computing is the use of on-demand servers, dedicated servers, nodes with limited capabilities and network devices to run large scale applications on distributed systems. It is a hierarchical network and computational structure that attempts to adapt to the variation of demands in complex technology systems such as those emerging in smart cities. Although the motive for such complex structure is to reduce delays and increase throughput, to do so requires good resource management algorithms and a load balancing techniques. Lacking good resource management could result in overloading computing resources while other resources are idle. This could result in loss of performance or overprovisioning of devices or both.

This problem is often best solved by people, but the growing size and complexity of these environments makes this increasingly challenging for humans and just adding manpower has limitations. There is a need to address these problems by computational means, including methods that rely on artificial intelligence and machine learning. AI approaches can take into account constraints, different characteristics and multiple objectives of resource management for a specific system. The learning strategy could take a good amount of time but, once it learns a good behaviour from experience, it will be able to predict future events and act in advance. Just like police boost their security forces on annual festivals and carnivals, an artificially intelligent agent will learn the date of such events along with the increased data collection based on history and will adapt to additional tasks and resource demands.

Smart cities are expected to rely on sensors and data collection devices - an Internet of Things (IoT) system. IoT devices are the input for smart city systems and are the connection between the digital and physical worlds. They are part of the input for the system, and may exist in the thousands or millions of devices [69]. Such input data is huge in volume, may have different forms, and is collected at different rates. This data often needs efficient algorithms for filtering and pre-processing before injecting it to the system, may be used by other algorithms that combine it with other data sources into merged data sets used by smart city application [74]. Large scale data collection and analyses are one of the challenges hindering the development and emergence of smart cities. In the fog computing paradigm, many different nodes may exist to handle much of the processing, but there is also a need to efficiently place tasks that need to execute on appropriate nodes i.e. allocating tasks. Task misplacement can affect the performance of the running tasks or impact the deployment of queued tasks and the overall cost of operating the infrastructure. The cost of the system includes, but is not limited to, the cost of using on-demand servers per time unit, energy consumed in both computation and network

nodes, and bandwidth usage. Cost could also include the cost for human intervention needed for running the system. To get the best out of fog computing, resource management and load balancing algorithms are essential.

To conclude, fog computing paradigm's performance depends on the load balancing and resource allocation technique/model used. For a smart city system to succeed it must have consistent and adequate performance and the algorithms must be able to handle changing workloads. Artificial intelligence approaches have been successful in a number of different domains in the past years. We propose a Reinforcement Learning algorithm based on an abstract model of a smart city infrastructure to address task allocation and load balancing.

## 1.2 Thesis Structure

The rest of this thesis is organized as follows. Chapter 2 provides some background and covers some related work. Chapter 3 introduces concepts that are necessary for the thesis. After that, Chapter 4 introduces the structure and dynamics of our abstract model of smart city infrastructure and we describe our simulator. Chapter 5 introduces the model used to address the allocation and balancing problems. Experimental results are presented and analysed in Chapter 6. Finally, the conclusions, limitations of the proposed model and future work are presented in Chapter 7.

# Chapter 2

# Literature Survey

Internet of things **(IoT)** is one of the dominant topics during the last few years. IoT is vital for many applications including but not limited to smart city, self-driving vehicles, drone and robot swarms, distributed sensor systems and many others. The more popular IoT becomes, the more vital it is to have scalable models for analysis of IoT environments, particularly where scalability is a concern. Many IoT systems have begun to look towards fog computing to help address computing, data management, timely processing and energy efficiency rather than just relying on cloud resources. Executing tasks on fog nodes can be more energy efficient than executing those tasks on a cloud and can provide enhanced performance measures, such as reduced response time. In this Chapter, we provide an overview of cloud and fog computing, and review previous research on resource allocation and load balancing in fog environments.

## 2.1 Background

Traditional approaches to handle the challenges of computational complexity by relying solely on heavy duty servers is no longer sufficient. These servers were typically available to bigger corporations due to the complexity, prices and required knowledge to operate them. A more modern approach to satisfy the demands of digital computation was through the introduction of distributed systems and parallel computation technology [61]. Distributed systems and parallel computing relaxed a lot of the hardware complexity in older servers but required more sophisticated software to operate. Example of this software are telecommunication networks, and the Internet. Distributed and parallel systems are still expensive to own and require both physical facilities and knowledge to perform computations on the system. The evolution of communication media and infrastructure has enabled more that can be done through the Internet. This evolution has also encouraged researchers to explore the field of cloud computation.

### 2.1.1 Cloud Computing

Cloud computation is the use of servers accessed across the Internet for storing data and executing computing tasks with varying complexity levels. This computation can be programmed and assigned to dedicated servers or left for the cloud to load balance the work. Cloud load

balancing will assign tasks to virtual machines taking under consideration the complexity of the task as well as the physical location of the task issuer.

Cloud computation is facilitated by service providers who have proposed the Pay-As-You-Go service model. Examples of typical service providers are, Microsoft Azure, AWS, Google and many more. One of the most popular activities that happens on today's Internet is storage which is provided by different vendors through a variety of applications, such as Dropbox, iCloud, Google drive, etc. The user of these services can purchase both storage space and processing power according to their needs. This service model facilitates computations and storage without the need of physical space or high purchase budget to meet the user's quality of service (QoS) and user experience (UE) expectations. The emergence of cloud computing has increased the appetite of many businesses in all fields to move their work to the cloud [15, 5]. Even with an incredible number of computational servers in the cloud, there still remain performance challenges that can impact the user's satisfaction.

Some researchers have suggested altering a cloud's topology as a network into smaller clusters, called cloudlets. This split has been introduced to provide processing units closer to the user. The idea of a cloudlet was first proposed as a two-tier network architecture [5]. The first tier is the well-known cloud and the second tier is the cloudlet which is a small-scale cloud with one-hop network latency [43]. Unlike the cloud, the proposed two-tier model has lower communication latency, which made it suitable for real-time applications like video streaming, online gaming, and voice over Internet Protocol (voIP).

### 2.1.2 IoT, Fog and Edge Computing

Since the birth of Internet of Things (IoT), many applications in various domains have been developed [90]. McKinsey [83] defined IoT as: "Sensors and actuators embedded in physical objects are linked through wired and wireless networks, often using the same Internet Protocol (IP) that connects the Internet." Therefore, any device that has a micro-controller and connection to the Internet could be used in an IoT platform. Devices such as smart sensors, smart phones, tablets, laptops etc. are deployed increasingly day by day as part of IoT platorms.

Cisco proposed a hierarchical architecture *fog computing* which makes use of cloud computing and other network components in the local network that are closer to the user. Specifically, making use of resources of network components that are located at the edge of a network is called *edge computing*. Fog computing and edge computing are often used interchangeably [35] because of the similarity of concepts of using edge devices/data centers. Edge computing and IoT development has facilitated better utilization of computational and storage units to better enhance performance [89].

#### Fog\Edge Network's Components and Characteristics

Fog\Edge networks consist of computation and network devices. The computational devices may range from single-board computers, like Raspberry Pis and commodity products like desk-

top, laptop, smartphones and smart objects, to dedicated micro data centers. Network infrastructure in Fog\Edge configurations typically includes access points (AP), wireless local area networks (WLANs), routers, switches and gateways [35]. The aforementioned network devices with improved computing capabilities represent part of system's resources. One example of using edge device capabilities would be to make a smart surveillance camera do image recognition of the street shot and notify the system only if there is something wrong. It is able to do so if it has enough storage space to save the weights of a trained network. By that, some computation load would be shifted from data centers to smart devices and usage of communication links would decrease. It is worth noting that performance enhancement is the main objective that has driven the development of 5G networks and edge computing. Though 5G networks have increased network bandwidth and performance, they can also leverage edge computing to reduce latency and increase Internet speed [2].

Edge network resources are:

- Heterogeneous: different processing units often have different architectures with different capabilities.
- Dynamic: workload differs as the number of tasks change, rate of new tasks starting or tasks stopping, more users engage with the network, more competition for limited resources.
- Mobile: certain devices may leave the network without any acknowledgement, e.g. end user devices such as phones, even before task running on them are completed.
- Resource limited: edge devices have small computation power compared to data centers.

In the following, we review some of the key concepts, terms and characteristics of Fog/Edge computing.

### Architecture

The architecture of edge/fog computing can be classified depending on many aspects. It could depend on the data flow in the system, or on the type of control of resources in edge computing. There have been many approaches for data aggregation modelling in edge/fog computing. The techniques used are cluster-based techniques [82, 32], graph-based techniques [41, 94], Petri net-based techniques [31], etc. Along with that, researchers have also proposed techniques to improve aggregation for different objectives. They directed their studies towards finding energy/latency/bandwidth-efficient techniques, quality-aware techniques, and security-aware techniques. The other architecture classification method depends on the resources control method. While edge computing should be distributed-controlled unlike cloud environments, still some edge computing models are center-controlled but in a smaller scale than cloud environments. In a centralized-control model, a controller like Software Defined Network (SDN) is used to manage the workload in local network area [36, 92].

### Offloading

In some cases where user demand for cloud resources exceeds capacities of the cloud, applications initially located to the cloud can be offloaded to edge nodes. Similarly, overloaded edge computing nodes might have tasks moved to the cloud to relieve demand on the edge computing nodes. Offloading could also occur to balance load between nodes or process urgent requests faster. Generally, offloading is the shift of workloads between processing entities.

### Virtual Machines and Containers

Typically, many applications are assigned to the same fog/edge node. To ensure isolation and independence of each application in a node, they usually each run as a single virtual machine or container. This virtualized system means that an application is served by limited resources and its failure does not affect other applications running in the same node. A virtual machine emulates a physical computer by virtualizing resources like CPUs, network, memory, storage devices and GPUs [35]. A fixed amount of each resource is specified to a single virtual machine. Generated tasks/applications could be assigned to any of the registered virtual machines located in any node in the network. With the variety types of tasks, heterogeneity of nodes and their different locations, finding the optimal assignment is not an easy problem.

On a lower level of partitioning, a container is another mean of system virtualization, which provides process-level virtualization. Containers can share a single operating system kernel, but applications have isolated environments for execution. Containers generally consume fewer resources than virtual machines.

### Resource Management/Allocation

A resource management algorithm is an algorithm that maps tasks/applications to computing nodes that have at least the minimum resources required to execute those tasks/applications. Additionally, the algorithm should also aim to ensure that there are resources for future requests and in order to avoid over committing tasks to the node. A successful resource management algorithm also aims to keep the system running and enables processing as many requests as possible. Generally, the overall state of the system is more important than the state of a single node or the success of a request. Resource management algorithms can be characterized based on their overall approach to handling resources [53, 28]:

- Provisioning is the act of allotting resources to workloads.
- Allocation is the distribution of resources between competing workloads.
- Modelling is providing a framework that can help predict resource needs for a workload.
- Brokering is defined in [28] to be: ”negotiation of resources through an agent to ensure their availability at the right time to execute the workload.”

- Scheduling is organizing events and resources in a timetable which coordinates workloads' requirements and duration with resources available.

Some workloads require predecessor tasks' results and/or external readings from specific devices. Coordinating workloads' external requirements and resources requirements with current system's available resources is not a trivial job. In this thesis, we focus mainly on having a broker for resource management in our system.

#### Load Balancing

The other important algorithm is the load balancing algorithm. Similar to resource management, load balancing is essential to maintain the performance of large systems. While equality is assigning exact number of requests to each node, a successful load balancing algorithm should ensure fairness in the system. Since requests and nodes are heterogeneous, the processing load capability is not identical for all nodes when assigning same quantity of requests. For example, ten requests could increase the load on node 'A' as much as one request of the same type increases load on node 'B' depending on computational capability of each, i.e., node 'A' could be a much more powerful computing device. Ensuring fairness in system leads to approximately equal waiting times tasks in every nodes' queue.

After laying out the fundamentals of cloud and fog computing, the following section will present some of the proposals regarding load balancing and resource allocation. In fact, some of those researchers focus on one or more characteristics of fog computing. Others, present techniques using limited settings like load balancing between two nodes only.

## 2.2 Related Work on Resource Management

In this section, we review and summarize a number of different approaches for load balancing and resource allocation. Common techniques are presented first, then smart approaches are presented; these latter approaches focus on use of Reinforcement Learning based models, since our research considers this approach as well.

### 2.2.1 Resource Allocation Algorithms

Resource allocation is needed in a hierarchical environment like edge computing, where multiple nodes may be connected to the same node in a higher level. The multiple nodes might use a mutual wired connection to the parent node and thus compete for bandwidth. Taneja, et al. [81], use a directed acyclic graph to achieve efficient resource utilization. In their approach, fog nodes have three attributes, namely, CPU, RAM, and Bandwidth. Every application module requires minimum attributes to run successfully depending on the nature of the task this module does. The researchers developed an algorithm to sort modules and network nodes and then the algorithm matches every module with its best fit available network node. It does this in an iterative manner all the way from fog nodes up to cloud nodes. A shortest-path equivalence design was proposed in [87]. The authors proposed dynamic service placement by predicting the cost

of service placement options to find the optimal service placement sequence. Service is generally defined as long running tasks of the same type [59]. Transmission time, computational delay and migration overhead are the costs the scheme trying to minimize. After finding the optimal look-ahead window size, the researchers use simulation to show that their design can reduce the average cost. Despite the good performance of their approach, the scheme assumed homogeneous services and did not take service's type and prerequisites into consideration.

Focusing on resource allocation, Ni et al. proposed a strategy based on priced, timed Petri nets (PTPN) [60]. This strategy predicts time and price needed for a task's completion and considers the reliability of users and fog resources. They managed to dynamically allocate fog resources by filtering and classifying them based on their computing capability and credibility.

### 2.2.2 Load Balancing Algorithms

*CooLoad*, a cooperative load balancing scheme, was introduced by Beraldi, Mtibaa, and Alnuweiri. In CooLoad, two nodes cooperate together to minimize task delays and blocking probability [13]. Blocking is the event in which a requested task cannot execute or continue execution and, as a result, it gets dropped. The authors built their model as Markov processes and they analyzed the possible scenarios. In their mode, nodes were heterogeneous. Similar to many other studies, after tuning their parameters, results showed that the proposed scheme is better than isolated and fully shared schemes in the perspective of blocking probability and delay. Nevertheless, balancing the load between two nodes cannot be scaled up for fog computing systems.

In [4], authors used the popular NSGA-II multi-objective algorithm in order to optimize time delay and energy consumption in fog and cloud architecture. In [91], authors designed an algorithm to balance the load and allocate resources efficiently between different nodes. Nodes are categorized based on their computing capability, memory storage and bandwidth. Services are also categorized based on their requested computing nodes and predefined start time. Services are partitioned to computing nodes that are classified based on the type of service they provide and predefined start time.

Designing services with predefined starting time does not cope with the nature of service architecture since services are on-demand by nature. In [24], an improved A* algorithm is employed in an SDN controller to select the best fog node for service allocation. This algorithm chooses the node based on the shortest path from a requester that is part of an IoT (thing as a requester) to fog node (executor). It also considers the memory, CPU, and GPU utilization in nodes and bandwidth link utilization. Service requests arrive in a rate that depends on many factors and have certain distribution based on type and location of network. That was not the case in both [91] [24] since they assumed services were all requested at the same time, which makes their models unable to deal with traffic and different workload rates.

A special type of the IoT is the Internet of Vehicles(IoV). IoV is one of the vital aspects of smart city since it is one of the keys to smart traffic. Authors in [33] integrated elements of net-

work to form software defined cloud/fog network (SDCFN) architecture. The purpose of this architecture is to reduce latency and fulfill QoS as they are the most essential criteria in IoV. The vital key to this is load balancing which is done through a modification they made to particle swarm optimization algorithm (MPSO-CO). Simulations showed that SDCFN architecture and the modified algorithm improved the QoS which brings IoV closer to reality.

Security-aware load balancing , a technique that was proposed by authors [67], uses breadth first search (BFS). The heuristic BFS is used to choose the best responder to the request of load sharing sent by the requester. The requester is an Edge Data Center (EDC) that broadcasts an encrypted request when its overloaded. EDCs with sufficient resources to execute the overflowed task respond with encrypted responses back to the requester. The requester then chooses the best candidate to migrate load to. After experimental results, they verified that their proposed model is secure and sustainable.

In [62], authors represented cloud as a set of discrete virtual machines while fog nodes as a single virtual machine each. Ningning et al. have used graph theory to minimize the number of migrating switches. This is done through defining every virtual machine as a node in a graph. Those nodes are connected through weighted edges. The weight of edges represent the bandwidth between the two virtual machines represented by the two nodes connected with that edge. After constructing the model, graph partitioning is used to solve the problem of load balancing in a dynamic structure where some nodes(like fog nodes) can join and leave networks during runtime. Experimental results show that the running time complexity of this method is reduced compared to classical hybrid strategy for load balancing (HDLB).

### 2.2.3 Load Balancing and Resources Allocation Models using Machine Learning

Anna Victoria Oikawa et al. in [7] used supervised learning to achieve load balancing in large systems. The model is trained to pick the best load balancing algorithm based on the characteristics of a given task. The model needs to be trained on all the scenarios possible for it to be realistic, which is not possible. Additionally, the load balancing problem is more dependent on the nodes and their capabilities than tasks' characteristics.

A number of approaches looked towards using reinforcement learning because of the heterogeneity of services and nodes, and because of the dynamic nature of real-life networks. Some of this work took heterogeneity of tasks into consideration when designing resource allocation management models. In [51], the authors divided the workload into three categories based on priority and type (streaming or single time) applications. One important criterion of this model is that streaming applications do not run fully on the one executor, which ensures less idling resource allocation. They used DQN RL (Deep Q Networks based Reinforcement Learning) agent to get all executor resources along with the task profile, to decide where to execute a task. $\epsilon$ is the parameter that controls the behaviour of the agent. Typically, at the beginning it leans toward random behaviour and as the agent learns more about the environment, it changes the agent's behaviour gradually to use its gained experience. While designing the model, $\epsilon$

has to be tailored so that merging with the system is done smoothly in multiple steps. Nevertheless, the executors in this design are assumed to be identical which is far away from real environments.

In the Mobile Edge Computing (MEC) system [47], the authors assumed having a number of user equipment nodes (UEs), MEC servers and eNodeB - an entity that is the connector between all UEs and the server. Each task is characterized by three properties, the size of computation input data needs, number of CPU cycles required to be executed and maximum delay tolerance. They designed a RL model that decides whether the user should execute the task locally or offload it to the server and its reward is negatively correlated to the net cost of the system. As the cost of the system increases the reward given to agent decreases. When the number of UEs increase this design will no longer be efficient because of the traffic congestion and server overload. In [57], a RL model is designed to deal with resource allocation in case of emergencies. It is assumed in [57] that there is a predefined priorities for IoT devices that have shared bandwidth. In case of emergency in a location, the device located in that location will have higher bandwidth in order to transmit high quality video streaming. That is accomplished by choosing one device that has a lower priority and choosing it as a victim. Bandwidth is taken from victim and is given to the higher priority one.

Authors of [58] assume mobile and stationary distributed fog nodes among fog computing regions. Each region has a fog computing domain controller, which is responsible for redirecting tasks coming from IoT devices connected to this domain (network). The Deep Q Networks (DQN) which is type of deep RL, takes as an input the location of the requester and fog nodes, current fog nodes connected, and task specifications as the state. The DQN then decides where to execute this task (fog nodes or cloud). In this design, nodes can join or leave during execution and those nodes could have different computing capabilities and storage capacities. The proposed solution outperforms the random node, nearest node, and optimal node algorithms. The results are good, but the tasks are assumed to be identical, and their dependencies are neglected. This makes the problem much easier than what it is in reality since homogeneity and task independence simplifies the problem.

Load balancing has been a long-term issue in all distributed large systems. That is why various of designs and algorithms have been proposed to fairly distribute workloads among entities. Some researchers tend to solve this problem at the network level where they can control the utility of a link and migrate connections to switches from one node to another or one region to another. Other studies have focused on the device level, where their approaches depend on the computational capabilities and workload properties of individual nodes. The authors of [44] use a method based on deep belief networks to balance the load in a network of IoT. More specifically, they combined Q-Learning with neural prior ensemble to come up with new balancing technique. Load-bot and Balance-bot are two agents that are designed in order to measure network load and process neural load prediction respectively. The model is compared with the dynamic scheme and the experimental results show that the number of migration of nodes and regions decreased.

In a more specific design, Li et al. in [48] aim to balance the load between SDN controllers. The

assumption is that every controller has different loading capacity. In other words, controllers are heterogeneous, an assumption that introduces more complexity to the system. An RL agent is designed to choose the switch that when migrating tasks to would balance the load significantly and would cost less. This design outperforms the compared methods of selecting the one with highest transmission rate and selecting the one with minimum migration cost. Performance measurement is the load on the controller plane and the migration overhead.

Authors of [11] use load index, which corresponds to workload on a node based on a global average, as a comparison key. This key is flagged when the CPU length of a node deviates from the load of other nodes in the same fog network. The RL agent in SDN controller is responsible for monitoring the states of the nodes and ensuring to lower the probability for nodes to reach overload state. With the rate variation of task arrivals, the performance is compared with least queue node method, nearest node selection method, and random selection method for offloading and resulted in lower cost.

For the sake of leveraging, other researchers use deep reinforcement learning. Although it takes longer time to train such models, its performance is worth it. In [6], they balanced the load in MEC network that consists of access-points, mobile devices (MDs) and multi-edge servers. The decision of where to execute a task is taken in MDs, where they have four options, either to execute locally, offload to a near server, offload to an adjacent server or offload to the cloud. One desirable feature in this study is the assumption of heterogeneity of nodes (MDs, servers, cloud). Using SARSA algorithm to determine the suitable action for current state of resources (processing space, memory, and bandwidth) resulted in significant performance. The agent was to be rewarded if response time in this time step is smaller than previous time step.

The authors of [49] designed RILNET which is an approach to balance the load among switches/ edges by finding the best distribution for path's utilities for pair of nodes. Deep RL, specifically the actor critic method using four neural networks is used in a centered-control architecture for its design. Agents in this case receive snapshots of links throughput in the network and determine the best proportions of usage for available paths. The error rate is calculated with the reference of optimal MLU (Maximal Link Utilization). Optimal MLU divides flow into equal-cost paths. RILNET almost reaches the performance of optimal MLU and it outperforms EMCP (standard scheme for flow-level). On the packet-level, RILNET outperforms DRILL (standard scheme for packet-level) by routing the data packets based on flowlets with low probability of out-of-order.

None of the previous models combine the different capabilities of nodes, task dependencies considering time delays and energy consumption. Some of them create constrained environments which makes the training successful, but they fall short when deployed in more complex environments. We propose an approach that encompasses greater heterogeneity of nodes and variability among tasks.

# Chapter 3

# Preliminaries

One of the most popular application domains that was born with the maturity of Fog and Edge Computing was smart cities. In this chapter we will review various definitions of smart cities, outline some of the characteristics and discuss some of its challenges. The second part of this chapter will lay out the essence of reinforcement learning that will be used to solve one of smart city system's problems.

## 3.1 Smart Cities

The evolution of IoT has encouraged researchers to work on novel ideas in many research fields. Smart city is one of them. The concept of a smart city has been defined in a variety of ways. One general definition for smart city was pesented in [88]. They defined a smart city as any municipality that relies on communication technology to increase its operational efficiency. This municipality also shares information with the public to better enhance the citizen's satisfaction for government services. Researchers believe that the popularity of smart cities will increase in the next few decades. Subsection 3.1.1 details some of the research work that shows the trends and reasons for the growing popularity of smart cities.

### 3.1.1 Necessity of Smart Cities

The concept of the smart city is essential because of its direct effect on the quality of lifestyle of residents of cities all around the world. Smart cities are predicted to embody 68% of world's population in 30 years. As Figure 3.1 shows, in 2017 an average of 81.71% of people live in urban areas in both Canada and the United States. Figure 3.2 presents the statistics of the population living in urban areas in each continent in 2015. Figures 3.1 and 3.2 show that cities' popularity is increasing all over the world. The continuously increasing trend in Figure 3.1 correlates to the workload on cities institutions, and technology support has never been more required. Technological systems deployment helps cities to function on cost effective, more accurate, and more environment friendly models. Hence, the research in the area of smart cities is now seen as a vital and basic necessity.

The "smart" concept can be integrated in many areas such as smart government, smart utilities,

smart mobility, smart buildings, smart environment, etc. The potential in smart city technology has already attracted investment from governments, industries and individuals. The growing attention to smart cities technology needs to be coupled with more efficient and feasible models to overcome the burdens and challenges. Those models bring more smart cities deployments to life.

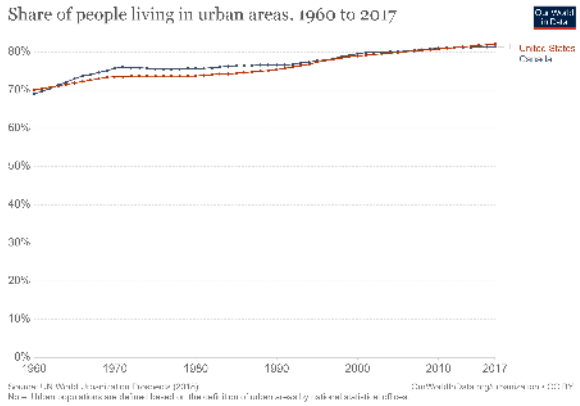


Figure 3.1: Ratio of people living in urban areas from 1960 to 2017 in both Canada and United States [72].

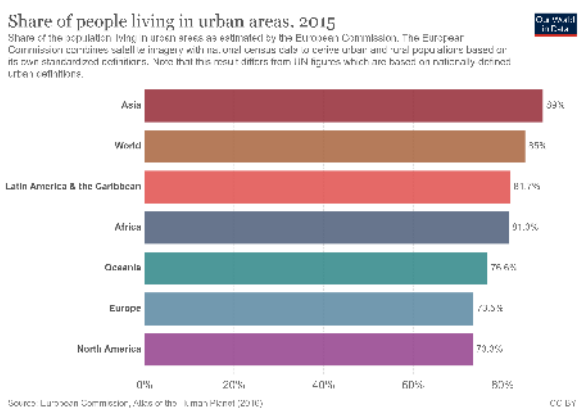


Figure 3.2: Ratio of population living in in urban areas in different continents in 2015 [72].

### 3.1.2 Challenges and Requirements

Smart city project deployment inherently possesses both physical and network challenges [71]. Physical challenges come from device installation and deployments, especially if the installation happens in areas that are not owned by the government, such as private properties. Challenges associated with the networks arise due to the complexity, redundancy and heterogeneity of the network software and hardware components. One of the most crucial and challenging aspects of these networks is security, since smart city networks may be accessible by a wide variety of users in very large urban areas. Another challenging problem is the demand for continuous maintenance for smart city systems and sensors around the city. Collected big data and the connectivity of things brings the concept of profiling to attention. Profiling is the ability to collect enough information about customers and users to create a profile. Tracking people by following their usage of public transportation is another issue that invades citizens privacy. To add to the complexity of the designs and solutions, smart city services that are provided through these networks often have to meet time, energy and cost constraints [65].

Generally, a successful smart city project involves highly interconnected devices, has to be energy efficient and cost effective. Last but not least, it has to be highly reliable [65]. In order to reduce the cost and improve reliability, it is desired for the system to be highly autonomous. The system should possess the ability to manage, monitor and properly respond to situations with minimal human interaction. One key characteristic that would contribute to the efficiency and success of the system is to be proactive.

One innovative approach to build a system with the mentioned desired characteristics and capabilities is to design the system using artificial intelligence. Artificial intelligence can help researchers to meet some of the key characteristics such as autonomy, proactivity and cost efficiency.

### 3.1.3 Smart City Projects in Progress

There have been many smart city projects at a variety of scales and for a variety of purposes. Some projects are partially deployed for evaluation and developments. Other are more fully developed around a single application.

The Padova smart city (PSC) [19] initiative was one project that addressed some of the challenges. The fact that devices are connected through different communication protocols, like CoAP, MQTT, XMPP, etc. [38], makes it complicated to find a standard way for communicating in one platform. PSC project tackled this problem along with mixing network protocols IPv4 and IPv6 in order to better generalize the platform [71].

Another smart city initiative is located in Chicago. The city installed hundreds of sensors around the city to collect data in order to be able to make smarter decisions regarding traffic, air quality, temperature around the city and more. Which made them the most data-driven government in the world [55]. The Array of Things (AoT) is a project that depends highly on the engagement between the government, academic institutions, and the citizens of Chicago, to collect real-time data of city's environment, and infrastructure for research purposes and for the public [66]. Chicagoans are cooperating with the government by continuously giving their feedback [20].

Barcelona embraced the idea of a smart city when the concept was first emerging. Since then, the Barcelona government has deployed nearly 20k sensors [29]. As a result, they are saving up to $37 million each year from smart energy savings. Connected citizens is one the ways that can make a city smart. Barcelona has increased the number of WiFi hotspots around the city to ensure that the city has almost full coverage. After that, they created a series of applications that help citizens better use the city's services.

The city of Curitiba in Brazil started boosting the communication and relationships with its citizens by responding to citizens' inquiries in the 1980's. The city established the "156 Central" hotline then to allow citizens to report any urban infrastructure damage, demand actions, like waste collection, inquire about bus routes, and much more. They have also started collecting data from citizens by holding interviews [34]. Today, the city is working with manufacturing companies like Volvo and SAAB to ensure sustainability for new technological systems.

Other smart city projects have focused on identifying the strengths and weaknesses of huge cities like the ones in India, as well as finding possible opportunities for each city in the country [43]. Extracting information and producing analysis of strengths and weaknesses of a city is one of the important applications to properly identify possible opportunities for improving the infrastructure, citizens' quality of life, and management cost of that city. Singapore,

London, and a number of cities in North America are already deploying ”smartness” in their architecture, and infrastructure [30].

### 3.1.4 Layers of Smart City

Most of research work on smart cities shares a number of common basic elements that can be divided into four layers:

1. The **Device Layer** is where all sensors gather data in the platform. Some of the devices, such as actuators, perform actions when required that may determined by other layers in the architecture. Sensors on the other hand acquire data and feed it to the infrastructure layer. The device layer depends on wired and wireless networks entailing different network protocols.

2. The **Infrastructure Layer** is where aggregated data flows into computing devices with greater capabilities than sensors and actuators, such as computers, servers, mobile devices, etc. These computational elements filter, transform, store, analyze and process data.

3. The **Middleware Layer** is where all data filtering and processing take place and often where automatic decisions are taken [65]. It is the core of the platform.

4. The **Stakeholder/Application Layer** is where applications are contained. Applications utilize the processed data from the middleware layer. It includes analysis, monitoring systems, decision making agents, and other high level applications.

Challenges in both the applications arising in smart cities and the networked infrastructure make it harder to optimize this complicated system with the interdependence between the layers in a smart city platform. This optimization problem could be addressed by formal and informal methods. Formal methods for solving optimization problems are formulated in mathematical algorithms which try to guarantee global optimal solutions. In many occasions, they take a very long time to reach the desired global solution. This makes it inconvenient for quite a number of real-time applications. Dynamic programming, ant colony optimization and annealing imitation are examples of these formal optimization methods [23]. On the other hand, informal methods can find close to optimal solutions [45]. The vast majority of real-time systems rely on this paradigm because the time it takes generally takes less time and fewer resources to reach a usable solution.

The problem is addressed by Artificial Intelligence approaches rather than formal optimization approaches because of the ability of the AI approaches to fit with the continuously growing network and dynamic nature of the system. Moreover, with AI approach, additional models can be integrated, like prediction models and more.

This section illustrated the importance of smart cities and presented some of the challenges and requirements of a smart city, as well as the need for heuristic methods to meet constraints of smart city management models. In the next section, we look at approaches to address some

of these problems based on Artificial Intelligence, particularly machine learning, including Reinforcement Learning, which is used in this thesis.

## 3.2 Artificial Intelligence

Artificial intelligence (AI) is a sub-field of computer science in which algorithms and systems are developed to enable a machine to perform tasks that mimic human intelligence. Within Artificial Intelligence there has been substantial research done on learning methods - Machine learning (ML). ML is a field in AI that aims to develop algorithms and methods that can identify patterns out of usually immense amounts of data. Machine learning is performed by iteratively introducing training sets until predetermined objective function is achieved [10].

ML is classified into Supervised, Unsupervised and Reinforced Learning. Supervised Learning depends on human intervention where the target and actual outputs are analyzed. The machine learns how to map the introduced inputs to the target outputs by adjusting internal factors. Basic Supervised Learning algorithms include Linear Regression [76], Naive Bayes Classifier (NBC), [10, 16] and Support Vector Machines (SVM) [64].

Unsupervised Learning, unlike the Supervised Learning, does not attempt to map inputs to target outputs. Rather, it aims to identify patterns in input sets. Since this learning type lacks mapping and actual/target outputs, this type lacks the ability to validate the machine outputs during learning. Clustering data [84] and dimensionality reduction [85] are two Unsupervised Learning methods.

One of the most appealing machine learning techniques is Reinforcement Learning (RL). While some research considers it as a type of Supervised Learning [14], RL is different from both traditional Supervised and Unsupervised Learning in the availability of training data. While in supervised and unsupervised training, the input data is mostly available, in RL only partial data is introduced to the machine. A reinforcement learning based agent learns by trial and error aiming to maximize a numerical value. The term agent is used instead of machine because it interacts with the environment and it has an impact on it. The learning process is guided by the reward which assesses the performance of the agent. In Supervised Learning, the machine is taught by setting examples, while in reinforcement learning the agent is taught by experience. Although both Unsupervised Learning and Reinforcement Learning share non-supervision learning process, Reinforcement Learning has a reward which is considered as a performance measure [52].

### 3.2.1 Reinforcement Learning

Generally, RL techniques depend on many factors, e.g. whether there is model or not, type of skill to be learned, etc. It is worth noting that it is applicable to a large number of domains like healthcare, trading, IoT, etc. RL is the only machine learning method that uses explorations and exploitation trade-off. While Supervised and Unsupervised Learning involve

some of machine learning challenges (generalization, exploration, delayed consequences, and optimization), dealing with all of them is applicable in RL methods.

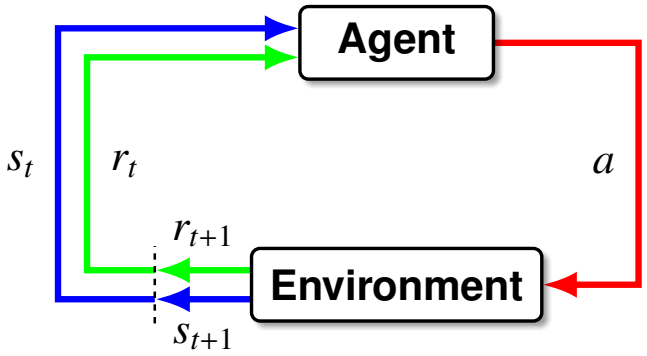


Figure 3.3: Reinforcement learning model cycle with input and output signals [77].

To understand the methodology of RL, several theoretical concepts must be introduced first. An abstract overview of RL is shown in Figure 3.3. There is an interactive closed loop, where an agent produces an action that affects the environment and receives an observation and a scalar reward. The environment is the surrounding system and context in which an agent operates, while the model is the agent's internal representation of the environment. Environments are either fully or partially observable; the agent state is not the environment state.

An action is defined as any effect the agent can do to the state or the environment. Which means, an action could change the environment in the agent's current perspective e.g. walking changes the state of the agent, or changes the environment itself e.g. changing room temperature changes the environment.

No matter what the technique used or what the application is, all RL agents share the same goal and that is maximizing expected cumulative reward. Reward is a scalar performance measure that indicates how well the agent is doing and guides the agent towards desired priorities. It is a way to evaluate the actions taken, which impact takes place immediately or later.

In RL, the decision making process is represented mathematically by a Markovian Decision Process (MDP). MDP is decision making model in discrete, stochastic and sequential environment [50]. It is defined by a set of states (S), a set of actions (A) and a transition probability kernel ($P_0$) [77]:

$$M = \langle S, A, P_0 \rangle \tag{3.1}$$

The basic rule in MDP states that the future is independent of the past given the present. Therefore, the current observation is as helpful as the history of observations. This means that it assumes that it is not beneficial to keep records of how the agent got to its present state. One exception is when states are limited, as the case in partially observable environments, the agent then uses history to derive helpful information about the environment.

The dynamics of a model is determined by the transition probability kernel which distributes the probability of transitioning from one state to another. Whereas the probability of taking a certain action in a state is defined by policy. Policy represents the behavior of an agent; it is the mapping between states and actions. From experience, the agent learns the optimal policy that could be deterministic or stochastic. A deterministic policy is one where the agent has one option to take in a single state,

$$\pi(s) = a, \tag{3.2}$$

Whereas the stochastic policy assigns probability for action candidates for each state,

$$\pi(a|s) = P(A = a \mid S = s) \tag{3.3}$$

where it directs the agent to the best action given the current state. An optimal policy is the one that maximizes the value function in all states.

To take the best choices in every state, agents tries to approximate future rewards,

$$G_t = r_t + \lambda r_{t+1} + \lambda^2 r_{t+2} + \lambda^3 r_{t+3} + ... \tag{3.4}$$

The return at each step $G_t$ is the sum of current reward and future rewards which are discounted by a discount factor $\lambda$, which is bounded by $0 \leqslant \lambda \leqslant 1$, for mathematical convenience. Future rewards are predicted by a value function, which determines the value for being in a state and/ or taking an action. For the agent to acquire more rewards, it uses the value function to measure potential future rewards [42].

$$V_\pi(s) = E_\pi[r_t + \lambda r_{t+1} + \lambda^2 r_{t+2} + ... \mid S_t = s], \tag{3.5}$$

$$V_\pi(s) = E_\pi[G_t \mid S_t = s], \tag{3.6}$$

where the value function is the expected sum of future rewards in the current state given the policy. It predicts how much reward the agent will get from the next state till the terminal state.

The discount factor $\gamma$ makes the value function finite. If the objective is to care about far sighted goal then $\gamma$ should be close or equal to 1, and if objective does not require future planning then $\gamma$ should be close or equal to 0. It also depends on the agent's belief of the model and how well the agent can predict. A discounted factor is needed because the future is founded on predictions and because the environment is stochastic.

To clarify the difference between reward and value, reward is the immediate measure the agent receives in a given state while the value is the long-term reward the agent might receive from being in a state [3]. Future rewards are expected because of the nature of the model.

The value of being in a state $v_s$ gives a good evaluation of the agent's current state. In other designs, the evaluation of state-action pair gives a better evaluation than $v_s$. This pair gives better indication of performance than the evaluation of a state and an action independently. For example, jumping (action) when being in front of a barrier (state) makes more sense and gives better evaluation of this action in that state, than giving value for jumping regardless the state.

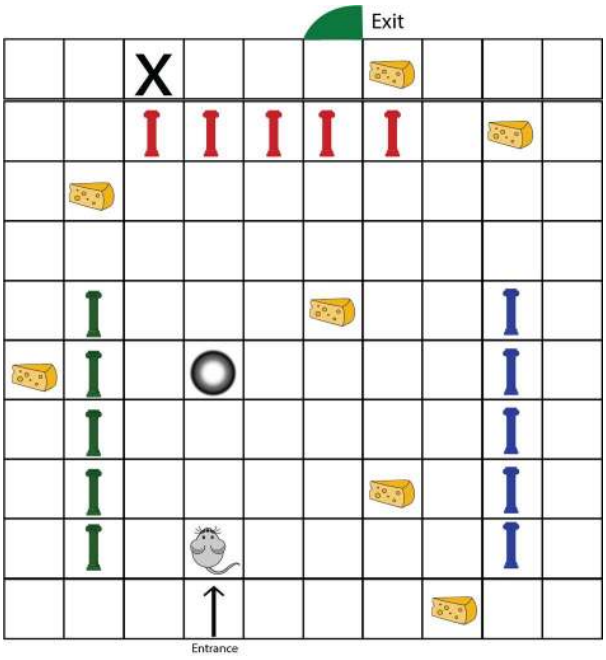


Figure 3.4: 10x10 maze as an environment for the rat.

The value of associating actions and states is described by the Q value. The following equation represent the Q value function that defines how good a state-action pair is, when following policy $\pi$.

$$Q_\pi(s,a) = E_\pi[G_t \mid S_t = s, A_t = t] \tag{3.7}$$

The agent will keep on learning until it finds an optimal policy (or near-optimal policy). Finding the optimal policy means that the agent is acting in the best behaviour to get most rewards possible in all states. There always exist at least one optimal policy. When finding an optimal policy, which existence is proved [75], the optimal value function and the optimal Q value function are achieved accordingly. Reaching optimal policy means having the best behaviour possible which is derived from the best evaluations of states (optimal value function).

The following example will help make the previously mentioned elements of RL a little clearer. Consider a rat in a 10x10m maze; we assume that the ability of the rat is limited for simplicity. The rat can either walk in one of the four directions or eat a piece of cheese; these represent the set of possible *actions*. Eating the cheese will make the rat satisfied while leaving the maze will make it free, both are considered as *rewards*. The maze, which represents the *environment*, consists of distributed colored pillars (some of them are blue, red, and green), a tile marked X, tiles with pieces of cheese, and one hole. The red pillars exist only around the exit which provides a significant hint for winning the game, but the rats cannot use this helpful information, since rats are green-red blind [22] and so their *model* of the environment consists of blue and green pillars only.

In this example, the rat is the *agent* and its constrained abilities represent the action space. The *state* of the rat represents what the rat observes at each step. Reaching the state marked X does

not provide the agent any reward since it did not eat cheese, nor did it exit the maze, still it gives a very good impression. That is because of its previous experience of winning shortly after observing an X marked tile. This illustrates the difference between *rewards* and *value function*. The value function gives a high value for this state because it learned by repetition that in the upcoming few steps, the rat will leave the maze which will return a very big reward. Whereas the *policy* defines the behavior of the rat since at any step the policy tells the rat which action to take. In a tile where there is a cheese, the optimal policy will assign the highest probability for the eating action. Both policy and value function will be progressed after each episode.

Generally, an *episode* is a series of states, which starts with resetting the reward and the current state of the agent and ends in any terminal state. *A terminal state* is a state in which any action taken does not change the state nor gains a reward. In our example, the episode terminates when reaching the exit or falling in the hole.

If the desired scenario is getting the rat to leave the maze using the shortest path, then there are many adjustments that could be made to the design. First, a punishment, which is a negative reward, could be assigned to all tiles but the terminal. Second, the rat should plan better to avoid unnecessary movements. That is done by increasing the *discount factor* in order to care more about long-term rewards.

From another perspective, if the goal was the survival of the rat, then the design is *value function* based. This means that the value function should give the states around the hole a very low value in order to avoid being trapped in a hole.

*Exploring* is the task of taking random actions for the sake of discovering the environment. In contrast, using previous knowledge of the environment is called *exploiting*. The balance between exploring and exploiting is attained by using a greedy epsilon policy.

$$behavior = \begin{cases} explore, & \text{if } r < \epsilon. \\ exploit, & \text{Otherwise.} \end{cases} \tag{3.8}$$

Here, $r$ is a random number between 0 and 1. If $r$ is less than $\epsilon$ then the agent takes a random action to explore the environment, otherwise the agent uses its experience to exploit the environment. At the starting episode, $\epsilon$ should be 1 and after each episode, $\epsilon$ is decayed by a decay factor. When $\epsilon$ reaches 0, then the agent should be experienced with the environment and does not need more exploring. The decay factor should be small to allow the agent to explore many possible paths; this helps the agent avoid getting trapped in a good yet not the best path.

### Dynamic Programming

Dynamic programming in reinforcement learning refers to optimizing a policy for a given perfect model with a sequential components property [75, 77]. Although dynamic programming is computationally heavy and having a perfect model of an environment is not common, it sets basic fundamentals for understanding other methods. It solves for an optimal value function

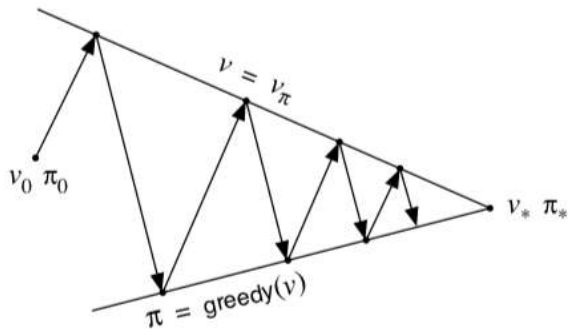


Figure 3.5: Generalized policy iteration optimally approximation.

and optimal policy by a algorithm called Generalized Policy Iteration (GPI). This algorithm starts with a random value function estimates for all states and a random policy. First, it evaluates the value of each state with respect to the initialized policy. Then, it improves the policy by those new evaluations. Basically, it evaluates the policy then improves it. It repeats those two steps until it reaches optimal value function and optimal policy. This method, however, is not applicable to models with huge state spaces, since it needs to update the whole state space value estimates at every step and so becomes computationally expensive.

**Model Free Methods**

Model free methods are useful when the dynamics of the model is unknown or they are too complicated to formulate for an agent. An example of a model free method is flying a helicopter, where it is complicated to provide all the aerodynamics and physics of gravity and balance. Which means we no longer have the transition function that tells us the reward *r* and the next state *s'* if the agent took action *a* in state *s*.

Monte Carlo is one of the model-free methods which depends ultimately on experience to update states' value estimates. More specifically, it is an episode based method. It depends on sampled experience (sampled episodes). It stores all the states and actions taken through out the experience until the episode terminates. Once it does, it updates the value of the states it has been through based on average of returns *G* from that state till the end of the episode. The return $G_t$ is the sum of current and future discounted rewards at step t. Both Monte Carlo and Temporal-Difference Learning (see below) methods' goal is to learn the optimal policy online which means to learn directly from episodes of experience [75].

Temporal Difference Learning (TD) is another model-free method. Unlike the Monte Carlo method, it does not wait for the end of the episode to update the states' values, it uses incomplete returns (incomplete episodes). TD(0) depends on a one step look ahead technique. One advantage of TD is that it can be used in continuous (non-terminating) environments where there are no episodes. One disadvantage is that it is biased because updates depend on one action, transition,and reward i.e. one step experience. To reduce this bias, the update could

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state,\cdot) = 0$
Repeat (for each episode):
  Initialize $S$
  Repeat (for each step of episode):
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until $S$ is terminal

Figure 3.6: Q Learning pseudo code [77].

depend on more than one step ahead like the case in TD(n).

Q-Learning is one of the TD algorithms that balances exploration and exploitation. It focuses on evaluating the value of the Q function instead of the value function of the state. It is an off-policy algorithm, which means the policy followed to choose state-action pair is not the same policy being updated after each time step. One of the reasons motivating off-policy learning is to learn about optimal policy $\pi(a|s)$ while following exploratory policy $\mu(a|s)$. Always choosing the action that will return maximum Q value is an example of an optimal policy and $\epsilon$ greedy is an example of exploratory policy. This method is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[r_{t+1} + \gamma\, max_a\, Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{3.9}$$

where $Q(S_t, A_t)$ is the Q value of the state-action pair of state at step t and action taken at step t, $\alpha$ is the learning rate, $r_{t+1}$ is the reward the agent receives for taking action $A_t$ in state $S_t$ [77]. The discount factor $\gamma$ weakens the value of the future rewards to help the agent focus on the current reward.

As shown in figure 3.6, the first step in this method is to initialize Q value of the dot product of all states in the state space and all actions in the action space arbitrarily (initializing them to zero is also an option in other sources [78, 68]) i.e initializing the Q table. The Q value of the terminal state/states are set to zero, regardless what action is associated with them. For each episode, the environment should be reset to get the beginning state. For each step in the episode, the agent does the following sub-steps in order. First, the agent chooses an action given its state and based on the policy. This policy $\mu(a|s)$ is followed to choose actions and accordingly the pair Q(S,A) to be updated 3.8. The next step is to execute this action in the environment and observe the reward and the new state S' based on the action taken. Then, the agent adds the reward acquired and the difference between this Q value and next Q value based on the observed new state. Choosing the next Q value is based on the policy $\pi(a|s)$ which is in this example,

$$\pi(a|s) = max_a Q(S', a) \tag{3.10}$$

The current state is then updated to the new state. This is done for each step and until the agent reaches terminal state. Therefore, one of Q table's entries is updated at each step. The

agent keeps on repeating in episodic fashion until convergence. Convergence is met when the difference between Q value $Q(S_t, A_t)$ and the updated Q value $Q(S_{t+1}, A_{t+1})$ for all Q values is less than a small positive number.

Q-Learning is the chosen method for our proposed model. There are some modifications added to tailor this method to our problem.

# Chapter 4

# Environment

In this chapter, the model of the smart city environment and its dynamics are mathematically described and the design of the simulator is presented. The simulator is implemented in Python and used to test and validate the algorithms (Chapter 5).

## 4.1 Simulators

There have been a number of efforts on developing IoT simulators in the past ten years [17, 26, 56, 40]. The focus of much of the research and algorithms around Cloud Computing, Edge Computing, and IoT has been directed towards developing the network's architecture and its dataflow. Researchers make use of simulators since it is often difficult to access physical servers and data centers to validate and evaluate their proposals. The CloudSim project [17] originated to simulate cloud computing environments and has been the core of many other projects, including as part of an IoT simulator with the basic elements of cloud computing [17]. Other extensions of CloudSim have emerged, each focusing on developing one feature to aid solving certain problems. Some of those problems included adding heterogeneous nodes to edge computing computing environments, provisioning VM/container technologies, incorporate security-related issues, and orchestrating services/tasks in the system.

IoTNetSim is another Java-based platform that provide IoT end-to-end services [73]. It allows researchers to create heterogeneous nodes in cloud, fog, and edge layers with details like mobility, energy, etc. It also supports different connectivity and network models. This project focuses mainly on applications of IoT systems. The platform supports privacy and security in nodes and during data transfer.

Even though there exist a number of simulators that tackle different problems, none of them have all the basic elements and different characteristics of cloud, fog, and edge computing joined [54]. Based on a review and experimentation with some of the existing simulators, we chose to design a simulator that is tailored to have smart city characteristics relevant to our work. In IoT systems, many tasks are generated from edge devices where users originate requests or activities. Those tasks can be executed by any processing entity connected to the same network where the requester is. In a smart city, the management system could request

tasks to monitor or control IoT devices deployed in the city. Those tasks are typically generated by the cloud. None of the proposed simulators have features for generating tasks exclusively for management purposes. Those tasks have dependencies that should be taken into consideration when allocating tasks in order to ensure their execution and to minimize delays.

## 4.2 Simulator Characteristics

In this subsection, elements and dynamics of a smart city model is presented and this forms the basis for the simulator. The infrastructure of this model can be categorized into the following layers: cloud layer, edge data center layer, edge devices layer, and IoT devices layer. Both cloud and edge data center layers have enormous processing entities and other resources. Since generated tasks have dependencies that require real-time data from the IoT devices layer, communication delay is an important consideration. The devices in the edge layer are generally the closest devices to data collection from sensors in the IoT layer and with processing capabilities they are the entities to execute tasks with real-time or time-sensitive constraints.

### 4.2.1 Nodes

The building block of this simulator is the node. We assume that it has communication capabilities that enable it to connect to other nodes. Each node has its own unique ID which indicates its type in the simulator. The following types of nodes are included in the simulator.

- A *Cloud node* represents the computational facilities in data centers with resources-rich servers that can handle computationally expensive tasks. However, communication delay plays a great role in dropping many delay sensitive tasks due to cloud's multi-hops distance from user layer.

- *Edge Data Center* is a cloud-like node that is closer to user layer. It is a resources-rich node compared to edge devices, but we assume that it has less computational capabilities than a cloud node. With edge data centers nearer to sensors and edge devices, edge data centers can serve delay-sensitive tasks successfully if they are not loaded.

- *Broker* is an edge data center which runs a management system. This system would include the management systems, including the monitoring components and the algorithms for resource allocation of tasks. Designing an intelligent agent in the broker for orchestrating tasks requires monitoring of node operations.

- *Edge Device* is the closest computing node to the IoT layer which is best suitable for delay-sensitive tasks. Unlike cloud and edge data centers, it is resources-constrained node which may fail in executing computationally heavy tasks. Edge devices are available in abundance, and are heterogeneous in capabilities and characteristics.

- *IoT device* is a device that is deployed on sidewalks, in buildings, in subways and around the city. IoT devices like sensors and actuators collect real-time data and execute actions on the environment as ordered by management system.

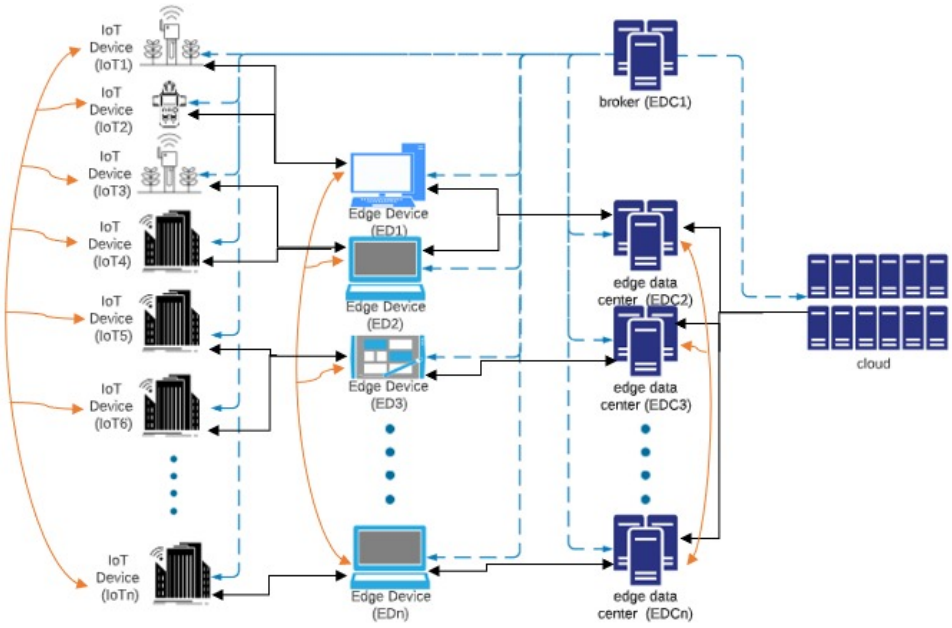


Figure 4.1: System architecture.

The set of nodes N consist of nodes of different types.

$$N = N_C \cup N_{EDC} \cup N_{ED} \cup N_{IoT} \tag{4.1}$$

where $N_C$,$N_{EDC}$,$N_{ED}$, and $N_{IoT}$ are sets of cloud nodes, edge data center nodes, edge device nodes, and IoT device nodes respectively. Nodes are connected by a link, which represents wired or wireless connection between nodes.

$$L = l_1, l_2, l_3, ..., l_m \tag{4.2}$$

$$where, l_1 =< n_i, n_j, B > \tag{4.3}$$

where the set of links $L$ in the system has $m$ number of links and a link $l_i$ represents a pair of nodes $n_i$ and $n_j$ connected in the network with bandwidth $B$. Each link has a bandwidth $B$ which is used to calculate communication delays.

After introducing these two basic elements in the environment, the environment of a smart city system can be defined as a pair of nodes and links:

$$E =< N, L, b > \tag{4.4}$$

where, $N$ is the set of nodes, $L$ is a set of links, and $b$ is a broker, it is a node in $N$.

Figure 4.1 graphically illustrates a smart city environment. The cloud node is at the rightmost in the Figure and is connected to a set of edge data centers. Edge data centers, which are in the second right column, represent distributed data centers that are connected to each other, to the cloud, and to the lower layer (edge devices layer). The same applies to the remaining

two layers (edge devices and IoT devices). We are assuming that cloud node is not connected directly to devices in the IoT layer, instead it is connected by at least one intermediate node. Edge devices and edge data centers can be described as intermediate nodes. The broker which is located in the edge data center layer, is connected to all nodes. There is one broker which is responsible of one region which represents one city. As this is an initial model. we have only assumed a single broker, but the model could be scaled to have multiple networks and multiple brokers in a city. This is beyond the scope of this research.

### 4.2.2 Tasks

Tasks are requests that are generated by end users, by applications and by the management layer and require resources to be executed. Each type of the the aforementioned nodes, except the ones in the IoT layer, is suitable to execute tasks with specific characteristics. Tasks have the following characteristics:

- **Maximum delay tolerance** is the maximum delay this task can face and still successfully execute. Critical tasks which need to be executed in a short time, have low maximum delay tolerance. This time starts when the task is generated. This task need to be fully executed before this delay is exceeded to ensure successful execution.
- **Length** is the number of instruction that should be executed to finish this task. Time to execute a task depends on task's length and MIPS of the executor.
- **Min PE** is the minimum number of processing entities required to execute this task.
- **Min storage** is the minimum storage size required for this task or its results to be saved in a virtual machine; it is in GBs.
- **Min RAM** is the minimum RAM memory size (in GBs) required for this task to execute.
- **Dependencies** is a list of IoT the devices that must be read, i.e., to get data from the device, required for this task to execute.
- **Steadiness** indicates if this task requires multiple readings from its dependent devices.

Tasks with certain characteristics become important in allocation:

- A Computationally heavy task is one where the task execution's length exceeds a threshold in the projected number of instructions to execute; in our simulation this is set at 5 billion instructions. Continuous tasks are considered to be computationally heavy tasks as well.
- Delay-sensitive task is a task where the maximum delay tolerance for the task falls below a threshold; in our simulation this is set at 2 seconds.

Computationally heavy tasks are best allocated in cloud or edge data centers, because both nodes have multiple virtual machines (VMs) and occupying some of them for a relatively

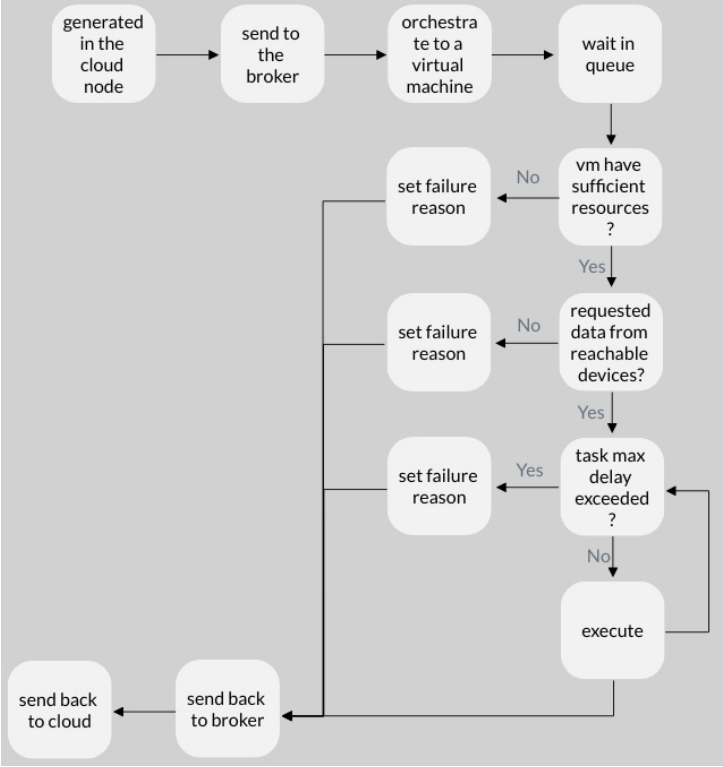


Figure 4.2: Task timeline

long time does not affect the node's performance. On the contrary, allocating those tasks to nodes with relatively limited resources, like edge devices, might result in failure to execute such tasks because of insufficient resources. It will also increase the chance of dropping delay sensitive tasks that are best allocated in these nodes. Delay sensitive tasks are easily dropped if task lifetime exceeds its maximum delay tolerance. That is why many delay-sensitive tasks fail to execute in nodes with high communication delay when reaching IoT devices for task dependencies if any.

There are mainly two types of delays that are created by tasks. First is the input-output delays, which is the time needed to get its dependencies. Along with that, there may be unexpected network traffic while requesting dependencies' readings that will lead to extra delay. The second type of delay is the time needed to execute this task i.e. computation delay. Although tasks can be generated by any node connected to the system, we are focusing on tasks that are generated by the cloud node only. Tasks may be generated to control and manage aspects of the system as well as to carry out computations on data from sensors, filtering, etc.. Other tasks may be generated for data inquiry, monitor, and operate IoT devices. Some tasks are complex and have many dependencies and requirements that needs to be fulfilled prior to execution. We are considering that all types of tasks are generated by a cloud node, yet it is possible for our system to handle tasks generated in any connected node. Then, nodes may execute this task locally or forward it to the broker to orchestrate it to the appropriate node where it will be executed.

As shown in the previous timeline (Figure 4.2), task's life starts when it is generated in the cloud node. Exactly then, the timer for maximum delay tolerance begins the "ticktock". After generation, the task is immediately sent to the broker. Then, the broker will orchestrate this task according to the orchestration algorithm. The task will then reside in the virtual machine's queue and wait for its turn. When the task is popped from queue, it starts executing the task unless it prematurely fails. Once the execution is done and whether it is successfully executed or not, the task's result will return to the broker. The broker passes its result to the cloud node.

### 4.2.3 Virtual Machine Characteristics

We assume that tasks can be executed in nodes that have virtual machines e.g. cloud, edge data center, and edge devices. We assume that virtual machines are defined as in Subsection 2.1.2 with the following characteristics:

$$VM =< PE, RAM, Storage, MIPS > \tag{4.5}$$

where $PE$ is number of processing entities, $RAM$ and $Storage$ are short-term RAM memory (GB) and storage (GB) respectively, and $MIPS$ is how fast this virtual machine execute instructions at a time, i.e. million instructions per second.

These characteristics represent the virtual machine's share of resources. Clearly, total resources of a virtual machine in one node cannot exceed node's resources. Generated tasks are forwarded to virtual machines in nodes that will actually execute them. We assume that a virtual machine can execute one task at a time and that if a virtual machine is sent more than one task at a time, subsequent tasks are queued in the virtual machine's queue. Tasks will be executed in virtual machines, therefore we are taking virtual machine's resources in consideration instead of node's resources.

### 4.2.4 Environment Characteristics

In some simulators, all tasks are generated before the simulation starts which makes the orchestration job straightforward since quantity and rate of tasks are known beforehand. In this case, tasks are sorted based on their importance and traffic congestion is not considered. That is not the case in real systems. In dynamic task orchestration, tasks are generated while the system is running. The simulation manager has no idea about the tasks' quantity and rate and should adapt over time. We assume that tasks arrive randomly based on a Poisson distribution [67, 13]. Whenever a task is generated, it is sent to the broker node. The broker node executes the task orchestration algorithm where it allocates tasks to registered virtual machines.

In distributed systems, the load balancing problem can be addressed in two different general approaches. First, there is static load balancing. One approach for it [67] explained it as, nodes will offload their tasks to a predefined neighbour node/nodes. This could also be done in two ways. The first is a deterministic offloading mapping which assigns one neighbour node to each node for offloading in case of overloading. A second approach is probabilistic offloading

which maps multiple neighbour nodes to each node. In case of overloading, the node offloads to neighbour node *A* with probability of x and to neighbour node *B* with probability of 1-x, if it had two assigned neighbours. Nevertheless, neighbour nodes could be overloaded as well. Since current status of neighbour nodes is not taken into consideration, this load balancing type is not efficient. The second approach is dynamic load balancing which takes into consideration the status of all nodes. Therefore, tasks are offloaded from overloaded nodes to less-loaded nodes only [67]. Despite the complexity of this approach and the continuous monitoring required, it is more flexible.

In our simulation, the term ”task orchestration” is used instead of ”task offloading”, because the decision of task assignment is made by one node (broker). As described earlier, we assume that tasks are generated by the cloud node but can be executed in many nodes; the allocation of a task to a node is determined by the broker.

## 4.3 Simulation Settings

In the following, we outline the behavior of the simulator by walking through one run cycle.

### 4.3.1 Initialization

First, nodes are initialized with resources and virtual machines associated with them are created. All communication connections between nodes are established and then the broker node is created. The broker has a record of all nodes and their characteristics, connections, and registered virtual machines. Then, a list of different classes of tasks is created to be the source for tasks sampling.

This list of task classes contains delay sensitive, computationally heavy tasks and tasks that are neither delay sensitive nor heavy. Some tasks require more than one processor to execute its instructions in parallel. Others require large storage size in order to be executed. Each task class in this list has a ratio which determines its portion of total generated tasks. For example, generated tasks will be 10% of class *A*, 30% of class *B*, 40% of class *C*, and 20% of class *D*. Each type of task has a Boolean ”steadiness” value which represents whether this type of task needs steady and verified IoT readings. Steadiness means that one reading of an IoT device is not enough to serve this type of task, and multiple readings must be obtained. Lastly, there are the dependencies which is required by a class of task. Dependencies could be the sensors that a task depends on for readings and/or results of previously executed tasks. To lower the communication delays caused by requests for data from those dependencies, tasks are best located as near as possible to all of their dependencies. Results of executed tasks are saved in the nodes where they were executed for as long as the task request to be saved or for a period set by the node. Therefore, if task of class *A* depends on the result of task of class *B* and task of class *B* was executed in node *x*, then task of class *A* is best located in node *x* or in a node that is close to node *x*. Nevertheless, other types of tasks with unmentioned characters can be added with few modifications needed.

### 4.3.2 Task Generation

The next step in the simulation cycle is task generation. Tasks are generated randomly at an initialized rate at the beginning of the simulation. Simulation duration is also initialized then. These settings help with making experiments finite and repeatable. They do not change throughout cycles.

Parallel running threads best simulate a fog computing network with multiple nodes. Each node executes multiple tasks at the same time, communicates with other nodes, communicates with broker, and more at a time. Virtual machine run in parallel with each other, in order to simulate utilizing links and occupying resources of all virtual machine at the same time. Simultaneously, all virtual machines in all nodes start running, waiting for tasks to execute. The VMs execute tasks and communicate with other nodes in the case that the current running task requires an IoT device's stream of real-time data or to communicate with other nodes. When the broker determines a virtual machine to execute a task, the broker sends the task to this virtual machine's node that will in turn assign it to the virtual machine. All actions that take place in nodes and through links take time and that time is simulated by putting a thread to sleep or by a timer. Therefore, these actions actually takes time to be done. Sending and receiving tasks, requesting and responding to real-time data, forwarding requests and generating responses are actions that utilize links. When data is transferred through a link, it is sent to the designated destination after a computed communication delay. As previously noted, utilizing bandwidth in links is a cost that smart systems aim to reduce.

## 4.4 Simulation Behavior

**Algorithm 1** Main

```
1: procedure
2: Initialize OrchAlgorithms with standard Orchestration Algorithms
3: for iteration ← 1, numberOfRepetitions do
4:     for i ← 1, sizeOf(OrchAlgorithms) − 1 do
5:         resetSystem()
6:         runNodes()
7:         brokerRun(OrchAlgorithms[i], currentTime, 1000)
8:         endSystem()
9:     end for
10: end for
11: end procedure
```

Algorithm 1 presents part of the main procedure, where the run is repeated for a number of times, each run is using different kind of orchestration algorithm (the specific orchestration algorithms used for benchmarks are explained in Chapter 6). At each run, the parameters are saved for comparison purposes. At first, the broker callsthe *resetSystem* function, where the

system is reset by clearing all parameters of the broker object. After that, the function *runNodes* starts all the virtual machines in nodes to run in parallel so that they are ready to receive tasks. It is noteworthy to mention that along with parallel run of virtual machines, nodes run in parallel as well in order to run the communication system of the node. The communication system in broker receives returned tasks from all virtual machines in nodes. Then, the broker will start running based on a given orchestration algorithm, at certain start time, for a given simulation duration e.g. 1000 ms. Finally, the system kills all its virtual machines and stops running. Performance measurements are taken after this step.

**Algorithm 2** Broker Run

```
1:  procedure brokerRun(orchAlgorithm, startTime, simuDuration)
2:  task ← getNextTask()
3:  while task is not NULL and currentTime ≤ startTime + simulationDuration do
4:      chosenVirtualMachine ← directTask(task)
5:      sendTask(task, chosenVirtualMachine)
6:      task ← getNextTask()
7:  end while
8:  waitForResponds()
9:  printSuccessRatio()
10: End procedure
```

The run procedure in the broker is described by Algorithm 2. First, a task is obtained by *getNextTask* function. This function checks if any task has been received; it returns with NULL if end time is reached (*startTime* + *simulationDuration*). In most cases, tasks are queued up and this function adopts the first come first serve strategy. Then, the broker keeps on repeating the following steps as long as there is new task and until the end of the simulation duration. It receives the chosen registered virtual machine based on the given orchestration algorithm. After that it calls the procedure that takes both the task and the chosen virtual machine and sends the task to that virtual machine. In this simulator, the task is sent to the node where the chosen virtual machine is, after a calculated communication delay. This is done by a timer thread in Python [1]. This timer initiates a thread to call a function after a computed delay, once the function execution is done it terminates. For example, this timer calls a function that sends the task to a node, after the computed communication delay. Then, the broker gets the next task and repeats this loop. As mentioned earlier, the communication system of the broker is working in parallel with this run. The other thread simulate sending tasks and receiving their results. Received results in the broker are returned to the cloud node. To make sure all actions are done before the simulation ends, all timer threads created during run time are saved before ending the run. After finishing the simulation, in line 8 the broker waits for all responses from all nodes, and until number of returned tasks is the same of the directed ones . After logging history of the simulation in files, success ratio $\eta$ is calculated and printed out as in line 9. The success ratio $\eta$ is calculated as in Equation 4.6,

$$\eta = \frac{\alpha_{\mathbb{T}}}{\alpha_{\mathbb{T}} + \beta_{\mathbb{T}}} \tag{4.6}$$

where $\alpha_{\mathbb{T}}$ is the number of successfully executed tasks and $\beta_{\mathbb{T}}$ is the number of failed tasks. In this simulator, task execution could fail and accordingly be dropped for one of the following reasons,

- the chosen virtual machine for a task does not have sufficient resources. For example, when the broker follows round robin algorithm for tasks' orchestration, it focuses only on balancing the load. Neglecting the heterogeneity of tasks and/or nodes results in this kind of failure. It could also result from a weak monitoring system. As previously mentioned, tasks may require minimum size of RAM, size of storage, and number of PE, therefore if this task is orchestrated to a virtual machine with a resource (RAM, storage, or PE) less than the requested minimum, this task fails.
- the task requested a reading from an unreachable device. That is the case when a task requests reading from a device that is not connected to network for any reason.
- the maximum delay for a task is exceeded for delay sensitive tasks.

Algorithm 3 describes the run procedure in a virtual machine within the simulator. It first fetches one of the tasks that is queued in its queue by *getNextTask* function. The virtual machine is expected to run as long as it is not killed (status is active). Killing the virtual machine can be controlled only by its node, it is killed when the defined simulation duration ends. When the virtual machine is killed it stops executing tasks and returns all its resources to the node. While active, the virtual machine first checks if it has sufficient resources to execute the *task* by *suffResources* function. Second, it checks whether maximum delay tolerance for this task is exceeded or not. Third, it gets all dependencies required by the task, if any. After passing through all of these check points, it is now ready to "execute" the task. This execution delay is represented by thread sleep for that computed period of delay (computation is discussed further in the next Chapter). Then, it records time taken to execute this task by calling the function *ended* in task object. Finally, the virtual machine saves results of this task for future tasks if needed, and sends the results back to the broker. It is worth noting that instead of setting the thread to sleep, the virtual machine can call a function to actually execute the task and get actual results. This will not affect its performance.

After defining tasks' characteristics and timeline, and the environment's characteristics and dynamics, now we have an idea about the complexity of this system. Considering all details regarding tasks, nodes, etc, orchestrating tasks is a complicated problem. Reinforcement Learning orchestration algorithm may be a useful approach; we describe our algorithm in the next Chapter.

**Algorithm 3** Virtual Machine Run

```
1:  procedure vmRun()
2:  task ← getNextTask()
3:  while status is active do
4:      if task is not NULL then
5:          if suffResources(task) then
6:              if not task.maxDelayExceeded() then
7:                  if hasDependencies(task) then
8:                      respond ← NULL
9:                      respond ← getTaskDependencies(task)
10:                 end if
11:                 if not hasDependencies(task) or respond is not NULL then
12:                     processingDelay = task.length /MIPS
13:                     thread sleep for (processingDelay)
14:                     task.ended()
15:                     appendResults(task)
16:                 else
17:                     task.setFailureReason("Unreachable Dependencies")
18:                 end if
19:             else
20:                 task.setFailureReason("Maximum Delay Exceeded")
21:             end if
22:         else
23:             task.setFailureReason("Unsufficient Resources")
24:         end if
25:         sendRespond(task)
26:     else
27:         task ← getNextTask()
28:     end if
29: end while
30: End procedure
```

# Chapter 5

# Proposed RL Orchestration Agent

In this chapter, we will introduce the RL orchestration agent (algorithm) which will be responsible for orchestrating tasks to suitable nodes and virtual machines. In order to formulate this complex environment for an RL agent, the formulation should include all characteristics of the nodes, links, virtual machines and tasks. It is also important to make sure that the state space does not grow exponentially as the system grows. Like this problem, there are many applications of reinforcement learning that need feature engineering to enhance the formulation of an environment [70]. Feature engineering helps with representing raw data from the environment as higher level abstractions. It requires extra hand tuning, but can result in making the representation of the environment concise, yet contain all important features.

## 5.1 Feature Engineering

We focused on a virtual machine's characteristics in the model instead of node's characteristics because task execution depends on the status of the assigned virtual machine and not the node itself. In order to get all virtual machines' characteristics, we need to have a monitoring system in the broker. This monitoring system records virtual machines' resources (MIPS, RAM, storage, processing elements) when they register with the broker. In our simulation, nodes register their profile (ID, node type, and their resources) and virtual machines profile (resources) to the broker at the beginning of the simulation. If a node registers multiple virtual machines where those resources exceed the node's resources, then the monitoring system will raise an error. In a real systems, virtual machines resources cannot exceed its node resources. This monitoring system also updates the load on those virtual machines whenever a new task arrives. It updates the node's information on the number and length of remaining tasks in all registered virtual machines' queues, in order to monitor and update their load. Monitoring helps with recording an updated representation of the system. The following parameters are introduced to better represent tasks, virtual machines characteristics and their nodes' type as recorded by the monitoring system.

Waiting delay ($D^W$), which was mentioned previously, is essential to represent the load on the virtual machine. The recorded load of a virtual machine may change according to an

unexpected waiting time due to network traffic and other delaying factors. It primarily depends on the virtual machine's MIPS.

$$D_x^W = \sum_{i=0}^{j} \frac{tl_i}{MIPS_x} \tag{5.1}$$

where $j$ is the number of tasks in the queue of virtual machine $x$, $tl_i$ is the length of task $i$ and $MIPS_x$ is the MIPS of virtual machine $x$. Note that this parameter does not take into consideration delays due to input-output; we have adopted this as a simplification for the current work. It is a load measurement for virtual machine.

The other important parameter is the computation delay/processing delay ($D^P$), which depends on the current requested task length and the virtual machine MIPS. If a heavy task is assigned to an edge device, this value will be large and the RL agent will know its making a mistake. It is an estimated value because there maybe external CPU interruptions during the execution of this task. It is represented by,

$$D_{x,\mathbb{T}}^P = \frac{tl_{\mathbb{T}}}{MIPS_x} \tag{5.2}$$

This parameter has a value for each pair of virtual machines and tasks. Clearly, $D^P$ for task $\approx$ in virtual machines with faster processing elements is less than $D^P$ for the same task in virtual machines with slower processing entities. It is inversely proportional with virtual machine's MIPS.

The last important parameter is the communication delay ($D^C$), this delay depends on links, path, and traffic between two nodes. Passing through each link causes delay, therefore the fewer links a request is passing through, the less delay the request is facing. Typically, this delay increases as the distance between node and IoT layer increases e.g. cloud nodes have larger $D^C$ than edge nodes. In case of tasks not processing data from IoT devices in layer, this parameter is set to its lowest value.

$$LD = D^{PR} + D^T + D^Q \tag{5.3}$$

each link simulates its communication delay $LD$ by the three main components of delay. Propagation delay ($D^{PR}$) is the delay for each bit to travel in the link, it depends on the distance between the sender and the receiver [37]. It is simulated as a small constant delay value, 10ms for links connecting a cloud with other node and 5 ms for links connecting edge data centers with other nodes other than the cloud. Those propagation delay constants are the tested average delay based on [37]. Transmission delay ($D^T$) is delay for pushing all the packet bits into the link. It depends on the link's bandwidth and the packet size.

$$D^T = \frac{packetSize}{bandwidth} \tag{5.4}$$

Queuing delay ($D^Q$) is the delay that represents the links' congestion. It usually affects the link delay if the link is connecting cloud or edge data center to other nodes. Although the bandwidth of the links connected to cloud are large, the number of packets travelling from and to it may also be large. Nodes and links deployed in real systems are not exclusive for our smart

city application. Since our smart city system is not the only system using those resources, the queuing delay could increase and mainly depends on virtual machines share of network. That is because it depends on the average arrival rate, average packet size, and bandwidth. Network congestion may vary and may lead to an increase in the average arrival rate, which in turn will increase the queuing delay.

The node-to-node communication delay is formulated as follows,

$$D^{C}_{n,m} = \sum_{i=0}^{l} LD_l \tag{5.5}$$

where $D^{C}_{n,m}$ is the communication delay between node $n$ and node $m$, which is the sum of links' delays that connect $n$ and $m$. Whereas, the communication delay for a requested task (node-to-sensor) also depends on the tasks' steadiness, number of requested readings, and current link usage. It is formulated as follows,

$$D^{C}_{n,\mathbb{T}} = 2r_{\mathbb{T}}k^{s_{\mathbb{T}}} \sum_{i=0}^{l} LD_l \tag{5.6}$$

where the scalar 2 represents the request and response, and $r_{\mathbb{T}}$ represents the number of sensor readings requested by this task. While $s_{\mathbb{T}}$ represents the Boolean value of steadiness for task $\mathbb{T}$. If the task needs steadiness, then each reading is requested $k$ times for verification, $k$ is determined by the task. The links from node $n$ to the IoT devices (for acquiring readings) is requested from the broker. The broker finds the path from this node to the IoT layer and sends an array of links back to the node. This path is important to calculate the communication delay and to simulate utilizing those links in the time of sending request.

It is not practical to present the task's minimum requirements along with all registered virtual machines' characteristics for every time a task arrives. The RL orchestration agent should know how to check those characteristics at each step in order to eliminate the ones that do not have sufficient resources before making a decision. For the agent to learn how to prevent one of the failure reasons, which are presented in section 4.3. Therefore, sufficient resources parameter ($sf_{x,\mathbb{T}}$) is introduced as an indicator to indicate if a virtual machine $x$ can execute a task $\mathbb{T}$.

**Algorithm 4** Algorithm to Determine Sufficient Resources

```
1: hasSufficientResources(x, 𝕋)
2: if (x.MIPS >= 𝕋.MIPS & x.RAM >= 𝕋.RAM &
   (x.storage − x.storageCurrentUsage) >= 𝕋.storage) then
3:     sf_{x,𝕋} ← True
4: else
5:     sf_{x,𝕋} ← False
6: end if
7: return sf_{x,𝕋}
```

This algorithm takes a virtual machine $x$ and a task $\mathbb{T}$ and returns a Boolean value. Checking the current free space of virtual machine's storage is important to make sure that there is space to store the task until its turn.

The last parameter introduced indicates how close the node is to a task's dependencies. In case of an IoT reading dependency, one-hop away is the closest distance possible. Since IoT devices are assumed to have limited computing capabilities and the closest node with computing capability is an edge device. In case of a task depending on another task, the closest distance possible is zero-hops away, where another task is allocated to the same node. However, in many cases there is more than one dependency. For simplicity in this system, the node which is close to 50% or more of a task dependencies is considered as close.

The features/parameters of a virtual machine or of a pair of virtual machine and task, represent the inputs for our agent. Both $sf_{x,\mathbb{T}}$ and $cl_{x,\mathbb{T}}$ have a boolean value. The delays ($D^W_x$, $D^P_{x,\mathbb{T}}$, $D^C_{n,\mathbb{T}}$), which are floating values, will be digitized based on dynamic classes. Those classes depend on the history of records (R). Whenever a new value is recorded it is added to the array of records (R). R stores the last 100 records only, that is to discard old records. A dynamic class is a class in which the upper and lower limits change based on R. The median (m2) of R divides the array into two classes. Another median (m1) in the first sub-array and a third one (m3) in the second sub-array divides R into four classes. The first class in which the limits are minimum of R and m1 is shaded in yellow in figure 5.1. While the second, third, and forth classes are shaded in orange, green, and blue respectively. Figure 5.2 shows the new class limits after adding one element (one record).

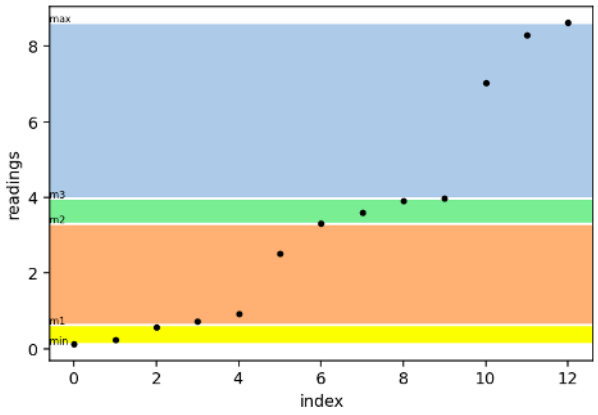


Figure 5.1: Dynamic classing for a an array with 13 elements.

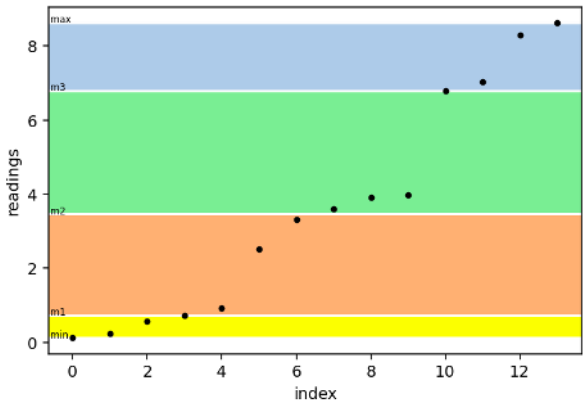


Figure 5.2: Dynamic classes after adding one element

Using this classification, the agent is able to cope with most situations. Whether the delay recorded is $10^{-4}$ or 5 seconds, the classification will differentiate between them in according to the system delay range. Dynamic classes should be better than static classes in adapting to different and updated situations. Since R is holding only last most recent records, the grouping is fitting with current status of the network. System delays will have the index of which class they belong to, instead of the actual float value of the delay. Therefore, the delay space is of

size four, which is finite. We only consider four classes in our current work which seems to be sufficiently general; having more than four classes is something to consider in the future.

To summarize, when the agent gets a task to orchestrate, it will try to match it with best virtual machine. Each candidate will have the five parameters i.e. $D_x^W$, $D_{x,t}^P$, $D_{n,t}^C$, $sf_{x,t}$ and $cl_{x,t}$. By that, all variables of the virtual machine, and some variables of the current task are included. There are two important task variables that are not yet included which are: heaviness of a task $\mathbb{T}.hv$ and/or delay sensitivity of a task $\mathbb{T}.ds$.

## 5.2 RL Components

The main components of the RL orchestration algorithm are described in this section. Designing the model for the agent is important because it directly affects the agent's learning. The reward function plays a vital role in the learning process as well.

### 5.2.1 States

At each step, the agent will get a new task to orchestrate, and updated parameters for all registered virtual machines. The agent chooses a virtual machine or this task. Although in most RL applications the agent gets one observation at each time step, it is not the case in this problem. Here, and in other optimization problems, the agent receives multi-observations and should choose one action at each time step. Therefore, the model can be described as a multi-observations single-state (MOSS) model. One observation $\phi_{x,\mathbb{T}}$ is comprised of the characteristics of one virtual machine and two task characteristics.

$$\phi_{x,\mathbb{T}} = \left\langle D_x^W, D_{x,\mathbb{T}}^P, D_{n,\mathbb{T}}^C, sf_{x,\mathbb{T}}, cl_{x,\mathbb{T}}, \mathbb{T}.hv, \mathbb{T}.ds \right\rangle \tag{5.7}$$

and a state at a step will look like,

$$s_{st} = \left[\phi_{x1,\mathbb{T}},\ \phi_{x2,\mathbb{T}},\ \phi_{x3,\mathbb{T}},\ \ldots,\ \phi_{xn,\mathbb{T}}\right] \tag{5.8}$$

where $s_{st}$ is the state at step $st$, $x_1$ is the first registered virtual machine, and $x_n$ is the last registered virtual machine. Having one state that concatenates all these observations is not practical since the state space will be of size 1024n and the size of a state will be 7n, where n is the number of registered virtual machines. The number of registered virtual machines may also change from time to time due to nodes leaving and joining the network, and the size of state cannot be dynamic. Virtual machines may be unavailable if their nodes are disconnected from the network, for example: edge device running out of energy or losing connectivity. Additionally, it is not efficient because the number of registered virtual machines can be very large, especially in large systems like smart cities. This is one of the reasons for using the MOSS model. In a MOSS model, the size of $\phi$ is 7 and the size of the observations space $\Phi$ is limited to 1024. At each step, the agent learns about observations $\phi$ rather than the state itself. Therefore, instead of having a table of Q values of pairs of state and action, we will have a table of observations $\phi$.

At each step the agent will choose one virtual machine, a virtual machine status is described in one observation $\phi$. Therefore, the agent is choosing the best observation. If this action returns the maximum reward possible, the rest of observations will be updated as non-preferable actions. As a result, n updates are done at one step. This mimics the behaviour of a cooperative multi-agent model in RL [80], where multiple updates are done at each time step. However, this model takes one action at a time. Figure 5.3 graphically represent one step in the MOSS model, where the agent receives multiple observations $\phi(n)$ at each step. The agent takes an action that affects the environment and receives a reward accordingly. Therefore, at each step, the agent receives $\phi(n)_(st)$ and $r_(st)$ and generates an action $a$.

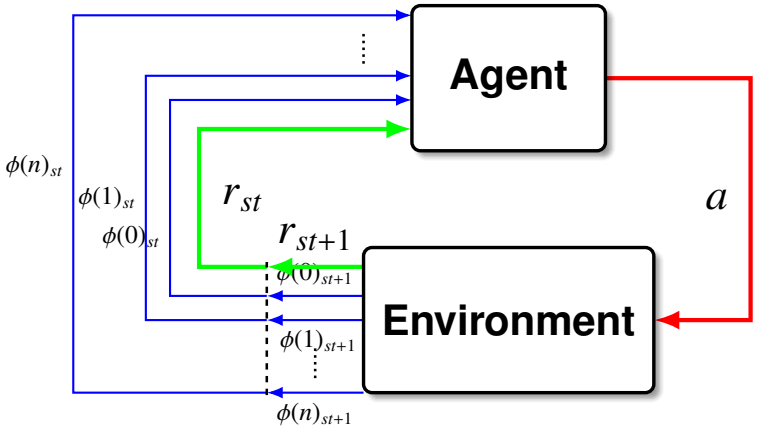


Figure 5.3: Suggested MOSS model

This algorithm derives features out of the environment's state to formulate the agent's state. As mentioned earlier, the state of the agent will be represented by multiple observations and one new task. Due to that, a step in this model depends on the arrival of a new task. The time gap between two tasks in real-systems can be 1 ms or 10 min, and therefore the agent updates the models' parameters at each step. A change in the environment which occurs due to the agent's action should be observed in order to decide the reward that should be given.

After extracting features from the new task, parameters of the environment are updated. To update those parameters, we have to check parameters of all registered virtual machines. This is the only step that depends directly on the number of virtual machines in the system, it is $O(n)$. After constructing the observation of a virtual machine, it is added to the table of observations if it is observed for the first time. It is also obvious that this new observation should be added to the Q table with initial value of 0. Once the agent updates all virtual machines' characteristics, it will return with a state that represents the whole system.

Usually states are initialized with all variations possible in the state space and in the Q table with Q value of zero. Due to the nature of the system, not all variations of characteristics in observations are valid. That is, the virtual machines with lowest waiting, communication, and computing delays, e.g. $D^W_x$, $D^C_{n,\mathbb{T}}$, and $D^P_{x,\mathbb{T}}$ in the lowest delay class, do not exist. Therefore, as the agent encounters a new observation, it will add it to the observation and Q table. Observations in the observations table are not associated with a particular virtual machine. Instead,

**Algorithm 5** get_state function pseudo code

**Function Name:** get_state()
**Takes:** task
**Returns:** state
**if** *task* **then**
  *initialize state as an empty list*
  $length \leftarrow task.getLength()$
  $hv \leftarrow task.isHeavy()$
  $ds \leftarrow task.isDelaySensitive()$
  **for** $vm \leftarrow vm_0, vm_n$ **do**
    $D^W \leftarrow vm.getWaitingDelay()$
    $updateEWDRecords(D^W)$
    $D^W \leftarrow digitize(D^W, EWDRecords)$
    $D^P \leftarrow length/vm.getMIPS()$
    $updateEPDRecords(D^P)$
    $D^P \leftarrow digitize(D^P, EPDRecords)$
    $D^C \leftarrow getCommunicationDelay(vm, task)$
    $updateECDRecords(D^C)$
    $D^C \leftarrow digitize(D^C, ECDRecords)$
    $cl \leftarrow isClose(vm.getNode(), task)$
    $sf \leftarrow haveSufficientResources(vm, task)$
    $\phi \leftarrow (D^W, D^P, D^C, cl, sf, hv, ds)$
    **if** $\phi$ *not in* $\Phi$ **then**
      *add* $\phi$ *to* $\Phi$
      *add* $\phi$ *to Q table with value = 0*
    **end if**
    *append* $\phi$ *to state*
  **end for**
  **return** *state*
**end if**
**return** None

an observation is a character of any virtual machine associated with a character of a task. The observation $\phi$ can represent the character(status) of virtual machine $A$ with task $t_1$ at step $st$, and can represent the character of virtual machine $B$ with task $t_1$ at step $st + 1$. That is the case if virtual machine $B$ at step $st + 1$ had the exact same characteristics of virtual machine $A$ at step $st$.

### 5.2.2 Actions

The agent follows the epsilon greedy policy for taking actions $\mu(a|s)$. At first, the agent takes random actions to explore the environment. At each step the exploration rate decays by a decay factor $\theta$. Gradually, the agent starts using its gained knowledge. After $\frac{1}{\theta}$ steps, the agent depends fully on its own experience. The experience of the agent lies in its Q table.

### 5.2.3 Reward

As previously mentioned, the broker is always listening to receive results of the orchestrated tasks. The agent is rewarded with a positive signal only if the task's execution succeeded. A task's execution succeeds if did not fail for any of the previously mentioned reasons, and its result was generated and sent back to the requested node successfully. Otherwise, if the task fails, the agent gets a punishment for orchestrating it to an unsuitable virtual machine. In some cases, tasks fail because of reasons beyond the agent's control. For simplicity and because our reward is positively biased, those uncommon cases are ignored. Our reward function is defined as follows:

$$r_{st} = \begin{cases} -1, & \text{if } \mathbb{T}_{st} \text{ failed.} \\ 1, & \text{if } \mathbb{T}_{st} \text{ succeeded.} \\ 10, & \text{if } \mathbb{T}_{st} \text{ succeeded and } a_{st} = A_{st} \end{cases} \tag{5.9}$$

where $\mathbb{T}_{st}$ is the task at step $st$, $a_{st}$ is the action taken by the agent and $A_{st}$ is the best action at step $st$. The agent commonly obtain a reward of 1 or a punishment of -1 after each step, therefore that will be its scale for reward-punishment. But, there are two positive signals given to the agent if the task succeeded. If the success is associated with choosing the best action, then it gets a big reward. Whenever the agent acquires a reward of 10, it will always seek for it since it is 10 times better than the good reward, 1. The best action is choosing the best observation. From the optimization perspective, it is choosing the best combination of task and executor (virtual machine). Because at each step there is one task, then the agent is choosing the most suitable virtual machine. To design the function of the best action at each step $A_{st}$, $\Omega$ is used as array of weights to scale the importance of each characteristic.

$$\Omega = [\omega_0, \omega_1, \omega_2, \omega_3] \tag{5.10}$$

The four weights are for characteristic $D^C_{n,\mathbb{T}}$, $D^P_{x,\mathbb{T}}$, $D^W_x$, and $cl_{x,\mathbb{T}}$ respectively. As for our

design of task types, the following represent the characteristic weights depending on task type.

$$\Omega = \begin{cases} [2,2,2,1], & \text{if } \mathbb{T}_{st} \text{ is delay sensitive and heavy.} \\ [1,3,2,1], & \text{if } \mathbb{T}_{st} \text{ is not delay sensitive but heavy.} \\ [3,1,2,1], & \text{if } \mathbb{T}_{st} \text{ is delay sensitive but not heavy.} \\ [1,1,1,1], & \text{if } \mathbb{T}_{st} \text{ is neither delay sensitive nor heavy.} \end{cases} \tag{5.11}$$

In using these weights, for a heavy task, the agent weights the processing time more than any other criteria. Whereas for a delay sensitive task, the agent considers the communication delay to be more important than other characteristics. In other words, the agent should consider the communication delay three times more important than the processing delay for a delay sensitive task.

$$sc_{x,\mathbb{T}} = sf_{x,\mathbb{T}} \left( \omega_0 D^C_{n,\mathbb{T}} + \omega_1 D^P_{x,\mathbb{T}} + \omega_2 D^W_x + \omega_3 cl_{x,\mathbb{T}} \right) \tag{5.12}$$

The score of each virtual machine for $\mathbb{T}$ $sc_{x,\mathbb{T}}$ is calculated by equation 5.12. If a virtual machine does not have all sufficient resources to execute $\mathbb{T}$ $sf_{x,\mathbb{T}}$, then its score will be zero. As stated earlier, $sf_{x,\mathbb{T}}$ holds a Boolean value. After discriminating the characteristics using their weights, they are added up to a final score. The higher the score, the more suitable this virtual machine is for $\mathbb{T}$.

$$Sc_{st} = [sc_{x0}, sc_{x1}, \ldots, sc_{xn}] \tag{5.13}$$

$$A_{st} = max(Sc_{st}) \tag{5.14}$$

The best action possible at step st is choosing the virtual machine with the highest score among all virtual machines. This way the agent seeks to choose the best virtual machine at all times to get the maximum reward possible.

## 5.3 Learning Process

The learning process in RL is represented by repeating the cycle of observing state, taking action, and gaining a reward accordingly. Immediately after gaining a reward, the agent updates its Q entries for the observed $\Phi$.

To gain preliminary knowledge about the environment, the agent will not learn about the value of each encountered $\phi$ at the beginning. Instead, the agent will learn only about the system delay ranges for the first five episodes. This is an important step to properly classify system delays ($D^W$, $D^C$, and $D^P$). Without it, the agent will mis-classify delays and will evaluate the $\phi$ based on wrongly initialized classifications.

As formally stated, the agent moves from step to step when a new task arrives, and the reward signal for the agent depends on the orchestrated task's result. This means that at each step the agent should be halted until the task is executed and returned. Meanwhile, new tasks could arrive and accumulate in the broker's queue waiting to be orchestrated. Due to some tasks' delay sensitivity, they will fail to execute even before orchestration if we follow this approach.

This is the motive for delaying the updating step until the end of the episode. Delaying it does not affect the learning process, yet it is necessary to keep the simulation run as it does in real

systems. Therefore, updating the Q table entries for the observed $\Phi$ can be delayed until the agent gets the feedback it needs. Delaying the update means that each step will consist of observing a state and taking action only. If task execution is finished and its result returned to the broker in time, then the update will be done in place. Otherwise, the agent will save all parameters needed to do the update later i.e. the agent will save this experience. At each step, the agent will check if a previously orchestrated task has returned to perform the update. Finally and before the episode ends, the agent will wait for all orchestrated tasks to return and will finish the delayed update step for all saved experiences. This way, the structure of the simulation will not affect the learning process.

Some components of $\phi$ depend on the agent's behaviour. For example, $D^W$ for a virtual machine might increase if the agent choose to assign a task to it. Therefore, it is a dependent component. On the other hand, other components like the next task's characteristics are independent of the agent's behaviour. Therefore, the next Q value in the update function is replaced by an average of the possible next observations excluding the independent components. The following update function is used to update the value of the chosen $\phi$,

$$Q[s[a]] \leftarrow Q[s[a]] + \alpha\,(r + \gamma\,(AvgNextQ - Q[s[a]])) \tag{5.15}$$

where $\alpha$ is the learning rate, r is the immediate reward the agent gets, and $\gamma$ is the discount factor which is used to reduce the importance of future rewards comparing to immediate reward. $s[a]$ is the chosen $\phi$ and $AvgNextQ$ is the average of the possible next observations. It is similar to update function in Q learning. Whereas the following function is used to update the omitted $\phi$s,

$$Q[\phi] \leftarrow Q[\phi] + (\alpha r) \tag{5.16}$$

where the value of the omitted $\phi$s is updated negatively (r will be -1) only if the agent choose the best option. If the chosen action is the best, then all other actions are non-preferable.

**Algorithm 6** Learning process

*initialize* $\alpha, \gamma, \epsilon, epsilon\_decay, n\_episodes,$ *and* $episodes\_score$
*randomly initialize* $D^W, D^C,$ *and* $D^P$ $classifications$
*runNodes*()
**for** $i \leftarrow 0, n\_episodes$ **do**
  $done \leftarrow False$
  $episode\_score \leftarrow 0$
  $resetSystem()$
  $task \leftarrow getNextTask()$
  $s \leftarrow get\_state()$
  $a \leftarrow choose\_action(s, \epsilon)$
  **while** *not done* **do**
    $s\_, r, done \leftarrow step(s, a)$
    **if** $i > 4$ **then**
      **if** *r is None* **then**
        $AvgNextQ \leftarrow getAvgNextQ(s\_[a])$
        $saveExp(\alpha, \gamma, task, s, a, AvgNextQ)$
      **else**
        $episode\_score \leftarrow episode\_score + r$
        $AvgNextQ \leftarrow getAvgNextQ(s\_[a])$
        $Q[s[a]] \leftarrow Q[s[a]] + \alpha\,(r + \gamma\,(AvgNextQ - Q[s[a]]))$
        **if** $r = 10$ **then**
          **for** $\phi \leftarrow s[0], s[n-1]$ **do**
            **if** $\phi$ *is not s[a]* **then**
              $Q[\phi] \leftarrow Q[\phi] + (\alpha * -1)$
            **end if**
          **end for**
        **end if**
      **end if**
    **end if**
    $s \leftarrow s\_$
    $a \leftarrow choose_action(s, \epsilon)$
    $episode\_score \leftarrow episode\_score + checkPreviousExp()$
  **end while**
  **while** *count of returned tasks != count of orchestrated tasks* **do**
    $episode\_score \leftarrow episode\_score + checkPreviousExp()$
  **end while**
  *add episode_score to episodes_score*
  $\epsilon \leftarrow \epsilon - epsilon\_decay$
  **if** $\epsilon < 0$ **then**
    $\epsilon \leftarrow 0$
  **end if**
**end for**

# Chapter 6

# Experiments

In the previous Chapters, the characteristics and dynamics of the model of our system are laid out, the approach for our Reinforcement Learning based orchestration method is presented, and the methodology for the model is explained. In this Chapter, several experiments are considered. The results of the proposed RL orchestration algorithm are compared to results from benchmark algorithms for load balancing and resource allocation.

## 6.1 Results and Discussion

Because of resource limitations for running different simulations, the experiments are small yet sufficient to demonstrate the concepts and models. In the current implementation of the simulation all virtual machines in nodes and all IoT devices are running in parallel using a thread for each. Additionally, the following experiments are run in a speed up mode. This mode reduces all delays by a speed up factor $S$, which is initialized at the beginning of the simulation. For example, a task that would normally take 2 minutes in real-time, would take 500ms in simulation with speed factor of 250. Speeding up is mainly used to reduce the time gap between task generation in the simulation. It is beneficial for experimental purposes, it helps with collecting performance measures and comparison in reasonable time.

As mentioned earlier, we consider a smart city to be a hierarchical system, therefore, number of nodes in the cloud is lower than number of nodes in the edge data centers layer. Correspondingly, the number of nodes in edge data centers layer is lower than number of nodes in the edge devices layer. This scale is respected in all the tested experiments.

All experiment were conducted on a personal computer that has an i7 processor with 4 cores. The software used was Visual Studio Code using the ANACONDA python compiler.

### 6.1.1 Experiment 1: Simple Hierarchy

In this test case, we have multiple nodes with multiple virtual machines in each of them. We also have sensors as instances of IoT devices and reading of those sensors is essential to the

completion of some tasks. In Table 6.1, the types and number of nodes are shown along with number of virtual machines in each node. The connections between those nodes are described by links in Table 6.2. Table 6.3 displays the source tasks list which is the source list from which tasks for the simulation are sampled. Each column represents one of the task characteristics, minimum requirements for task execution or this task class's ratio of total generated tasks. The characteristics represent in maximum delay this task can handle before failing, length of task, and steadiness, which is a Boolean value to indicate whether or not this task needs multiple readings for its dependencies. Minimum requirements represent the minimum resources, like number of processing entities, RAM, and storage, that the task requires. Last, the ratio represents each class's portion of the total generated tasks. It is used in all experiments. This list is passed to the cloud node which is responsible for task generation in our smart city system. Task generation depends on the task rate which is set as one of the simulation parameters, in this case it is $1x10^{-2}$. Tasks generated in a cycle also depend on each cycle's duration which is set initially to 5000 seconds (this equates to approximately 25 seconds when the speed factor is 250). The following equation shows how number of tasks is calculated based on tasks' rate and cycle duration.

$$n_T = R.D \tag{6.1}$$

where $n_T$ is the number of tasks that is expected to be generated in a single cycle. $R$ and $D$ are tasks' rate and cycle duration respectively. Table 6.4 provides the parameters used in the simulation where $R$ is set to 0.01 and $D$ is set to 5000. Thus, number of expected tasks to be generated is 50 tasks in each cycle. Although in real deployment, the system is not divided into cycles, it is essential to do that in order to analyze and compare. The time gap between tasks is determined by a Poisson random distribution with $\lambda = \frac{1}{R}$.

Table 6.1: Nodes and Virtual Machines in Experiment 1

| **Node type** | **Node IDs** (starting from) | **Number of nodes** | **Number of virtual machines** (each node) | **Total number of virtual machines** |
|---|---|---|---|---|
| cloud | 100 | 1 | 6 | 6 |
| edge data center | 200 | 1 | 4 | 4 |
| edge device | 300 | 4 | 2 | 8 |
| sensors | 400 | 5 | - | - |

Generally, benchmark algorithms are essential for evaluating any proposed algorithm. The following are some of the standard load balancing and resources allocation algorithms used by as benchmark by [56, 81, 33].

*Round Robin* is one of the famous standard load balancing algorithms typically used for cpu scheduling for multiple processes. A similar approach is used to choose a virtual machine for the next task. It assigns tasks to virtual machines by turns and in a circular order [86]. This algorithm mainly fails because it considers that all nodes have similar characteristics and can execute any task.

Table 6.2: Links Description

| **from node type** | **to node type** | **bandwidth** |
|---|---|---|
| cloud | edge data center | 70Gbps |
| edge data center | edge data center | 70Gbps |
| edge data center | edge device | 30Gbps |
| edge device | edge device | 10Gbps |
| edge device | IoT device | 10Gbps |

Table 6.3: Source Task List

| **Type** | **Max delay tolerance**(ms) | **Length** (MI) | **Min PE** | **Min RAM**(GB) | **Min storage**(GB) | **Ratio** | **Steadiness** | **Dependencies** |
|---|---|---|---|---|---|---|---|---|
| A | 1000 | 1500 | 1 | 1000 | 512 | 0.2 | No | [400,403] |
| B | 750 | 500 | 1 | 512 | 1500 | 0.3 | No | [404] |
| C | 5000 | 1000 | 1 | 265 | 2200 | 0.2 | No | [401,402,403] |
| D | 50000 | 10000 | 1 | 512 | 2300 | 0.2 | Yes | [401] |
| E | 20000 | 8000 | 2 | 512 | 2200 | 0.1 | Yes | [404,400] |

*To Cloud Only*, is an orchestration algorithm to send tasks to the cloud only; a round robin algorithm is used to constantly alternate between virtual machine in the cloud nodes. This algorithm has high chance of dropping delay sensitive tasks because of communication delays as mentioned earlier.

*To Edge Data Center Only* and *To Edge Device Only*, are two other algorithms similar to *To Edge Devices Only* algorithm by choosing one type of node constantly. *To Edge Data Center Only* algorithm has a good chance of dropping some delay sensitive tasks. Whereas *To Edge Devices Only* algorithm might fail because it takes longer time in executing complex tasks, which blocks delay sensitive tasks from being executed at its most suitable nodes in time.

*Lowest Expected Waiting Time* (lwt), is an algorithm proposed by us, that sends task to the virtual machine with the lowest expected waiting time in queue. Expected waiting time depends on the length of tasks in the virtual machine's queue and virtual machine's MIPS. Although this algorithm consider the status of nodes, it mainly fails because it does not take tasks' minimum requirements into consideration.

Each of the mentioned orchestration algorithm is used to run the system for at least 100 cycles for evaluation. Each cycle takes $D$ seconds and generates $n_T$ tasks which are orchestrated to registered virtual machines. Performance measurements are taken after each cycle (referred to as repetition in Algorithm 1, line 3 in Section 4.4). Then the system is run using a MOSS model. The result of the learning process of MOSS model is obvious in Figure 6.1. Each episode in the learning process is one cycle. The model stops learning after a thousand

Table 6.4: Cycle Parameters for Experiment 1

| Parameter | Value |
|---|---|
| $R$ | 0.01 task/sec |
| $D$ | 5000 sec |
| $n_T$ | 50 tasks |
| $\lambda$ | 100 |
| $S$ | 250 |

Table 6.5: Learning Parameters in Experiment 1

| Parameter | Value |
|---|---|
| $\alpha$ | 0.01 |
| $\gamma$ | 0.9 |
| $\epsilon$ | 1 |
| $\epsilon$-decay | 0.001 |
| no. episodes | 1000 |

episodes, then it is evaluated for 100 cycles. Those evaluation cycles start after the agent stops exploring, the agent then is using its own experience.

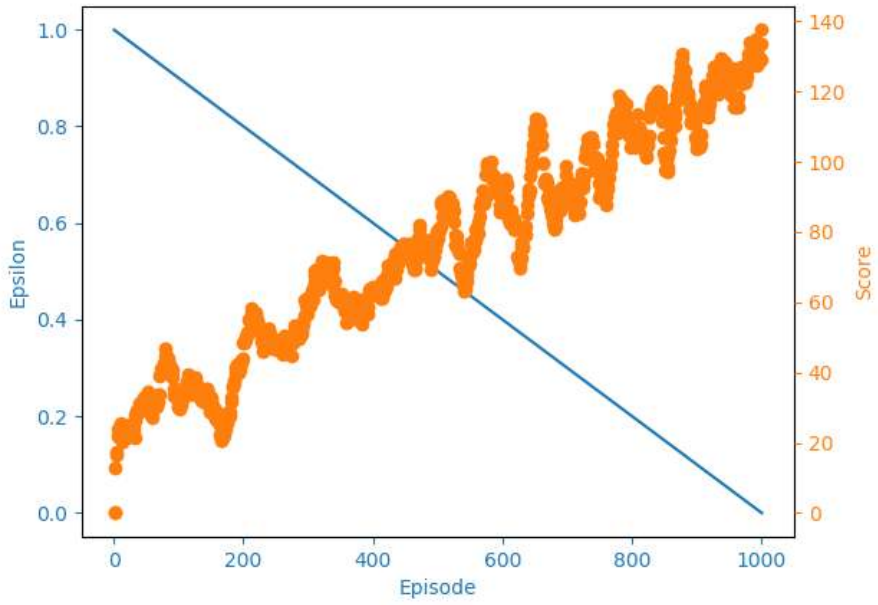


Figure 6.1: Learning process of MOSS model in simple hierarchy experiment.

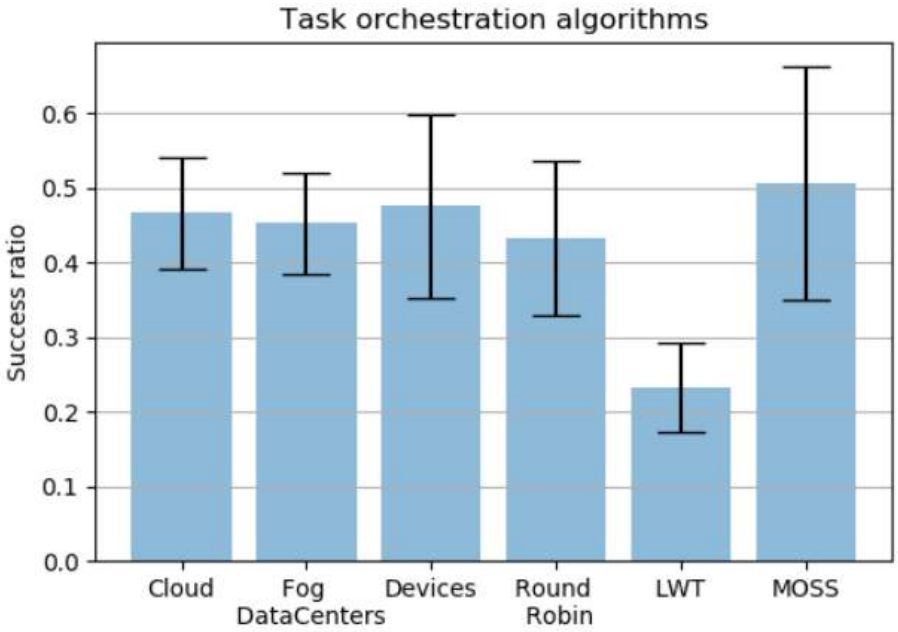


Figure 6.2: Performance comparison between benchmark algorithms and proposed model in simple hierarchy experiment.

The performance of MOSS model is increasing as the exploration rate (blue line in Figure 6.1 is decreasing. The blue line represent the epsilon value throughout cycles. Epsilon's value remains zero after learning episodes, which indicates that the agent is using its knowledge to make decision. This performance is preliminary and can be improved significantly by increasing number of learning episodes. As shown in Figure 6.1, the score gained in each episode is increasing proportionally with number of learning episodes. Meaning that, the learning did not converge or approached to a convergence point and proves that improvement for this model did not approach its end.

Figure 6.2 shows the mean of 100 cycles for each orchestration algorithm with the error range. The performance of MOSS model's 100 cycles is recorded after the learning episodes. It illustrates that MOSS model is better in orchestration than the other benchmark algorithms. MOSS model is 10% better than the best benchmark algorithm and 120% better than the worse benchmark algorithm. Such increase in the performance indicates better load balancing in the system.

### 6.1.2 Dynamic Approach

As mentioned earlier, the MOSS model can adapt the dynamic nature of this system, where nodes join and leave the network. This could be a challenge in task orchestration without a dynamic approach. Edge devices are the computing nodes which may leave the system without prior notice and rejoin later on. They may leave due to connectivity issue, running out battery, etc.. Although, IoT devices may also have connectivity disruption and battery limitations, they do not have computing entities. Therefore, such nodes are not candidates for tasks execution.

If they are disconnected for any reason, only tasks that depend on them will fail. If there is a task depending on the results of a task that failed, this task will fail as well. That is because a task with a missing dependency (whether it is a sensor reading or another task's result) cannot be executed.

The performance when running the system with cloud only algorithm for orchestration will not be affected. The same applies for running the system with Edge Data Centers only algorithm. Whereas, when using edge devices only algorithm for orchestration, the performance will drop dramatically. The reason behind that is the stability of data centers and the dynamic nature of edge devices. As shown in Figure 6.3, the success ratio dropped significantly comparing to its performance without dynamic nodes, as in Figure 6.2. The performance of the simulation with both round robin and LWT algorithm, have been affected slightly. That is because, if edge device nodes are not available, tasks will have alternative candidates for execution. Figure 6.4 shows the performance of running the system with round robin algorithm. Whereas, Figure 6.5 shows it with LWT algorithm for orchestration.

Our model receives multiple observations at a step, each observation represents a virtual machine. Regardless the number of connected virtual machines at each step, this model should always choose the most suitable machine to execute current task. Figure 6.6 represents the record of number of registered (in red) and connected virtual machines (in blue) during simulation. Nodes register their virtual machines when they join the system, and they are considered connected if they are connected to network. In this test, edge device nodes consume battery when they execute tasks. When a node's battery becomes too weak, it essential leaves the system; once it recharges, it can then rejoin. The recharge process is simulated as a 1 minute sleep and resetting the battery level to 100%. During this process, all tasks waiting in the queue of a disconnected node's virtual machines will fail and return to the broker. The success ratio is slightly affected due to the decrease of number of virtual machines. Table 6.6 includes the success ratio of the simulation using various orchestration algorithms.

Table 6.6: Performance measures with varying orchestration algorithms

| **Orchestration Algorithm** | **Success Ratio** |
|---|---|
| To Edge Devices Only | 0.42 |
| Round Robin | 0.48 |
| LWT | 0.37 |
| MOSS Model | 0.5 |

### 6.1.3 Experiment 2: Complex Hierarchy

The second experiment is larger and more complex than the first one, since it has more nodes. We have more virtual machines registered in our system as described in Table 6.7. We have also increased tasks rate, which in turn increases number of tasks generated in each cycle as in Table 6.8. The characteristics of links connecting nodes follows the same design as in Experiment 1.

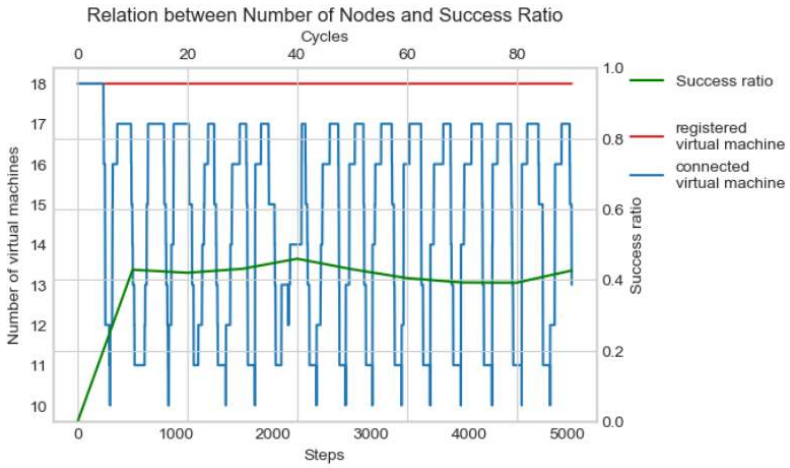


Figure 6.3: Simulation of dynamic edge devices with to devices only algorithm for tasks orchestration.

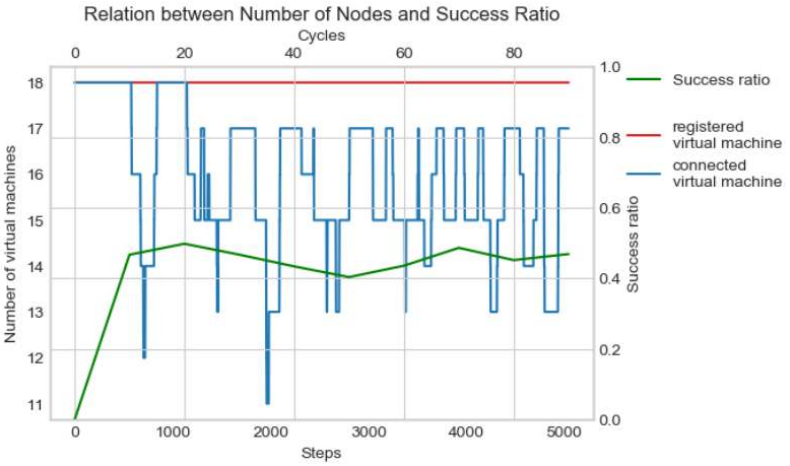


Figure 6.4: Simulation of dynamic edge devices with round robin algorithm for tasks orchestration.

This experiment also inherits source task list from experiment 1 with few modifications with the dependencies, since we have 10 sensors now.

As in experiment 1, all benchmark algorithms for tasks orchestration are tested for 100 cycles. They are tested under the same conditions (as specified in Table 6.8). Clearly, those algorithms do not need to train on the environment before testing, as they are not learning. The performance of these orchestration algorithms is clear in Figure 6.7. The success ratio of the best performing algorithm did not reach 0.4.

In this experiment, learning rate is varied along with number of learning episodes. As the learning rate decreases the agent needs more experience (more episodes) to learn well. Typically, the agent needs more episodes to converge. Due to hardware limitations, we increased number of learning episodes and decreased the epsilon decay rate to give the agent more time to learn,

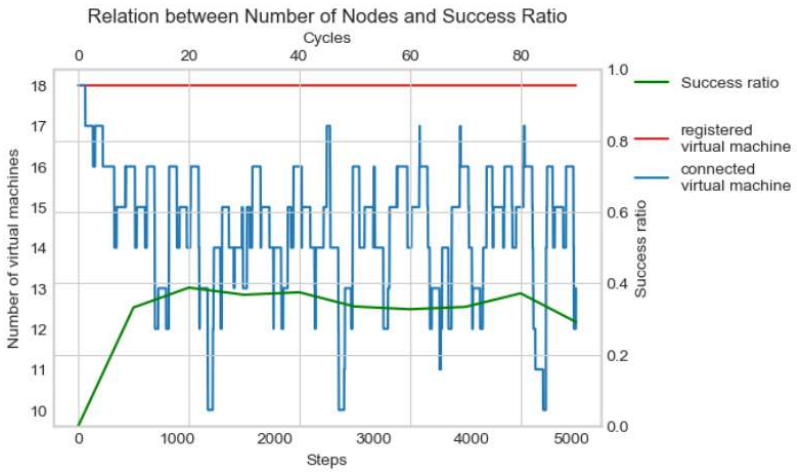


Figure 6.5: Simulation of dynamic edge devices with LWT algorithm for tasks orchestration.

Table 6.7: Nodes and Virtual Machines in Experiment 2

| **Node type** | **Node IDs** (starting from) | **Number of nodes** | **Number of virtual machines** (each node) | **Total number of virtual machines** |
|---|---|---|---|---|
| cloud | 100 | 1 | 6 | 6 |
| edge data center | 200 | 2 | 4 | 8 |
| edge device | 300 | 5 | 2 | 10 |
| sensors | 400 | 10 | - | - |

instead of waiting for it to converge.

Initially, we set the learning rate to 0.1 and number of episodes to 1000. Thus, epsilon decays by 0.001 (1/number of episodes) after each cycle. Meaning that the rate of exploring is initially 1, i.e. the agent is always taking random action to explore the environment. Then, it is decreasing by 0.001 after each cycle to increase the probability of exploiting, i.e. using its gained knowledge about the environment to take actions. Until the epsilon reaches 0, then the agent always uses its knowledge and stops taking random action. Only then, we start testing the agent's performance.

After that, we train the agent with learning rate equals to 0.05. Number of training episodes is increased to 1500 episodes to give more time for the agent to learn. Finally, we decrease the learning rate to 0.01 and increased number of episodes to 2000 training episodes as described in Table 6.9.

The three trained MOSS models have outperformed the best orchestration algorithm as in Figure 6.8. Training the agent with lower learning rate $\alpha$ for more episodes gives the agent chance to learn slowly about the environment. Then, it orchestrate tasks to their most suitable virtual

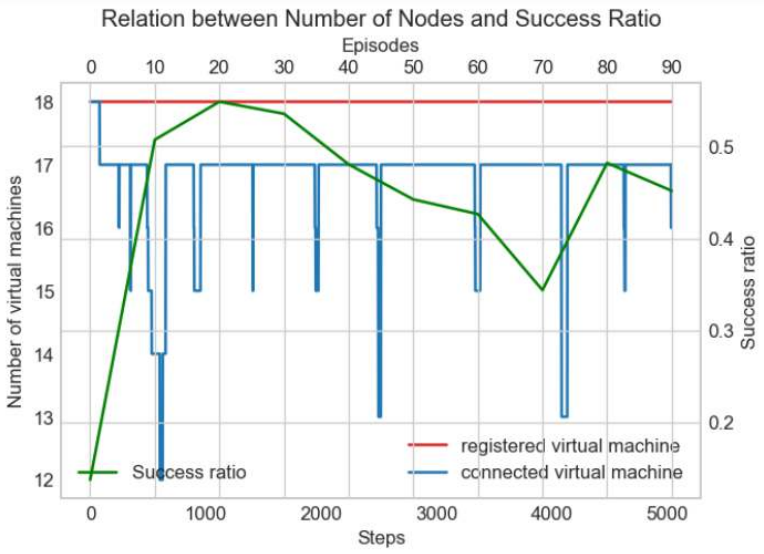


Figure 6.6: Number of connected virtual machines changing during simulation due to edge devices' instability.

Table 6.8: Cycle Parameters for Experiment 2

| Parameter | Value |
|---|---|
| $R$ | 0.02 task/sec |
| $D$ | 5000 sec |
| $n_T$ | 100 tasks |
| $\lambda$ | 50 |
| $S$ | 250 |

machines and thus manage the resources better. It also prevents machines from overloading, since it monitor and orchestrate tasks to virtual machines with lower waiting time, which corresponds to the load of that virtual machine. This model is approximately 60% better than the best orchestration algorithm.

Figure 6.9 presents the performance differences when the learning rate varies. It is obvious that the more learning episodes, the better the performance is. It is also beneficial to decrease the epsilon decay, to let the agent gradually use its knowledge rather than rushing into it. Increasing it will lead to the use of uncertain knowledge of the environment, since the agent did not explore enough observations. The learning rate could depend on the accuracy of the monitoring system. If we think that our monitoring system is not highly accurate, then it is better to lower the learning rate i.e. learn less from each experience. When the agent learns slowly, it has the chance to better formulate the relation between the input and its goal. In Figure, the accumulative average is recorded through testing cycles/episodes.

The learning process of the best performing model is recorded through the training episodes as in Figure 6.10. Throughout an episode, the agent learns how to acquire more points to increase its score. Slowly, the agent becomes greedy to gain more positive rewards, which correspond

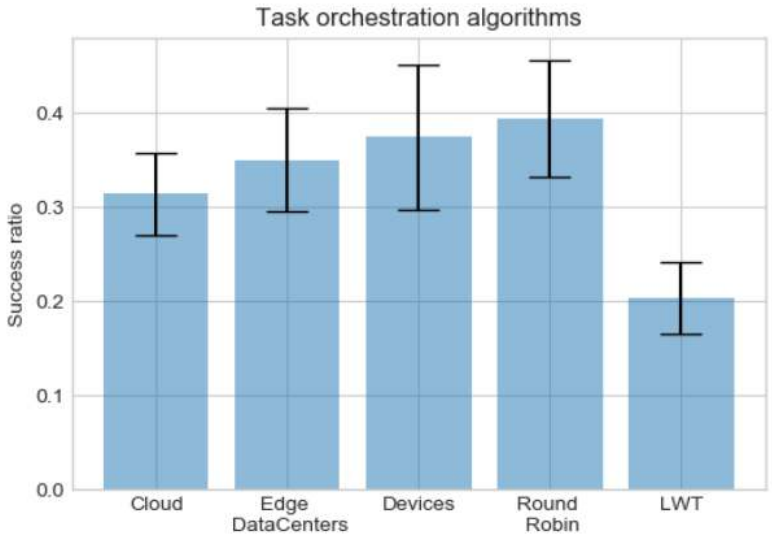


Figure 6.7: Comparison between benchmark algorithms in a complex hierarchy.

Table 6.9: Learning Parameters in Experiment 2

| **Parameter** | **Value in Run 1** | **Value in Run 2** | **Value in Run 3** |
|---|---|---|---|
| $\alpha$ | 0.1 | 0.05 | 0.01 |
| $\gamma$ | 0.9 | 0.9 | 0.9 |
| $\epsilon$ | 1 | 1 | 1 |
| $\epsilon$-decay | 1/1000 | 1/1500 | 1/2000 |
| no. episodes | 1000 | 1500 | 2000 |

to points. At the end of the training, the agent learns to obtain approximately 300 points per episode.

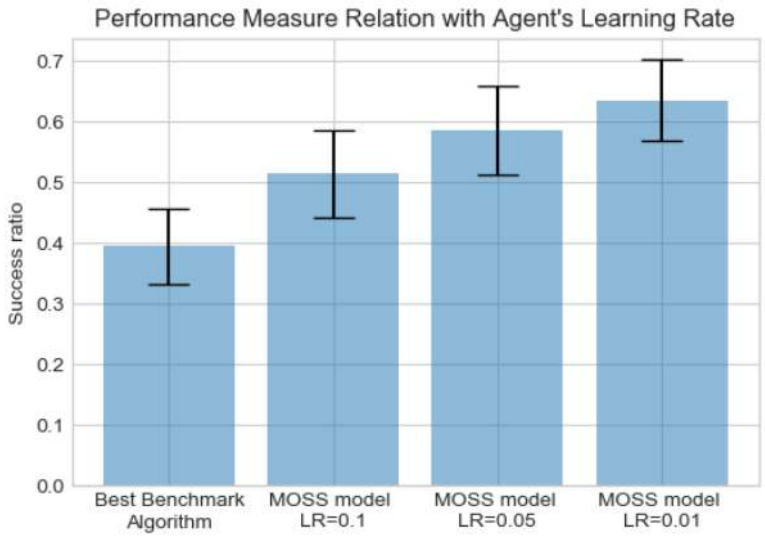


Figure 6.8: Three MOSS models outperform best benchmark algorithm for tasks orchestration.

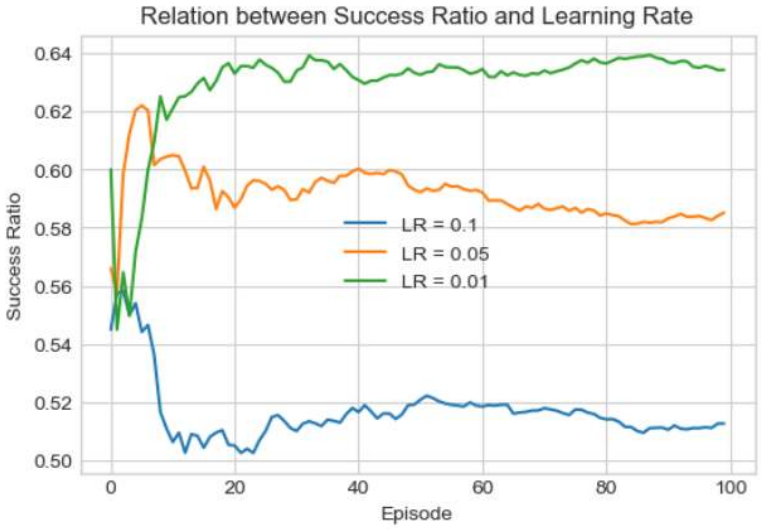


Figure 6.9: Accumulative average of success ratio in testing cycles for different learning rate.

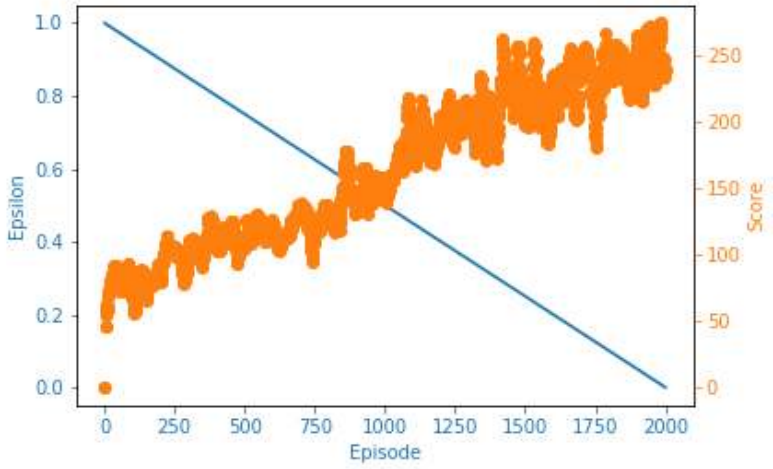


Figure 6.10: Learning process of MOSS model with learning rate 0.01 in Experiment 2.

# Chapter 7

# Conclusion

THIS thesis aimed to find a better method of allocating resources and balancing the load of task allocation in a smart city system. As previously described earlier in this thesis, there exist tremendous efforts on managing resources in distributed systems and that there are still ongoing challenges. We looked to take advantage of machine learning to improve on these methods.

## 7.1 Summary

Smart cities, like other distributed systems, are complicated in their features, dynamics, and restrictions. Despite its complexity, feature engineering was one of the helpful tools that contributed in achieving an effective load balancing and resource allocation model. It was used to best represent the system to the learning agent. The results of agent's training showed a better performance in managing the system than all other benchmark algorithms examined in our work.

In this thesis, we tackled the problem of load balancing and resources management by considering a reinforcement learning method. The reason behind choosing reinforcement learning method among all machine learning methods is its ability to capture the interaction between agent and system (agent's model). Another important aspect is that the agent is always adapting its learning parameters depending on the system. We described a smart city and then introduced the main components and dynamics of a smart city simulator. After that, we introduced a model of a smart city environment and described specific characteristics. A MOSS model was proposed to allow an agent to observe multiple observations at a step. The reward is calculated after the agent takes an action by a reward function. This reward function weighs each characteristic based on its significance in a state. The MOSS model also allows the agent to do one or multiple updates at a step. Finally, the tested experiment proved this concept and outperformed all other orchestration algorithms.

## 7.2 Contributions

The following provides a summary of the contributions of this thesis to smart city technology:

- A model of the network infrastructure of a smart city which encompassed the heterogeneity of nodes and tasks in a fog computing paradigm.
- A smart city simulator with all of its basic components. The simulator adopts a hierarchical model for nodes, similar to a fog computing network model. In this simulator, benchmark algorithms for tasks orchestration are set for comparison.
- The design of a MOSS RL model which suits the resource optimization problem. Preprocessing stage of system's features simplifies and presents the system for Reinforcement Learning orchestration agent. This model allows the agent to observe multiple observations in a single state and take an action. It also accelerates the learning process, by performing multiple updates in a step.
- A comparison of the performance of the Reinforcement Learning orchestration approach with other orchestration algorithm. Results of the comparison are analyzed and explained.

## 7.3 Limitations and Future Work

The designed MOSS model requires a single node like a broker to reserve its knowledge, which means it might suffer from a single point failure. Another requirement for the MOSS model to work is that it needs to communicate with all nodes and update their statuses whenever a new task needs orchestration. That consumes a lot of time, thus increases the cost of the system. The MOSS model also consumes the resources of the broker node, which could be used for task execution. For real life applications, the agent can use an efficient offline policy for taking actions (orchestrating) while learning to optimize its orchestration method. After that, it could start taking control gradually. Additionally, the agent should be trained with other arrival times e.g. scheduled times based on other applications with scheduled tasks.

The performance of RL might take thousands of episodes to achieve convergence which produces desired results. The results of this research demonstrated the utility of the approach, yet it can be improved significantly. This could be done by increasing the number of learning episodes and tuning learning parameters like $\alpha, \gamma, \epsilon \ldots$ etc and the weights of reward function. The agent could also lower the rate of updating the system's virtual machines' statuses, which lowers the cost of this model. The current agent does not consider congestion on the links when doing orchestration. The agent could be extended to consider load balancing on the links. Additionally, the simulator could be improved in order to accelerate the time required for each cycle to run and to improve its overall performance in order to simulate larger systems.

# Bibliography


[1] *16.2. _threading higher-level_threading_interface*. URL: https://docs.python.org/2/library/threading.html.

[2] *5G and edge computing — Why does 5G needs edge compute?* en-GB. Library Catalog: stlpartners.com. URL: https://stlpartners.com/edge-computing/5g-edge-computing/ (visited on 06/12/2020).

[3] *A Beginner's Guide to Deep Reinforcement Learning*. en. Library Catalog: pathmind.com. URL: http://pathmind.com/wiki/deep-reinforcement-learning (visited on 07/11/2020).

[4] Mahdi Abbasi, Ehsan Mohammadi Pasand, and Mohammad R. Khosravi. "Workload Allocation in IoT-Fog-Cloud Architecture Using a Multi-Objective Genetic Algorithm". en. In: *Journal of Grid Computing* 18.1 (Mar. 2020), pp. 43–56. ISSN: 1572-9184. DOI: 10.1007/s10723-020-09507-1. URL: https://doi.org/10.1007/s10723-020-09507-1 (visited on 06/30/2020).

[5] Ismet Aktaş. *Cloud and edge computing in IoT: a short history*. en-US. Library Catalog: blog.bosch-si.com. Feb. 2019. URL: https://blog.bosch-si.com/bosch-iot-suite/cloud-and-edge-computing-for-iot-a-short-history/ (visited on 07/18/2020).

[6] Taha Alfakih, Mohammad Mehedi Hassan, Abdu Gumaei, Claudio Savaglio, and Giancarlo Fortino. "Task Offloading and Resource Allocation for Mobile Edge Computing by Deep Reinforcement Learning Based on SARSA". In: *IEEE Access* 8 (2020). Conference Name: IEEE Access, pp. 54074–54084. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2981434.

[7] C. R. Anna Victoria Oikawa, Vinicius Freitas, Marcio Castro, and Laercio L. Pilla. "Adaptive Load Balancing based on Machine Learning for Iterative Parallel Applications". en. In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. Västerås, Sweden: IEEE, Mar. 2020, pp. 94–101. ISBN: 978-1-72816-582-0. DOI: 10.1109/PDP50117.2020.00021. URL: https://ieeexplore.ieee.org/document/9092148/ (visited on 07/01/2020).

[8] Leonidas G Anthopoulos and Athena Vakali. "Urban planning and smart cities: Interrelations and reciprocities". In: *The Future Internet Assembly*. Springer, Berlin, Heidelberg. 2012, pp. 178–189.

[9] J Anuradha et al. "A brief introduction on Big Data 5Vs characteristics and Hadoop technology". In: *Procedia computer science* 48 (2015), pp. 319–324.

[10] Surbhi Arora. *Supervised vs Unsupervised vs Reinforcement*. en-US. Library Catalog: www.aitude.com Section: Machine Learning. Jan. 2020. URL: https://www.aitude.com/supervised-vs-unsupervised-vs-reinforcement/ (visited on 07/11/2020).

[11] Jung-yeon Baek, Georges Kaddoum, Sahil Garg, Kuljeet Kaur, and Vivianne Gravel. "Managing Fog Networks using Reinforcement Learning Based Load Balancing Algorithm". In: *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. ISSN: 1558-2612. Apr. 2019, pp. 1–7. DOI: 10.1109/WCNC.2019.8885745.

[12] J. Bell and J. Holroyd. *Review of human reliability assessment methods, RR679*. Tech. rep. Health and Safety Executive, Londres, 2009.

[13] Roberto Beraldi, Abderrahmen Mtibaa, and Hussein Alnuweiri. "Cooperative load balancing scheme for edge computing resources". en. In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. Valencia, Spain: IEEE, May 2017, pp. 94–100. ISBN: 978-1-5386-2859-1. DOI: 10.1109/FMEC.2017.7946414. URL: http://ieeexplore.ieee.org/document/7946414/ (visited on 07/06/2020).

[14] Bonsai. *Why Reinforcement Learning Might Be the Best AI Technique for Complex Industrial Systems*. en. Nov. 2017. URL: https://medium.com/@BonsaiAI/why-reinforcement-learning-might-be-the-best-ai-technique-for-complex-industrial-systems-fde8b0ebd5fb (visited on 09/10/2020).

[15] *Bringing IoT to the Cloud: Fog Computing and Cloudlets - DZone IoT*. en. Library Catalog: dzone.com. URL: https://dzone.com/articles/bringing-iot-to-the-cloud-fog-computing-and-cloudl (visited on 07/18/2020).

[16] Jason Brownlee. *Naive Bayes for Machine Learning*. en-US. Apr. 2016. URL: https://machinelearningmastery.com/naive-bayes-for-machine-learning/ (visited on 11/14/2020).

[17] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". en. In: *Software: Practice and Experience* 41.1 (Jan. 2011), pp. 23–50. ISSN: 00380644. DOI: 10.1002/spe.995. URL: http://doi.wiley.com/10.1002/spe.995 (visited on 09/01/2020).

[18] Irene Celino and Spyros Kotoulas. "Smart Cities [Guest editors' introduction]". In: *IEEE Internet Computing* 17.6 (2013), pp. 8–11.

[19] Angelo Cenedese, Andrea Zanella, Lorenzo Vangelista, and Michele Zorzi. "Padova smart city: An urban internet of things experimentation". In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. IEEE. 2014, pp. 1–6.

[20] *Chicago ranked internationally as a top smart city - Chicago Agent Magazine Trends*. en-US. Dec. 2019. URL: https://chicagoagentmagazine.com/2019/12/09/chicago-ranked-internationally-as-a-top-smart-city/ (visited on 11/14/2020).

[21] Renata Paola Dameri. "Searching for smart city definition: a comprehensive proposal". In: *International Journal of Computers & Technology* 11.5 (2013), pp. 2544–2551.

[22] *Do Rats Have Good Eyesight?* en. Library Catalog: animals.mom.me. URL: https://animals.mom.me/rats-good-eyesight-11086.html (visited on 07/11/2020).

[23] Marco Dorigo. "Ant colony optimization". en. In: *Scholarpedia* 2.3 (Mar. 2007), p. 1461. ISSN: 1941-6016. DOI: 10.4249/scholarpedia.1461. URL: http://www.scholarpedia.org/article/Ant_colony_optimization (visited on 09/02/2020).

[24] Dzaky Zakiyal Fawwaz, Sang-Hwa Chung, and Hijoong Lee. "Dynamic IoT-Fog Task Allocation using Many-to-One Shortest Path Algorithm". en. In: *2019 IEEE International Conference on Internet of Things and Intelligence System (IoTaIS)*. BALI, Indonesia: IEEE, Nov. 2019, pp. 244–247. ISBN: 978-1-72812-516-9. DOI: 10.1109/IoTaIS47347.2019.8980395. URL: https://ieeexplore.ieee.org/document/8980395/ (visited on 06/30/2020).

[25] Rosario Ferrara. "The smart city and the green economy in Europe: A critical approach". In: *Energies* 8.6 (2015), pp. 4724–4734.

[26] Manoel C Silva Filho, Raysa L Oliveira, Claudio C Monteiro, Pedro R M Inácio, and Mário M Freire. "CloudSim Plus Documentation". en. In: (), p. 15.

[27] Pradyumna Gokhale, Omkar Bhat, and Sagar Bhat. "Introduction to IOT". In: *International Advanced Research Journal in Science, Engineering and Technology (IARJ SET)* 5.1 (2018).

[28] Nelson Mimura Gonzalez, Tereza Cristina Melo de Brito Carvalho, and Charles Christian Miers. "Cloud resource management: towards efficient execution of large-scale scientific applications and workflows on complex infrastructures". In: *Journal of Cloud Computing* 6.1 (2017), p. 13.

[29] Maria Gorini. *Smart Cities in Action: 5 Reasons Why Barcelona is a Smart City*. en. URL: https://blog.bismart.com/en/why-barcelona-is-a-smart-city (visited on 11/14/2020).

[30] Maria Gorini. *The 7 Top Smart Cities Around the World*. en. URL: https://blog.bismart.com/en/top-smart-cities-around-world (visited on 08/09/2020).

[31] Muhammad Habib ur Rehman, Prem Jayaraman, Saif Malik, Atta Khan, and Mohamed Medhat Gaber. "RedEdge: A Novel Architecture for Big Data Processing in Mobile Edge Computing Environments". en. In: *Journal of Sensor and Actuator Networks* 6.3 (Aug. 2017), p. 17. ISSN: 2224-2708. DOI: 10.3390/jsan6030017. URL: http://www.mdpi.com/2224-2708/6/3/17 (visited on 11/27/2020).

[32] Jing He, Shouling Ji, Yi Pan, and Demin Li. "Constructing Load-Balanced Data Aggregation Trees in Probabilistic Wireless Sensor Networks". In: *Parallel and Distributed Systems, IEEE Transactions on* 25 (July 2014), pp. 1681–1690. DOI: 10.1109/TPDS.2013.160.

[33] Xiuli He, Zhiyuan Ren, Chenhua Shi, and Jian Fang. "A novel load balancing strategy of software-defined cloud/fog networking in the Internet of Vehicles". en. In: *China Communications* 13.Supplement2 (2016), pp. 140–149. ISSN: 1673-5447. DOI: 10.1109/CC.2016.7833468. URL: https://ieeexplore.ieee.org/document/7833468/ (visited on 07/04/2020).

[34] Alexandre Hojda, Tharsila Maynardes Dallabona Fariniuk, Marcela de Moraes Batista Simão, Alexandre Hojda, Tharsila Maynardes Dallabona Fariniuk, and Marcela de Moraes Batista Simão. "Building a smart city with trust: the case of '156 central' of Curitiba-Brazil". en. In: *Economía, sociedad y territorio* 19.60 (Aug. 2019). Publisher: El Colegio Mexiquense A.C., pp. 79–108. ISSN: 1405-8421. DOI: 10.22136/est20191298. URL: http://www.scielo.org.mx/scielo.php?script=sci_abstract&pid=S1405-84212019000200079&lng=es&nrm=iso&tlng=en (visited on 11/14/2020).

[35] Cheol-Ho Hong and Blesson Varghese. "Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms". en. In: *ACM Computing Surveys* 52.5 (Sept. 2019), pp. 1–37. ISSN: 03600300. DOI: 10.1145/3326066. URL: http://dl.acm.org/citation.cfm?doid=3362097.3326066 (visited on 07/19/2020).

[36] Cheol-Ho Hong and Blesson Varghese. "Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms". In: *ACM Computing Surveys (CSUR)* 52.5 (2019), pp. 1–37.

[37] Ilya Grigorik. *High Performance Browser Networking*. en. O'Reilly Media, Inc., Sept. 2013. ISBN: 978-1-4493-4476-4. URL: https://www.oreilly.com/library/view/high-performance-browser/9781449344757/ch01.html (visited on 09/29/2020).

[38] *IoT Protocols: MQTT, CoAP, XMPP, SOAP, UPnP*. en. Section: Blog. Mar. 2019. URL: https://sirinsoftware.com/blog/iot-protocols-mqtt-coap-xmpp-soap-upnp/ (visited on 11/12/2020).

[39] Philip C Jackson. *Introduction to artificial intelligence*. Courier Dover Publications, 2019.

[40] Devki Nandan Jha, Khaled Alwasel, Areeb Alshoshan, Xianghua Huang, Ranesh Kumar Naha, Sudheer Kumar Battula, Saurabh Garg, Deepak Puthal, Philip James, Albert Y Zomaya, et al. "IoTSim-Edge: A Simulation Framework for Modeling the Behaviour of IoT and Edge Computing Environments". In: *arXiv preprint arXiv:1910.03026* (2019).

[41] Hongbo Jiang, Shudong Jin, and Chonggang Wang. "Prediction or Not? An Energy-Efficient Framework for Clustering-Based Data Collection in Wireless Sensor Networks". In: *IEEE Transactions on Parallel and Distributed Systems* 22.6 (June 2011), pp. 1064–1071. ISSN: 1045-9219. DOI: 10.1109/TPDS.2010.174. URL: https://doi.org/10.1109/TPDS.2010.174 (visited on 11/27/2020).

[42] Khok Hong Jing (Jingles). *Reinforcement Learning — The Value Function*. en. Library Catalog: jinglescode.github.io. June 2019. URL: https://jinglescode.github.io/2019/06/30/reinforcement-learning-value-function/ (visited on 07/10/2020).

[43] Vinay Kandpal. "A Case Study on Smart City Projects in India: An Analysis of Nagpur, Allahabad and Dehradun". en. In: *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*. Lyon, France: ACM Press, 2018, pp. 935–941. ISBN: 978-1-4503-5640-4. DOI: 10.1145/3184558.3191522. URL: http://dl.acm.org/citation.cfm?doid=3184558.3191522 (visited on 08/09/2020).

[44] Hye-Young Kim and Jong-Min Kim. "A load balancing scheme based on deep-learning in IoT". en. In: *Cluster Computing* 20.1 (Mar. 2017), pp. 873–878. ISSN: 1573-7543. DOI: 10.1007/s10586-016-0667-5. URL: https://doi.org/10.1007/s10586-016-0667-5 (visited on 06/22/2020).

[45] Mohammadreza Koopialipoor and Amin Noorbakhsh. "Applications of Artificial Intelligence Techniques in Optimizing Drilling". In: *Emerging Trends in Mechatronics*. IntechOpen, 2020.

[46] Yehia Kotb, Ismaeel Al Ridhawi, Moayad Aloqaily, Thar Baker, Yaser Jararweh, and Hissam Tawfik. "Cloud-based multi-agent cooperation for IoT devices using workflow-nets". In: *Journal of Grid Computing* 17.4 (2019), pp. 625–650.

[47] Ji Li, Hui Gao, Tiejun Lv, and Yueming Lu. "Deep reinforcement learning based computation offloading and resource allocation for MEC". In: *2018 IEEE Wireless Communications and Networking Conference (WCNC)*. ISSN: 1558-2612. Apr. 2018, pp. 1–6. DOI: 10.1109/WCNC.2018.8377343.

[48] Zhuo Li, Xu Zhou, Junruo Gao, and Yifang Qin. "SDN Controller Load Balancing Based on Reinforcement Learning". In: *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*. ISSN: 2327-0594. Nov. 2018, pp. 1120–1126. DOI: 10.1109/ICSESS.2018.8663757.

[49] Qinliang Lin, Zhibo Gong, Qiaoling Wang, and Jinlong Li. "RILNET: A Reinforcement Learning Based Load Balancing Approach for Datacenter Networks". en. In: *Machine Learning for Networking*. Ed. by Éric Renault, Paul Mühlethaler, and Selma Boumerdassi. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 44–55. ISBN: 978-3-030-19945-6. DOI: 10.1007/978-3-030-19945-6_4.

[50] M. L. Littman. "Markov Decision Processes". en. In: *International Encyclopedia of the Social & Behavioral Sciences*. Ed. by Neil J. Smelser and Paul B. Baltes. Oxford: Pergamon, Jan. 2001, pp. 9240–9242. ISBN: 978-0-08-043076-8. DOI: 10.1016/B0-08-043076-7/00614-8. URL: http://www.sciencedirect.com/science/article/pii/B0080430767006148 (visited on 07/10/2020).

[51] Zixia Liu, Hong Zhang, Bingbing Rao, and Liqiang Wang. "A Reinforcement Learning Based Resource Management Approach for Time-critical Workloads in Distributed Computing Environment". In: *2018 IEEE International Conference on Big Data (Big Data)*. Dec. 2018, pp. 252–261. DOI: 10.1109/BigData.2018.8622393.

[52] Gustavo Machado. *ML Basics: supervised, unsupervised and reinforcement learning*. en. Library Catalog: medium.com. Oct. 2016. URL: https://medium.com/@machadogj/ml-basics-supervised-unsupervised-and-reinforcement-learning-b18108487c5a (visited on 07/11/2020).

[53] Sunilkumar S Manvi and Gopal Krishna Shyam. "Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey". In: *Journal of network and computer applications* 41 (2014), pp. 424–440.

[54] Andras Markus and Attila Kertesz. "A survey and taxonomy of simulation environments modelling fog computing". en. In: *Simulation Modelling Practice and Theory* 101 (May 2020), p. 102042. ISSN: 1569190X. DOI: 10.1016/j.simpat.2019.102042. URL: https://linkinghub.elsevier.com/retrieve/pii/S1569190X1930173X (visited on 09/01/2020).

[55] 4th May 2016. *Why Chicago is a Smart City King*. en-GB. URL: https://www.iotworldtoday.com/2016/05/04/why-chicago-smart-city-king/ (visited on 11/14/2020).

[56] Charafeddine Mechalikh, Hajer Taktak, and Faouzi Moussa. "PureEdgeSim: A Simulation Toolkit for Performance Evaluation of Cloud, Fog, and Pure Edge Computing Environments". en. In: (), p. 9.

[57] Motahareh Mobasheri, Yangwoo Kim, and Woongsup Kim. "Toward Developing Fog Decision Making on the Transmission Rate of Various IoT Devices Based on Reinforcement Learning". In: *IEEE Internet of Things Magazine* 3.1 (Mar. 2020). Conference Name: IEEE Internet of Things Magazine, pp. 38–42. ISSN: 2576-3199. DOI: 10.1109/IOTM.0001.1900070.

[58] Amina Mseddi, Wael Jaafar, Halima Elbiaze, and Wessam Ajib. "Intelligent Resource Allocation in Dynamic Fog Computing Environments". In: *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*. Nov. 2019, pp. 1–7. DOI: 10.1109/CloudNet47604.2019.9064110.

[59] Tung Nguyen. *Gentle Introduction to How AWS ECS Works with Example Tutorial*. en. Oct. 2017. URL: https://medium.com/boltops/gentle-introduction-to-how-aws-ecs-works-with-example-tutorial-cea3d27ce63d (visited on 11/11/2020).

[60] Lina Ni, Jinquan Zhang, Changjun Jiang, Chungang Yan, and Kan Yu. "Resource Allocation Strategy in Fog Computing Based on Priced Timed Petri Nets". en. In: *IEEE Internet of Things Journal* 4.5 (Oct. 2017), pp. 1216–1228. ISSN: 2327-4662. DOI: 10.1109/JIOT.2017.2709814. URL: http://ieeexplore.ieee.org/document/7935527/ (visited on 07/07/2020).

[61] Bin Ning, Tao Tang, Hairong Dong, Ding Wen, Derong Liu, Shigen Gao, and Jing Wang. "An Introduction to Parallel Control and Management for High-Speed Railway Systems". en. In: *IEEE Transactions on Intelligent Transportation Systems* 12.4 (Dec. 2011), pp. 1473–1483. ISSN: 1524-9050. DOI: 10.1109/TITS.2011.2159789. URL: http://ieeexplore.ieee.org/document/5967915/ (visited on 09/02/2020).

[62] Song Ningning, Gong Chao, An Xingshuo, and Zhan Qiang. "Fog computing dynamic load balancing mechanism based on graph repartitioning". In: *China Communications* 13.3 (Mar. 2016), pp. 156–164. ISSN: 1673-5447. DOI: 10.1109/CC.2016.7445510. URL: http://ieeexplore.ieee.org/document/7445510/ (visited on 07/01/2020).

[63] Soraia Oueida, Yehia Kotb, Moayad Aloqaily, Yaser Jararweh, and Thar Baker. "An edge computing based smart healthcare framework for resource management". In: *Sensors* 18.12 (2018), p. 4307.

[64] *Overview of Supervised Learning model SVM (support vector machines) — by Hakob Avjyan — Towards Data Science*. URL: https://towardsdatascience.com/overview-of-supervised-learning-model-svm-support-vector-machines-20b683a4eaf (visited on 11/14/2020).

[65] Riccardo Petrolo, Valeria Loscrí, and Nathalie Mitton. "Towards a smart city based on cloud of things". en. In: *Proceedings of the 2014 ACM international workshop on Wireless and mobile technologies for smart cities - WiMobCity '14*. Philadelphia, Pennsylvania, USA: ACM Press, 2014, pp. 61–66. ISBN: 978-1-4503-3036-7. DOI: 10.1145/2633661.2633667. URL: http://dl.acm.org/citation.cfm?doid=2633661.2633667 (visited on 08/09/2020).

[66] M. J. Potosnak, P. Banerjee, M. B. Berkelhammer, R. Sankaran, V. R. Kotamarthi, R. L. Jacob, P. H. Beckman, S. Shahkarami, D. E. Horton, A. Montgomery, and C. E. Catlett. "Array of Things: A high-density, urban deployment of low-cost air quality sensors". In: *AGU Fall Meeting Abstracts*. Vol. 2019. Dec. 2019, A24G–04.

[67] Deepak Puthal, Mohammad S. Obaidat, Priyadarsi Nanda, Mukesh Prasad, Saraju P. Mohanty, and Albert Y. Zomaya. "Secure and Sustainable Load Balancing of Edge Data Centers in Fog Computing". en. In: *IEEE Communications Magazine* 56.5 (May 2018), pp. 60–65. ISSN: 0163-6804. DOI: 10.1109/MCOM.2018.1700795. URL: https://ieeexplore.ieee.org/document/8360851/ (visited on 07/04/2020).

[68] *Q-Learning Explained - A Reinforcement Learning Technique*. URL: https://deeplizard.com/learn/video/qhRNvCVVJaA.

[69] Steve Ranger. *What is the IoT? Everything you need to know about the Internet of Things right now*. en. URL: https://www.zdnet.com/article/what-is-the-internet-of-things-everything-you-need-to-know-about-the-iot-right-now/ (visited on 09/02/2020).

[70] Emre Rençberoğlu. *Fundamental Techniques of Feature Engineering for Machine Learning*. Apr. 2019. URL: https://towardsdatascience.com/feature-engineering-for-machine-learning-3a5e293a5114.

[71] C. I. O. Review. *Looking at a Smart City Deployment Model*. en. Library Catalog: smartcity.cioreview.com. URL: https://smartcity.cioreview.com/cxoinsight/looking-at-a-smart-city-deployment-model-nid-24109-cid-134.html (visited on 08/09/2020).

[72] Hannah Ritchie and Max Roser. "Urbanization". In: *Our World in Data* (June 2018). URL: https://ourworldindata.org/urbanization (visited on 08/08/2020).

[73] Maria Salama, Yehia Elkhatib, and Gordon Blair. "IoTNetSim: A modelling and simulation platform for end-to-end IoT services and networking". In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. 2019, pp. 251–261.

[74] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The hadoop distributed file system". In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10.

[75] David Silver. *Introduction to reinforcement learning*. 2015. URL: https://www.youtube.com/playlist?list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ.

[76] *Supervised Learning: Basics of Linear Regression — by Victor Roman — Towards Data Science*. URL: https://towardsdatascience.com/supervised-learning-basics-of-linear-regression-1cbab48d0eba (visited on 11/14/2020).

[77] Csaba Szepesvári. "Algorithms for Reinforcement Learning". en. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 4.1 (Jan. 2010), pp. 1–103. ISSN: 1939-4608, 1939-4616. DOI: 10.2200/S00268ED1V01Y201005AIM009. URL: http://www.morganclaypool.com/doi/abs/10.2200/S00268ED1V01Y201005AIM009 (visited on 06/12/2020).

[78] Phil Tabor. *philtabor/Youtube-Code-Repository*. URL: https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/Fundamentals/cartpole_qlearning.py.

[79] Saber Talari, Miadreza Shafie-Khah, Pierluigi Siano, Vincenzo Loia, Aurelio Tommasetti, and João PS Catalão. "A review of smart cities based on the internet of things concept". In: *Energies* 10.4 (2017), p. 421.

[80] Ming Tan. "Multi-agent reinforcement learning: Independent vs. cooperative agents". In: *Proceedings of the tenth international conference on machine learning*. 1993, pp. 330–337.

[81] Mohit Taneja and Alan Davy. "Resource aware placement of IoT application modules in Fog-Cloud Computing Paradigm". en. In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. Lisbon, Portugal: IEEE, May 2017, pp. 1222–1228. ISBN: 978-3-901882-89-0. DOI: 10.23919/INM.2017.7987464. URL: http://ieeexplore.ieee.org/document/7987464/ (visited on 07/07/2020).

[82] J. Tang, Z. Zhou, J. Niu, and Q. Wang. "EGF-Tree: An Energy Efficient Index Tree for Facilitating Multi-region Query Aggregation in the Internet of Things". In: *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. Aug. 2013, pp. 370–377. DOI: 10.1109/GreenCom-iThings-CPSCom.2013.81.

[83] *The Internet of Things — McKinsey*. URL: https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-internet-of-things (visited on 08/08/2020).

[84] *Unsupervised Learning and Data Clustering — by Sanatan Mishra — Towards Data Science*. URL: https://towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a (visited on 11/14/2020).

[85] *Unsupervised Learning: Dimensionality Reduction — by Victor Roman — Towards Data Science*. URL: https://towardsdatascience.com/unsupervised-learning-dimensionality-reduction-ddb4d55e0757 (visited on 11/14/2020).

[86] *Using Round Robin for Simple Load Balancing*. en-US. URL: https://www.nginx.com/resources/glossary/round-robin-load-balancing/ (visited on 09/22/2020).

[87] Shiqiang Wang, Rahul Urgaonkar, Kevin Chan, Ting He, Murtaza Zafer, and Kin K. Leung. "Dynamic service placement for mobile micro-clouds with predicted future costs". en. In: *2015 IEEE International Conference on Communications (ICC)*. London: IEEE, June 2015, pp. 5504–5510. ISBN: 978-1-4673-6432-4. DOI: 10.1109/ICC.2015.7249199. URL: http://ieeexplore.ieee.org/document/7249199/ (visited on 07/07/2020).

[88] *What is Edge Computing? Introduction to Edge Computing*. en-GB. URL: https://stlpartners.com/edge-computing/what-is-edge-computing/ (visited on 09/02/2020).

[89] *What is machine learning?* en. Library Catalog: www.technologyreview.com. URL: https://www.technologyreview.com/2018/11/17/103781/what-is-machine-learning-we-drew-you-another-flowchart/ (visited on 07/10/2020).

[90] *Why it is called Internet of Things: Definition, history, disambiguation*. en-US. Library Catalog: iot-analytics.com. URL: https://iot-analytics.com/internet-of-things-definition/ (visited on 08/08/2020).

[91] Xiaolong Xu, Shucun Fu, Qing Cai, Wei Tian, Wenjie Liu, Wanchun Dou, Xingming Sun, and Alex X. Liu. "Dynamic Resource Allocation for Load Balancing in Fog Environment". en. In: *Wireless Communications and Mobile Computing* 2018 (2018), pp. 1–15. ISSN: 1530-8669, 1530-8677. DOI: 10.1155/2018/6421607. URL: https://www.hindawi.com/journals/wcmc/2018/6421607/ (visited on 06/30/2020).

[92] Qiao Yan, F Richard Yu, Qingxiang Gong, and Jianqiang Li. "Software-defined networking (SDN) and distributed denial of service (DDoS) attacks in cloud computing environments: A survey, some research issues, and challenges". In: *IEEE communications surveys & tutorials* 18.1 (2015), pp. 602–622.

[93] Shanhe Yi, Cheng Li, and Qun Li. "A survey of fog computing: concepts, applications and issues". In: *Proceedings of the 2015 workshop on mobile big data*. 2015, pp. 37–42.

[94] F. Yuan, Y. Zhan, and Y. Wang. "Data Density Correlation Degree Clustering Method for Data Aggregation in WSN". In: *IEEE Sensors Journal* 14.4 (Apr. 2014). Conference Name: IEEE Sensors Journal, pp. 1089–1098. ISSN: 1558-1748. DOI: 10.1109/JSEN.2013.2293093.

# Curriculum Vitae

| | |
|---|---|
| **Name:** | Aseel AlOrbani |
| **Post-Secondary Education and Degrees:** | American University of the Middle East<br>kuwait, Kuwait<br>2015 - 2019 BSc.<br><br>University of Western Ontario<br>London, ON<br>2019 - 2021 MSc. |
| **Honours and Awards:** | AUM Scholarship<br>2015-2019<br><br>Western Graduate Research Scholarship (WGRS)<br>2019-2020 |
| **Related Work Experience:** | Teaching Assistant<br>The University of Western Ontario<br>2019 - 2020 |